QEMU/CPC: Static Analysis and CPS Conversion for Safe, Portable, and Efficient Coroutines

Gabriel Kerneis Charlie Shepherd
University of Cambridge
{gk338,cs648}@cam.ac.uk

Stefan Hajnoczi
Red Hat
stefanha@redhat.com

Abstract

Coroutines and events are two common abstractions for writing concurrent programs. Because coroutines are often more convenient, but events more portable and efficient, it is natural to want to translate the former into the latter. CPC is such a source-to-source translator for C programs, based on a partial conversion into continuation-passing style (CPS conversion) of functions annotated as cooperative.

In this article, we study the application of the CPC translator to QEMU, an open-source machine emulator which also uses annotated coroutine functions for concurrency. We first propose a new type of annotations to identify functions which never cooperate, and we introduce CoroCheck, a tool for the static analysis and inference of cooperation annotations. Then, we improve the CPC translator, defining CPS conversion as a calling convention for the C language, with support for indirect calls to CPS-converted function through function pointers. Finally, we apply CoroCheck and CPC to QEMU (750 000 lines of C code), fixing hundreds of missing annotations and comparing performance of the translated code with existing implementations of coroutines in QEMU.

Our work shows the importance of static annotation checking to prevent actual concurrency bugs, and demonstrates that CPS conversion is a flexible, portable, and efficient compilation technique, even for very large programs written in an imperative language.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Coroutines; CPS conversion; static analysis.

1. Introduction

Most computer programs are *concurrent* programs, which need to perform several tasks at the same time. For example, a network server needs to serve multiple clients at a time; a GUI needs to handle multiple keyboard and mouse inputs; and a network program with a graphical interface (e.g. a virtual machine with an emulated network card) needs to do both simultaneously.

Threads and events There are many different techniques to implement concurrent programs. A very common abstraction is provided by *threads*. In a threaded program, concurrent tasks are executed by a number of independent threads which communicate through a shared memory heap. Threads are generally either *native threads*, preemptively scheduled by the Operating System (OS), or *user-space threads*, cooperatively scheduled by a library.

An alternative to threads is *event-driven* programming. An event-driven program interacts with its environment by reacting to a set of stimuli called *events*. At any given point in time, to every event is associated a piece of code known as the *handler* for this event. A global scheduler, known as the *event loop*, repeatedly waits for an event to occur and invokes the associated handler. Performing a complex task requires to coordinate several event handlers by exchanging appropriate events.

Unlike threads, event handlers do not have an associated stack; event-driven programs are therefore more lightweight and often faster than their threaded counterparts [33, 44]. They are also more portable than native threads, because they do not require OS support for task switching. However, because it splits the flow of control into multiple tiny event handlers, event-driven programming is generally deemed more difficult and error-prone [5], in particular in imperative languages such as C with no support for closures and first-class functions. Additionally, because of its cooperative nature, event-driven programming alone is often not powerful enough, in particular when accessing blocking APIs or using multiple processor cores; it is then necessary to write *hybrid* code, that uses both native threads and event handlers, which is even more difficult.

Continuation-Passing C Since event-driven programming is more difficult but more efficient than threaded programming, it is natural to want to at least partially automate it. *Continuation-Passing C* (CPC [24]) is an extension of the C programming language for writing concurrent systems, built on top of the C Intermediate Language (CIL) framework [32]. The CPC programmer manipulates very lightweight threads, annotating cooperative functions and choosing whether they should be cooperatively or preemptively scheduled at any given point. The CPC program is then compiled in two steps: it is first processed by the *CPC translator*, which produces highly-efficient sequentialized event-driven code, and then linked with the *CPC runtime*, a small optimised C library scheduling the continuations introduced by the CPC translator. The translation from annotated cooperative functions into events is performed by a series of classical source-to-source program transformations: splitting of the control flow into mutually recursive nested functions, lambda lifting of these functions, and CPS conversion. This approach retains the best of both worlds: the relative convenience of programming with threads, and the low memory usage of event-loop code.

The QEMU emulator *Quick EMUlator* (QEMU [7]) is an open-source machine emulator and virtualizer that supports 16 CPU architectures and many individual devices including network cards, storage controllers, and graphics cards. Code execution is done either through dynamic translation, or through hardware virtualization support available in modern x86 CPUs. It is a large and complex project: 750 000 lines of code written by 645 contributors over more than 10 years.

Running a guest system with QEMU involves executing guest code, handling timers, processing I/O, and responding to the management console. Performing all these tasks at once requires an architecture capable of mediating resources in a safe way without pausing guest execution if a disk I/O or a command from the management console takes a long time to complete. QEMU uses a hybrid architecture that combines event-driven programming with threads.

To simplify the event-driven part, QEMU uses *coroutines*. Coroutines are an old control abstraction, commonly characterised by the ability to resume and suspend execution, as well to preserve the values of local data between successive calls. Among the many existing styles of coroutines [14], QEMU implements first-class, stackful, asymmetric coroutines, with no value passing upon suspension and resumption. As a result, programming with QEMU coroutines feels a lot like programming with threads, with the exception that cooperating passes control back to the parent coroutine instead of some global scheduler. Each coroutine is responsible for registering itself with the main event loop before yielding.

Similarly to CPC, QEMU functions that should be executed within coroutines are annotated. These annotations are not intended to drive a source-to-source transformation with static checking, but they are an essential piece of documentation for developers to write correct coroutine code. However, they are never statically checked.

Coroutines in QEMU are implemented in a platform-dependent manner. There are currently two classes of coroutine implementations: *stack-switching backends*, which allocate a new stack for each coroutine and perform a context switch when entering and yielding; and *thread-based backends*, which create a new thread for each coroutine and use synchronization primitives to ensure that only one coroutine runs at a given time and that control is transferred in the correct order. The former use non-portable functions, such as `sigaltstack` and `swapcontext` on Unix, and `SwitchToFiber` on Windows: the context switch is triggered from user-space, and in most cases involves only a cheap function call to set the current stack pointer. The latter is built on top of GLib's threads, a more portable but slower approach.

These approaches can be made to work well but require maintenance and a relatively high porting effort when targeting new platforms. Consequently, new QEMU ports sometimes only use the slower thread-based backends. The stack-switching backends require in-depth knowledge of the CPU architecture and low-level support for switching the runtime stack and process context such as signal masks.

The maintenance cost of coroutine backends, and the lack of static verification of coroutine annotations, create a need for a new approach that offers safety guarantees, and good portability with no performance degradation.

Our approach in this paper is to process the whole QEMU source code with the CPC translator in order to convert it to continuation-passing style, and then link the resulting code with a new, portable coroutine backend for QEMU based on continuations. Therefore, we reuse the CPC translator, but not the original CPC runtime: the point of our work is indeed not to rewrite QEMU in the CPC language, but to use the CPC compilation technique to implement the coroutine API defined by QEMU.

Overview of the paper

This article is a case study in applying CPS conversion and static analysis on a large C program, to implement safe, portable and efficient coroutines.

Contributions We make the following contributions:

1. a new type of annotations to mark blocking functions;
2. CoroCheck, a tool for the static analysis and inference of cooperation and blocking annotations, used to rectify hundreds of annotations in QEMU, a real-world open-source project of over 750 000 lines of C code;
3. a performance comparison of continuation-based coroutines to several other existing implementations of coroutines for QEMU.

Our work shows the importance of static annotation checking to prevent actual concurrency bugs, and demonstrates that CPS conversion — as implemented by the CPC translator — is a flexible, portable, and efficient compilation technique, even for very large programs written in an imperative language.

Outline We first give an overview of related work (Section 2), and QEMU coroutines (Section 3). Then, we introduce CoroCheck, our tool for the static analysis and inference of coroutine annotations (Section 4). Next, we present the CPC transformation technique (Section 5), and dive into the challenges associated with applying CPS conversion to QEMU (Section 6). Finally, we evaluate performance results (Section 7), and conclude (Section 8).

Timeline The work described has been carried out over the course of three months, in the context of a Google Summer of Code project. The second author, who had no prior knowledge of either QEMU or CPC, is the sponsored student. The first and third authors co-mentored his work, respectively for the CPC and QEMU sides of the project. The timeline went roughly as follows:

Before the begining of the project The first author spent two weeks improving CIL (Section 6.1) and CPC (Section 6.2).

First month The second author studied the implementation of the CPC translator and runtime, QEMU coroutines, and implemented a prototype of the `coroutine-cpc` backend (Section 6.3).

Second month The second author started adding missing coroutine annotations in the block layer (Section 4.1), to compile two small, stand-alone utilities provided by QEMU (`qemu-img` and `qemu-io`). To support his refactoring effort, the first author spent a few days writing a first prototype of CoroCheck, a tool for the static analysis of coroutine annotations (Section 4.2), and two more weeks adding a plug-in mechanism to CIL and rewriting CoroCheck as a CIL plug-in.

Third month The second author published two series of patches adding coroutine annotations to QEMU, and improved them with the help of the third author and other QEMU developers. After some minor clean-up and optimisations, the first author was able to run a virtual machine using QEMU with a `coroutine-cpc` backend, and perform micro- and macro-benchmarks (Section 7).

Software availability The code developed as part of this work is available as free software. Whenever possible, the changes have been integrated directly in the original projects that we had to adapt (CIL [1], CPC [3], QEMU [4]). Our new tool CoroCheck is also available in its own repository [2]. Some patches for the CPC backend of QEMU[1] are still pending review by the QEMU team at the time of writing.

[1] Available online at `http://github.com/kerneis/qemu/`.

2. Related work

Continuations and concurrency Delimited continuations are the general abstraction to think of threads, events and coroutines. A delimited continuation can be realized in many ways: as a stack implicitly associated with a thread, as an explicitly copied part of the stack, as a sequence of activation frames stored on heap or in a data structure (as happens in continuation-passing style), or as a closure (an event handler). In functional languages, thread-like primitives are commonly built either on top of first-class (delimited) continuations, or encapsulated within a continuation monad.

The former approach is best illustrated by Concurrent ML constructs [35], implemented on top of SML/NJ's first-class continuations, or by the way coroutines are typically implemented in Scheme using the `call/cc` operator [20]. More recently, Scala uses first-class delimited continuations to implement concurrency primitives [18, 36]. Anton and Thiemann build pure OCaml coroutines [6] on top of Kiselyov's delimcc library for delimited continuations [27].

Explicit translation into continuation-passing style, often encapsulated within a monad, is used in languages lacking first-class continuations. In Haskell, Claessen proposes a monad transformer yielding a concurrent version of existing monads [10]. Li and Zdancewic also use a monadic approach to build event-driven network servers [29]. In OCaml, Vouillon's Lwt [43] provides a lightweight alternative to native threads. The asynchronous model in F# is implemented with a localized continuation-passing translation of control-flow and a heap-based allocation of the closures, using three continuations for success, exceptions and cancellation [41].

From threaded to event-driven style In imperative languages, first-class continuations are generally not available and monadic style extremely inconvenient. This makes program transformation techniques more widespread, with two main approaches: translating loops and gotos into state machines [8], or converting functions into continuation-passing style [34, 40].

Deriving state-machines from a threaded-style code is as old as *Duff's device* [15]. Implementations have then been improved in multiple directions: as C preprocessor macros [16], as source-to-source transformations on C++ [28] or Java [17] programs, as a transformation on JVM bytecode [38], or as LLVM code blocks and macros based on GCC's nested functions [19].

CPS conversion for imperative languages is less common, probably because it is harder to implement and prove correct. CPS conversion has been applied at least to C [26], C++ [30], and Javascript [31]. To the best of our knowledge, CPC is the only public implementation for the C language, as well as the only one using lambda-lifting to avoid the runtime overhead of environments [24].

The main downside of these program transformation techniques is that CPS conversion changes function signatures, which makes it harder to mix concurrent functions with external libraries expecting callbacks. Unsurprisingly, similar issues arise when using events and threads simultaneously; Adya et al. show how to use adaptors to connect both styles [5].

Static verification of real-world programs The constraint that only cooperative functions can call cooperative functions is a very natural and common one in concurrent systems. In functional languages with a static type-checking, it is generally enforced by the monadic structure or the type system itself [10, 36, 43]. Interestingly enough, authors of similar systems for imperative languages commonly acknowledge that static checking would be preferable, but do not implement it [5, 19]. It seems that Kilim statically checks `@pausable` annotations, although the authors do not mention it explicitly [38].

There is a long history of static analysis to enforce safety properties, in particular for real-world programs written in languages lacking a strong type system [12, 13]. However, most of them require to add explicit annotations in ad-hoc domain-specific languages. A noteworthy exception is Dialyzer, a static analyser reusing the annotation format already found in the documentation of many Erlang programs [37].

3. QEMU coroutines

As described in Section 1, QEMU is an open source machine emulator and virtualizer using a hybrid architecture that combines event-driven programming with threads. The use of threads mitigates two well-known limitations of event-driven architectures: an event loop cannot take advantage of multiple cores because it only has a single thread of execution; and long-running computations or blocking system calls freeze every task in the event loop, not only the current one. Nevertheless, the core of QEMU is event-driven and most code executes in that environment.

The main event loop is executed by a dedicated thread, called *iothread*. When a file descriptor becomes ready, or when a timer expires, it invokes a callback that responds to the event. On the other hand, guest code is executed by a number of *vcpu* threads, one per virtual (emulated) CPU. In addition, *worker* threads are also used to offload blocking operations outside of the main loop.

In 2011, in response to an increase of complexity in asynchronous code, QEMU developers began to use coroutines to run concurrent tasks in the iothread without splitting them into individual callback functions [45]. A coroutine has its own stack and is therefore able to preserve state across blocking operations, which traditionally require callback functions and manual marshalling of parameters. Coroutines are now used heavily in the *block layer*, the subsystem that provides access to disk image files and supports background operations like live storage migration. Coroutines allow tasks requiring multiple disk updates to be expressed as sequential code rather than breaking them (in event-driven style) into many functions and explicitly passing on local variables.

3.1 Coroutine API

QEMU is written in C. Since the C programming language does not include support for coroutines, QEMU uses its own implementation that is based on two annotations, two type definitions and five functions (Figure 1).

```
#define coroutine_fn  /* implementation-dependent */
#define blocking_fn   /* implementation-dependent */
typedef struct Coroutine Coroutine;
typedef void coroutine_fn CoroutineEntry(void *);
Coroutine *qemu_coroutine_create(CoroutineEntry *);
void qemu_coroutine_enter (Coroutine *, void *);
void coroutine_fn qemu_coroutine_yield(void);
bool qemu_in_coroutine(void);
Coroutine * coroutine_fn qemu_coroutine_self(void);
```

Figure 1. Coroutine interface in QEMU

Creating and starting a coroutine is very straightforward:

```
coroutine = qemu_coroutine_create(my_coroutine);
qemu_coroutine_enter(coroutine, my_data);
```

The function `qemu_coroutine_create` takes an entry function that will be run inside a new coroutine, and returns a pointer to an opaque structure `Coroutine`, or *coroutine handler*. The entry function must be of type `CoroutineEntry`, i.e. taking an opaque `void*` pointer and returning nothing. The function `qemu_coroutine_enter` transfers control to the coroutine.

The coroutine then executes until it returns, in which case it is automatically freed, or yields:

```
void coroutine_fn my_coroutine(void *opaque) {
  MyData *my_data = opaque;
  /* ... do some work ... */
  qemu_coroutine_yield();
  /* ... do some more work ... */
}
```

Yielding is done either directly by calling `qemu_coroutine_yield`, or indirectly by calling a function that yields (itself directly or indirectly). Yielding switches control back to the caller of `qemu_coroutine_enter`. This is typically used to switch back to the main thread's event loop after issuing an asynchronous I/O request. The request callback will then invoke the function `qemu_coroutine_enter` once more to switch back to the coroutine.

Finally, the QEMU coroutine interface provides a simple introspection mechanism based on two functions: `qemu_in_coroutine` can be used to check if the current function is executed in coroutine context, and `qemu_coroutine_self` to get a pointer to the current coroutine if this is the case.

3.2 Coroutine and blocking annotations

Functions that are run inside a coroutine and may yield are called *coroutine functions*, and annotated with `coroutine_fn`. Note that any function that calls a coroutine function is prone to yielding itself. Therefore, a coroutine function may only be called by another coroutine function; in other words, it is forbidden to call a coroutine function from a non-coroutine, *native* function. Coroutine functions, on the other hand, are allowed to call native functions. Coroutine annotations are used twice in the coroutine API (Figure 3): coroutine entry points (`CoroutineEntry`) must be coroutine functions, since they are executed in a coroutine, and `qemu_coroutine_yield` is of course annotated as a coroutine function.

Blocking functions, on the other hand, are native functions that must not be called from a coroutine; they are annotated with `blocking_fn`. We introduce this new annotation, which did not exist in QEMU before this work, to identify native functions that could block the main event loop for a long time and have a coroutine equivalent that should be used instead. Note that in principle, it would be even safer to consider every native function as potentially blocking, and annotate explicitly those that we wish to allow in coroutine context; however, such a white-list mechanism would be intractable in practice on a project of the size of QEMU, and we opted for a black list of blocking functions instead.

One important limitation of QEMU before our work was that these global constraints on the function call graph were not enforced in any way: as shown in Figure 1, `coroutine_fn` and `blocking_fn` are simply defined as empty macros by default, hence discarded from the final source-code by the C preprocessor. We discuss in Sections 6.2 and 4 how to give them a rigorous semantics, and how to check that annotated functions are used correctly.

3.3 Indirect coroutine calls

QEMU uses indirect coroutine calls and function pointers to coroutine functions intensively. We have seen in Figure 1 that every call to `qemu_coroutine_create` involves a pointer to a coroutine function, but this is far from the only place where they are used.

As explained above, coroutines are mainly used to provide non-blocking accesses to emulated disk images in the iothread. To support multiple disk image formats in an extensible way, the block layer of QEMU defines a generic block driver interface (Figure 2). This interface consists in a set of more than 40 callback functions that each driver needs to implement; among them, 17 are coroutine functions.

```
struct BlockDriver {
  const char *format_name;
  int (*bdrv_probe_device)(const char *filename);
  int coroutine_fn (*bdrv_co_flush_to_os)
                          (BlockDriverState *bs);
  /* ... */
};
```

Figure 2. Native and coroutine callbacks in block driver interface

The block layer then uses this abstract interface to implement I/O operations, performing many indirect calls to coroutine functions provided by each driver. For instance, the coroutine function `brdv_co_flush` calls the coroutine callback `brdv_co_flush_to_os`.

```
int coroutine_fn
bdrv_co_flush(BlockDriverState *bs)
{
  /* Write back cached data to the OS */
  if (bs->drv->bdrv_co_flush_to_os) {
   int ret = bs->drv->bdrv_co_flush_to_os(bs);
   if (ret < 0) {
     return ret;
   }
  }
  /* ... */
}
```

To preserve the coroutine constraint on the call graph, it is essential that function pointers be explicitly annotated. Coroutine annotations on function declarations and definitions alone are not enough to ensure the correctness of coroutine calls.

4. Static analysis of coroutine annotations

Coroutine annotations are not only useful to document which functions might block the event loop, and which ones can safely be used in a non-blocking way. In an event-driven program, as well as in a concurrent system with cooperative threads or coroutines, the main loop acts as a global lock, and it is common for programmers to rely on it to synchronise access to shared resources or preserve global invariants. Even in a stack-switching approach to coroutines, calling a coroutine function outside of coroutine context can then lead to serious bugs.

Ironically, such a bug occurred in QEMU during the course of our study. The code responsible for throttling disk I/O was causing a segmentation fault because a function to reschedule coroutines, `qemu_co_queue_next`, was missing a coroutine annotation and was called from native functions. It remained broken for two months before Canet identified the bug and fixed it [9]. However, as we discovered later when checking statically the impacted file, the fix itself still misses some coroutine annotations: getting all of them correct without some form of automated verification is a daunting task (see Section 4.3 for more details).

The CPC translator enforces this rule statically, because it needs correct annotations to drive its transformation. However, it only performs limited checking and is not convenient to analyse and infer coroutine annotations on a large scale (Section 4.1). The fact that it interleaves the analysis of coroutine annotations with source-to-source transformations increases the opportunities for bugs, and makes it harder to add new features. In order to fix a large number of annotations in an efficient and reliable way, we decided to write CoroCheck, a generic tool for the static analysis and inference of coroutine annotations, designed to be usable for CPC, QEMU, or any C other library with coroutine annotations (Section 4.2).

We illustrate the use of CoroCheck on a small example from QEMU in Section 4.3, and evaluate the number of annotations that CoroCheck enabled us to fix in QEMU in Section 4.4.

4.1 Missing coroutine annotations

In theory, it should have been enough to change a single line in the header file `coroutine.h` to make the CPC translator recognize and convert coroutine functions in QEMU:[2]

```
#define coroutine_fn __attribute__((cps))
```

In practice, however, this early attempt failed because many coroutine annotations in QEMU were missing, and CPC produced hundreds of errors caused by inconsistent annotations. As shown in Table 1 (Section 4.4, more than 70 % of the coroutine annotations required to compile QEMU were missing. In hindsight, this should have come as no surprise. As explained in Section 3, coroutine annotations are used exclusively for documentation purposes. QEMU is a very large project, and not every contributor understands how the coroutine mechanism works. Even fewer keep in mind the rules about coroutine annotations, and mistakes easily go unnoticed since there is no automated check to detect them.

A naive approach to fix coroutine annotations is to blindly follow the error messages reported by the CPC translator: adding missing annotations where errors are reported, which will produce more errors at the call points of these newly annotated functions, and iterating until one reaches a fixed-point. Apart from being extremely tedious, there are two reasons why this naive approach does not work well, or even at all in the case of QEMU: spurious annotations and hybrid functions.

Spurious annotations There is a risk of introducing too many annotations. Each illegal call to a coroutine function from a native function can be fixed either by annotating the caller, or by wrapping the callee in a native function allocating a dedicated coroutine for this call. This is a design choice that only the programmer can make, based on the concurrent structure of the program: it would be unreasonable to systematically add annotations all the way to the `main` function at the root of the call graph because of a single annotation at one of the leaves.

Conversely, it is sometimes a deliberate choice to spuriously annotate functions which do not call any other coroutine functions, e.g. for documenting the intention and further plans to add cooperation in some place of the code. Hence spurious annotations should not be removed automatically.

Hybrid functions Hybrid functions use `qemu_in_coroutine` to check dynamically whether they are called in coroutine context, and execute a different code path in each case. Such functions do not work with CPC, and need to be rewritten to split the coroutine and native code paths.

The need for blocking functions (Section 3.2) arose mainly when splitting hybrid functions into a native and a cooperative version. Annotating the native one as a blocking function would make sure that it is not called by mistake from a coroutine, which would block the whole event loop. Both the splitting of hybrid functions and the introduction of the `blocking_fn` annotation were discussed with and approved by QEMU developers — several of them considered hybrid functions as a temporary work-around, used to ease the transition when coroutines were first introduced in QEMU.

4.2 CoroCheck

CoroCheck is a generic tool for the static analysis and inference of coroutine annotations, written in OCaml. To make CoroCheck easily available and usable by as many QEMU developers as possible, we extended CIL with a modular plug-in system, and wrote CoroCheck as a CIL plug-in. This improves the previous cumbersome CIL architecture, which required users to recompile their own version to add new features, and should make it easier for every programmer to distribute analysis and program transformation tools based on CIL in the future.

CoroCheck analyses one C file at a time. It assumes that coroutine functions and function pointers are marked with an attribute,[3] as detailed in Section 6.2, and that the prototypes of functions implemented outside of the analysed file are correctly annotated. This is a necessary assumption, since CoroCheck has no means to determine whether extern functions are actually cooperative or not.

Assuming the correctness of external annotations implies that the programmer is responsible for analysing the files in a topological order based on their dependencies, or to iterate the analysis over the whole project until reaching a fixed-point. This proved not to be a problem in practice.

For each file that it analyses, CoroCheck performs a coroutine annotation inference, prints warnings for missing and spurious annotations, outputs an annotated call graph to help the programmer analyse and fix those errors, and checks that type casts and assignments respect coroutine annotations. We detail each of these steps in the rest of this section.

Coroutine annotation inference CoroCheck builds a directed graph of the functions calls in the analysed file, with a node for each function (either defined in the file, or simply declared), and edges from caller to callee. It then uses the `Fixpoint` module of the Ocamlgraph library [11] to propagate coroutine annotations. As explained above, we start with the coroutine functions implemented outside of the current file as a trusted root, and propagate the annotations backwards to their callers until we reach a fixed-point.

There is in fact another class of functions that we add to the roots of the algorithm: coroutine functions which have their address retained in a function pointer. This turned out to be necessary to avoid generating too many warnings about spurious annotations: when a function is used to implement an interface such as `BlockDriver` (Figure 2), it is essential to obey the annotation constraints mandated by the interface. For further safety, CoroCheck also verifies automatically that these function pointers are used consistently.

CoroCheck then iterates over every function defined in the file and prints warnings when the inferred coroutine annotation does not match the original one. It also warns if a blocking function is inferred as cooperative, or called from a coroutine function (either inferred or annotated in the original file).

Annotated call graph As explained in Section 4.1, blindly following CoroCheck suggestions for annotating coroutine functions is not necessarily enough: one might need to refactor an interface or split a hybrid function for example. Having a per-file graphical view of the function call graph proves very helpful in understanding the reasons for the suggested fixes and analysing the root causes of erroneous annotations.

CoroCheck uses Ocamlgraph facilities to output a file that can be processed with the `dot` utility (from the GraphViz project) to generate an image representing the annotated function call graph. Consider for example the input program in Figure 3; it yields the output shown in Figure 4.

Functions *inferred* as coroutine functions are represented in rectangles; native functions in ovals. Annotations *provided by the programmer*, on the other hand, are printed within the boxes (`coroutine_fn` or `blocking_fn` in Figure 4). For indirect calls, since the name of the called function is unknown, we print

[2] See Section 6.2 for an explanation of `__attribute__((cps))`.

[3] The name of the attribute is `coroutine_fn` by default, and configurable with a command-line option.

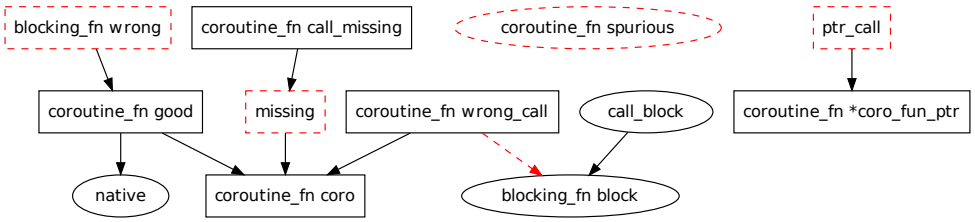


Figure 4. Function call graph annotated by CoroCheck

```
extern void coroutine_fn coro();
extern void blocking_fn block();

void native() { };
void coroutine_fn (*coro_fun_ptr)(void) = &coro;

void coroutine_fn spurious()     { }
void coroutine_fn good()         { coro(); native(); }
void             missing()       { coro(); }
void coroutine_fn call_missing() { missing(); }
void blocking_fn  wrong()        { good(); }
void             call_block()    { block(); }
void coroutine_fn wrong_call()   { coro(); block(); }
void             ptr_call()      { coro_fun_ptr(); }
```

Figure 3. Input program for Figure 4

the expression used to perform the call instead (eg. `coroutine *coro_fun_ptr`).

Mismatching nodes, corresponding to either spurious or missing annotations, are ouput with dashed, red lines. For example, from top-left to bottom-right in Figure 4, the function `wrong` should be annotated with `coroutine_fn` instead of `blocking_fn` because it calls the coroutine function `good`; the function `spurious` does not call any coroutine function, so it does not need a `coroutine_fn` annotation; the function `ptr_call` should be a coroutine functions because it performs an indirect call to a coroutine function through the function pointer `coro_fun_ptr`; and the function `missing` should be annotated as well since it calls the coroutine function `coro`. Note that the function `call_missing` is not flagged as spurious, since it calls `missing` which should be a coroutine function (even though it lacks the proper annotation). Finally, dashed, red lines are also used for forbidden edges from coroutine functions to blocking functions, such as `wrong_call` calling `block`.

Files from a real-world program can contain a huge number of functions, making the call graph cluttered and challenging to decipher. To focus on the relevant information, CoroCheck removes every native function defined outside of the current file from the graph. This strategy removes many leaves, in particular all functions from the `libc` standard C library. For example, when applied to `block.c`, it removes more than half of the nodes and edges (from 657 nodes and 781 edges down to 316 nodes and 302 edges).

Type cast and assignment verification As an additional safety check, unrelated to call graph analysis, CoroCheck ensures that coroutine annotations are not lost when function pointers are type-cast or assigned to a variable. This is relatively straightforward thanks to CIL making every type cast explicit in its intermediary AST.

4.3 An example of coroutine-safety violation

In this section, we show that annotations are significant even in the original QEMU, and that a missing annotation can cause (and has in fact caused) a serious and hard-to-find bug.

Coroutine locks (`CoMutex`) are built on top a coroutine queues (`CoQueue`). The basic operations are to enqueue the current coroutine, which transfers control to its caller (`qemu_co_queue_wait`), and to restart the next coroutine in a queue (`qemu_co_queue_next`). Locking operations are thin wrappers around these functions which check if the `CoMutex` is already locked before proceeding (`qemu_co_mutex_lock` and `qemu_co_mutex_unlock`). Figure 5a gives an overview of the file implementing coroutine queues and locks.

CoroCheck detects no less than six functions missing a coroutine annotation, and one (`qemu_co_queue_run_restart`) spuriously annotated. These missing annotations caused a serious bug: the code responsible for throttling disk I/O trusted them, and called `qemu_co_next` in native context. This ultimately led to calling `qemu_coroutine_self` in native context, which returned an invalid coroutine pointer and caused a segmentation fault in `qemu_co_queue_do_restart`.

Canet [9] fixed the segmentation fault two months later by introducing `qemu_co_enter_next`, a native-mode counterpart to `qemu_co_queue_next`. However, as shown on Figure 5b, quite a few annotations are still missing or incorrectly placed after the patch. Without an automated verification tool such as CoroCheck, it is probably only a matter of time before the wrong function is used again by mistake in another part of QEMU.

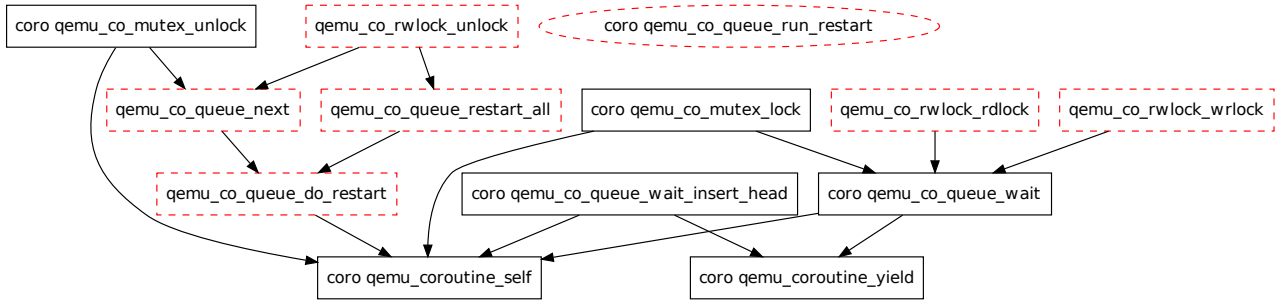
4.4 Evaluation

We have no doubt that it would have been much less likely to obtain fully correct annotations in the short time-frame of our study without using CoroCheck. Although this is but a subjective impression, the figures shown in Table 1 give a better idea of the amount of work required. Compared to the correctly annotated `cpc` branch, the original `master` branch lacks more than 70 % of annotations. We firmly believe that bug-free annotations cannot be reached in a project of this scale without automated, static verification tools.

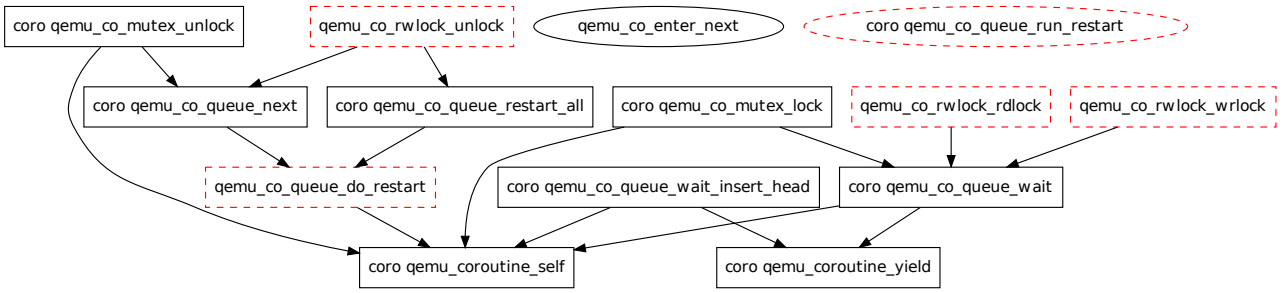
Table 1. Coroutine annotations in QEMU

QEMU branch		`master`	`cpc`
`coroutine_fn`	in `.h` files	31	100
	in `.c` files	127	421
Hybrid functions	(`qemu_in_coroutine`)	6	1

A few files non-essential files have not been annotated yet, hence the remaining hybrid function.



(a) Before Canet's patch



(b) After Canet's patch

Note: the graph has been slightly simplified, and we use `coro` instead of `coroutine_fn` to keep it readable.

Figure 5. Function call graph from `qemu-coroutine-lock.c`, annotated by CoroCheck

5. Continuation-Passing C

Continuation-Passing C (CPC) [24] is an extension of the C language to write concurrent programs. The programmer writes synchronous code in threaded style, using common synchronisation techniques such as condition variables. This code is then automatically transformed into an equivalent standard C program written in asynchronous, event-driven style. Our previous experiments on small to medium-sized programs have shown that the translated code is at least as efficient as stack-based thread implementations, while providing a significantly smaller memory footprint [23, 25]. However, CPC had never been tested on programs as large as QEMU before this case study.

The CPC programmer uses a superset of the C language that we call the *CPC language*. It extends C with two keywords: `cps` to annotate cooperative functions, and `cpc_spawn` to execute them in a new thread; it implements half-a-dozen synchronisation primitives (`cpc_yield`, `cpc_sleep`, etc.); and it provides a moderate amount of syntactic sugar to write common concurrency idioms in a concise way. As we have seen in Section 1, a CPC program is compiled in two steps:

1. The *CPC translator* first applies a series of source-to-source transformations to convert the original program into an equivalent one written in continuation-passing style. The resulting code is compiled by a C compiler, such as GCC.
2. This compiled code is then linked with the *CPC runtime*, or, in the context of this paper, with a new coroutine backend for QEMU based on continuations (Section 6.3).

In the rest of this section, we give an overview of the compilation technique used by the CPC translator on a small example. More details, including proofs of correctness, are available in a previous article [24].

5.1 The CPC compilation technique

The CPC translator is structured in a series of proven source-to-source transformations, which turn a threaded-style CPC program into an equivalent event-driven C program. *Boxing* first encapsulates a small number of variables in environments. *Splitting* then splits the flow of control of each annotated function into a set of nested functions. *Lambda lifting* removes free local variables introduced by the splitting step; it copies them from one nested function to the next, yielding closed nested functions. Finally, the program is in a form simple enough to perform a one-pass partial *CPS conversion*. The resulting continuations are used at runtime to schedule threads.

Consider the following function, which counts seconds down from an initial value `x` to zero.

```
cps void countdown(int x) {
  while(x > 0) {
    printf("%d\n", x--);
    cpc_sleep(1);
  }
  printf("time is over!\n");
}
```

This function is annotated with the `cps` keyword to indicate that it yields to the CPC scheduler. This is necessary because it calls the CPC primitive `cpc_sleep`, which also yields to the scheduler. In the rest of this article, we call such functions *cps functions*.

We show below how splitting, lambda lifting and CPS conversion transform the function `countdown`. The boxing pass has no effect on this example because it only applies to address-taken variables (the address of which is retained by the "address of" operator `&`). It is necessary in the general case to transform address-taken stack variables into heap variables, because lambda-lifting and CPS-conversion passes duplicate stack variables, which invalidates pointers storing their address.

5.2 Splitting

The next transformation performed by the CPC translator is *splitting*. Splitting has been first described by van Wijngaarden for Algol 60 [42]. It translates control structures into mutually recursive functions.

Splitting is done in two steps. The first step consists in replacing every control-flow structure, such as `for` and `while` loops, by its equivalent in terms of `if` and `goto`.

```
cps void countdown(int x) {
 loop:
  if(x <= 0) goto timeout;
  printf("%d\n", x--);
  cpc_sleep(1);
  goto loop;
 timeout:
  printf("time is over!\n");
}
```

The second step uses the fact that `goto` are equivalent to tail calls [39]. It translates every labelled block into an nested function, and every jump to that label into a tail call (followed by a `return`) to that function.

```
cps void countdown(int x) {
  cps void loop() {
    if(x <= 0) { timeout(); return; }        /* (1) */
    printf("%d\n", x--);
    cpc_sleep(1); loop(); return;            /* (2) */
  }
  cps void timeout() {
    printf("time is over!\n"); return;       /* (3) */
  }
  loop(); return;
}
```

Splitting yields a program where each cps function is split in several mutually recursive, atomic functions, very similar to event handlers. Additionally, the tail positions of these nested functions are always one of the following three cases (numbered in the previous example):

1. a tail call to another cps function (eg. `timeout`),
2. a call to an external cps function (`cpc_sleep`) followed by a tail call to an nested cps function (`loop`),
3. or a single `return` statement with no preceding call to a cps function.

This restricted form, that we call *CPS convertible* form, provides strong enough guarantees to enable a correct and straightforward CPS conversion at a later stage.

Another effect of splitting is that variables bound in the original outer function appear free in the nested ones. For instance, the variable `x` is free in the function `loop` above. Because C does not support nested functions, we need a pass of lambda lifting to eliminate those free variables.

5.3 Lambda lifting

The CPC translator makes the data flow explicit with a lambda-lifting pass. Lambda lifting, also called closure conversion, is a standard technique introduced by Johnsson [22] to remove free variables. It is also performed in two steps: parameter lifting and block floating.

Parameter lifting binds every free variable to the nested function where it appears (for instance, `x` is bound to `loop` on line (1) below). The variable is also added as a parameter at every call point of the function (lines (2) and (3)).

```
cps void countdown(int x) {
  cps void loop(int x) {                     /* (1) */
    if(x <= 0) { timeout(); return; }
    printf("%d\n", x--);
    cpc_sleep(1); loop(x); return;           /* (2) */
  }
  cps void timeout() {
    printf("time is over!\n"); return;
  }
  loop(x); return;                           /* (3) */
}
```

Block floating is then a trivial extraction of closed, nested functions at top-level.

Note that because C is a call-by-value language, lifted parameters are duplicated rather than shared. Therefore, this transformation is not correct in general: mutating a copied parameter would leave the original one intact, which could in principle be observed and yield different results. It is however sound in the case of CPC because lifted functions are always called in tail position (Section 5.2): they never return, which guarantees that at most one copy of each parameter is reachable at any given time [24, Section 6].

5.4 CPS conversion

Finally, the control flow is made explicit with a CPS conversion [34, 40]. The continuations store callbacks and their parameters in a regular stack-like structure `cont` with two primitive operations: `push` to add a function on the continuation, and `invoke` to call the first function of the continuation.

```
cps void loop(int x, cont *k) {
  if(x <= 0) { timeout(k); return; }
  printf("%d\n", x--);
  cpc_sleep(1, push(loop, x, k)); return;
}
cps void timeout(cont *k) {
  printf("time is over!\n");
  invoke(k); return;
}
cps void countdown(int x, cont *k) {
  loop(x, k); return;
}
```

Just like lambda lifting, CPS conversion is not correct in general in an imperative call-by-value language, because variables are duplicated to be stored on the continuation. It is however correct in the case of CPC, for reasons similar to the correctness of lambda lifting [23, Chapter 5].

6. CPS-converting QEMU

Our continuation-based backend for QEMU is made of two parts, similar to the two-step approach used to compile CPC programs (Section 5). We first need to perform a CPS conversion of every annotated coroutine function, and then use the continuations introduced by the CPC translator to implement QEMU's coroutine API (Figure 1).

In this section, we study the challenges associated with these two steps: parsing correctly such a large-scale base of C code (Section 6.1), and defining CPS conversion as a C calling convention to support indirect calls to CPS-converted functions (Section 6.2); then implementing the runtime `coroutine-cpc` backend (Section 6.3).

6.1 Compiling QEMU with CIL

CPC is built on top of the C Intermediate Language (CIL) framework [32]. CIL is a front-end for the C programming language that facilitates program analysis and transformation. It parses and typechecks an input program, and translate it into an equivalent, simplified subset of C. In CIL, for example, expressions have no side-effect and all looping constructs are normalised to a `while(1)` loop with `break` statements. The programmer then manipulates this simplified AST, which greatly reduces the number of cases that must be considered. CIL finally outputs the resulting AST in a C file which is handed over to the actual C compiler. A Perl script acts as a drop-in replacement for GCC, automating these stages and making it easy to compile existing projects through CIL.

Before attempting to translate coroutine functions with CPC, we needed to make sure that the CIL front-end was able to correctly parse and translate QEMU source code. Our very first step was therefore to try and compile QEMU using CIL alone. Although CIL is reasonably complete and well-tested, QEMU is a large project with over 750 000 lines of C code, and it uses a wide range of C features, including a number of non-standard GCC extensions.

While attempting to compile QEMU with CIL, we discovered and fixed at least two major bugs, and several lacking features. All those fixes and features are now included in CIL and will be available in the next release.

Major bugs in CIL The most disconcerting CIL bug that we encountered did not prevent QEMU from compiling. Instead, it caused a crash of the guest operating system when we tried to start a Linux 3.2 kernel with hardware virtualization disabled. Older kernels seemed to work completely fine, as well as newer kernels when virtualization was enabled. It turned out to be an erroneous implementation in CIL of C rules for arithmetic conversion. This fundamental bug had gone unnoticed for so long because it did not produce observable effects except in some corner case on `long long` integers, triggered in particular by QEMU's emulation of Intel's SSE extensions.

We also found a bug in initialization of arrays: CIL discarded trailing empty initializers. For instance, `{{.x = 3}, {.x = 5}, {} }` was interpreted as an array of size 2 instead of 3; the last initializer is equivalent to `{.x = 0}` in that case. This idiom is used extensively in QEMU to mark the end of arbitrarily sized arrays of complex structures without setting explicitly each field of the last element to `0`.

A number of more minor bugs were also fixed along the way. The Perl wrapper script in particular needed to be updated because it failed to pass through some GCC options.

New C features and GCC extensions We added support for flexible array members, a C99 feature allowing an array of unspecified size to be the last member of a C structure. This is used in several places in QEMU, to allocate an array of data and some meta-data in a compact, cache-efficient way.

We also improved support of GCC extensions. We added many new GCC builtins, including concurrency operations for the C11 memory model. We implemented first-class support for case-ranges, allowing to write `switch` statements with conditionals of the form `case 0x0000 ... 0xc000`; CIL already accepted that extension, but translated it into an exhaustive list of all cases, which did not scale to very large ranges.

As an aside, some GCC extensions are not available on every platform, and QEMU provides fall-back mechanisms for those cases. This has allowed us to disable two extensions that would have required an unreasonable amount of work to be added to CIL: 128-bit integers (`int128_t`) and SSE vector instructions.

6.2 Indirect coroutine calls

The next stumbling block on the path to CPS conversion was indirect coroutine calls. In the original CPC implementation, the `cps` annotation is a new keyword defined as a *function specifier*. This means that it applies to the function being annotated, rather than to its type. This unfortunate decision, taken at an early stage in CPC design, forbade `cps` annotations at the type level, and made it impossible to apply a CPS conversion to QEMU source code, because of the pervasive use of coroutine function pointers (Section 3.3). In this section, we argue that coroutine annotations denote a *calling convention* [21], and we detail how to implement this approach in CPC.

Native functions need to agree on a calling convention, often defined in the Application Binary Interface (ABI) of the architecture they are compiled for, for instance to decide how to pass function parameters (on the stack or via register), or whether stack frames should be cleaned by the caller or the callee. Even given a particular language and architecture, several calling conventions might coexist. For instance, on Intel 386, the *cdecl* convention passes every argument on the stack, whereas the *fastcall* convention passes the first two arguments in registers `ECX` and `EDX`.

There is no standard way in C to specify the calling convention associated to a function or a function pointer. Fortunately enough, to ease interoperability, most compilers provide a means of specifying the calling convention explicitly on a per function. The *de facto* standard is to use attributes, a generic mechanism which allows to annotate types, function declarations and even expressions.

Figure 6 shows how function attributes for calling conventions are used in practice. The attribute can be applied to a function prototype, like `f` (1), or to the type of a function pointer like `p` (2). Tracking the calling convention within the type of each pointer then allows to perform indirect calls with the correct convention (4). The assignment of a value to the pointer (3) and the implementation of a prototype (5) are checked by the compiler, which will issue a warning or an error if an incompatible calling convention is used; in our example, line (3) is correct but (5) is forbidden.

```
int __attribute__((fastcall)) f(int a, int b);    /* 1 */
int __attribute__((fastcall)) (*p)(int a, int b); /* 2 */
p = &f;                                           /* 3 */
int z = (*p)(1,2);                                /* 4 */
int f(int a, int b) { return a + b; }             /* 5 */
```

Figure 6. Calling convention attributes

Those are exactly the properties that we need to keep track of coroutine annotations within types, and to be able to perform indirect CPS-converted calls. As a matter of fact, CPC actually defines a calling convention for cps functions, specifying how to store parameters on the continuation and how to pass return values to the next function. This calling convention is precisely the set of assumptions used by the CPC runtime to interface primitive functions with the code produced by the CPC translator.

Implementing coroutine annotations with function attributes rather than the `cps` keyword simplified the lexing and parsing stages, and made the internal data structures and transformations used by the CPC translator less ad-hoc. Backward-compatibility merely required to add a single line to the runtime header:

```
#define cps __attribute__((cps))
```

We encountered two technical difficulties. First, we needed to be very careful to be compatible with the way `gcc` and other compilers implement calling-convention function attributes. The rules for positioning are very liberal, and not formally defined in the documentation of compilers: in general, attributes apply to the nearest component of the type, but in the case of calling convention

attributes, they can be placed almost anywhere within the return type and still apply to the function type as a whole. Then, calling-convention attributes also need to be treated differently than other attributes when merging the attributes of a prototype with those of the implementation: because they create a constraint on the caller, it is essential to ensure that the calling convention is the same in all cases to ensure a consistent usage between the header and the implementation (Figure 6, line 5).

6.3 The coroutine-cpc backend

The last step of the QEMU/CPC project is `coroutine-cpc`, a new implementation of coroutines for QEMU based on the continuations introduced by the CPC translator.

One important requirement was to write a simple, self-contained backend: having as few changes to QEMU as possible, except for the missing coroutine annotations that are dealt with separately, means that it is easier to track the evolution of QEMU, and to potentially merge our backend at some point in the future. The implementation of `coroutine-cpc` is short, with less than 200 lines of portable, carefully-optimised code. It also limits the amount of new code to the minimum, by re-using code from the original CPC runtime implementation for each low-level management task: allocating, deallocating, resizing continuations, and passing a return value to the next continuation [23, Section 3.2.4].

The QEMU coroutine API and the CPC runtime are very close, offering the same kind of primitives. One noticeable difference is that QEMU exposes coroutines, providing an explicit handle for the programmer to schedule them, whereas CPC exposes threads with an implicit scheduler and no thread handle. The interface of the former is therefore lower-level but slightly more expressive than that of the latter. As a result, fewer native cps functions need to be implemented for the QEMU API than for the CPC runtime: functions such as `cpc_sleep` or `cpc_io_wait` need to hook into the CPC scheduler, whereas they are built independently from the backend in the case of QEMU, on top the basic coroutine API. For the `coroutine-cpc` backend, we only need to implement four QEMU-specific functions: `qemu_coroutine_enter`, `qemu_coroutine_yield`, `qemu_coroutine_self` and `qemu_in_coroutine`.

Entering and yielding coroutines The implementation of the function `qemu_coroutine_enter` initialises the continuation if this is the first time it is entered, and starts a trampoline loop to run it (see Figure 7). The function pointer on top of the continuation is extracted (2), and called with the rest of the continuation as a unique parameter (4); it returns a new continuation, and the process is repeated until reaching the empty continuation (1) or a call to `qemu_coroutine_yield` (3).

```
while(1) {
  /* (1) If continuation is empty, return */
  if (k->length == 0) return k;
  /* (2) Otherwise, extract function pointer */
  k->length -= sizeof(cpc_function *);
  f = *(cpc_function **)(k->c + k->length);
  /* (3) Intercept yield if necessary */
  if (f == qemu_coroutine_yield) return k;
  /* (4) Otherwise, run the extracted function */
  k = (*f)(k);
}
```

Figure 7. Trampoline loop for continuations

The usual way to implement yield in continuation-passing style is to write the CPS-form directly by hand, returning a null pointer to indicate that there is no continuation left to execute because the coroutine has yielded (and adapting the trampoline loop accordingly). This approach is not convenient in the case of QEMU because `qemu_coroutine_yield` is defined in a supposedly backend-independent way, which in fact relies implicitly on a stack-switching implementation. To work-around this limitation without modifying files outside of `coroutine-cpc`, we intercept the call to `qemu_coroutine_yield` in the trampoline loop, by comparing function pointers (Figure 7).

Dynamic bookkeeping To implement the introspection functions `qemu_coroutine_self` and `qemu_in_coroutine`, all existing QEMU backends use a thread-local variable to keep track of the current coroutine. This variable is updated on each coroutine switch by `qemu_coroutine_enter`. As it turns out, this dynamic bookkeeping is no longer necessary for `coroutine-cpc`.

It is enough to annotate these functions with the special attribute `cpc_need_cont`. Then, the CPC translator passes them the current continuation directly when they are called from coroutine context, and a null pointer otherwise. In fact, once hybrid functions have been eliminated, `qemu_in_coroutine` becomes completely redundant with coroutine annotations, its use being subsumed by the static verification of CoroCheck.

In addition to making the code simpler and safer, we measured a speedup of up to 10 % in micro-benchmarks when we removed dynamic bookkeeping in `coroutine-cpc`.

7. Experimental results

In this section, we compare the performance of coroutine backends on micro-benchmarks (Section 7.1) and evaluate the performance impact of CPS conversion on a typical QEMU work-load (Section 7.2).

7.1 Micro-benchmarks

To evaluate the efficiency of basic coroutine operations, we use three micro-benchmarks from the test suite of QEMU.[4] *Lifecycle* repeatedly creates an empty coroutine, which is entered then destroyed immediately. *Nesting* repeatedly creates 1 000 nested coroutines, each of them incrementing a shared counter, then creating and entering the next coroutine. *Yield* repeatedly enters a single coroutine which decrements a counter, then yields immediately.

We test the continuation-based backend `cpc`, the stack-switching backends `ucontext` and `sigaltstack`, and the thread-based backend `gthread`. All benchmarks are compiled directly with gcc, except the `cpc` backend which uses the CPC translator; we have verified that the results are unchanged when compiling the other backends with the CIL front-end. The results are shown in Table 2.

Because allocating coroutines is a costly process for most backends, QEMU uses a pool of 64 coroutines: instead of freeing coroutines that have completed, they are kept in a linked-list, and re-used when a new coroutine is needed. To measure the impact of allocation, we perform each benchmark with and without the coroutine pool (except for `gthread`, which does not support the pool at all).

The `cpc` backend is consistently faster than every other backend. Allocation is faster because continuations are resized dynamically when needed, whereas other backends need to allocate a large chunk of memory at once. As a result, the coroutine pool is extremely effective for other backends, but turns out to slow down `cpc` in the *nesting* benchmark: when the number of coroutines is one order of magnitude larger than the size of the pool, the cost of managing the pool becomes higher than its benefits. As expected, the pool has

[4] Our benchmarks scripts are available online: `http://github.com/kerneis/cpc-qemu-bench/`.

Table 2. Speed of basic coroutine operations for various backends

	Lifecycle		Nesting		Yield	
Pool	no	yes	no	yes	no	yes
cpc	75	54	94	125	19	19
ucontext	464	108	3 899	682	83	84
sigaltstack	1 796	108	5 843	1 988	85	87
gthread	10 802	—	2 826 905	—	5 703	—

All speeds are in nanoseconds, averaged over 10 runs (except *gthread-nesting*, over 5 runs) of millions of iterations, on an 8-core Intel Xeon E5-1620 at 3.6 GHz. For *nesting*, the time is per nested coroutine, hence comparable to *lifecycle* directly.

almost no impact for *yield*, since only one coroutine is used in this test.

We are not sure exactly why the gap is so large between the *lifecycle* and *nesting* benchmarks for most backends: both tests perform exactly the same task, except that the latter uses more coroutines simultaneously and maintains a shared counter between coroutines. The memory pressure is certainly higher, but it is not clear why it slows down the creation of each coroutine so much, especially when the pool is disabled anyway. We believe that cache effects are involved here, which would explain why `cpc` performs much better with its small memory footprint.

The result of the *yield* benchmark is not surprising. Coroutine switching is a mere function return in the case of `cpc`, hence faster than the signals, stack-switching mechanisms or locks used by the other backends.

7.2 Macro-benchmarks

The main downside of CPS conversion is that it adds an overhead to each call to a CPS-converted function. It is hard to predict the global overhead on a large program because the splitting pass introduces a number of calls to CPS-converted functions which varies with the complexity of the control-flow and the position of cooperation points. While the CPC translator tries to limit the number of inserted calls by performing incremental transformations, we need to perform macro-benchmarks to evaluate the overall effectiveness of our approach.

Since coroutines are mainly used in the block layer implementation, we need to generate a lot of disk I/O from the virtual machine. We use a virtualized guest OS with Debian "squeeze", ran on a Debian "squeeze" host on an 8-core Intel Xeon 3.6 GHz, and `fio` to generate various intensive disk accesses patterns, with up to 500 simultaneous readers or writers. The guest is installed in a disk image using QEMU's `qcow2` disk format. We take particular care to disable disk caches in both the guest OS and QEMU's disk layer, to make sure each access in the guest translates to an actual read or write of the image file on the host; disk cache is kept enabled on the host. We compare the mean and median access time for each access pattern and coroutine backend.

Unfortunately, none of the coroutine backend performs significantly faster or slower than the others in these macro-benchmarks. To get a finer-grained understanding of the results, we profile each QEMU instance with the Linux `perf` utility, which uses hardware performance counters provided by the CPU. As it turns out, coroutines are not at all on the critical path when emulating or virtualizing a whole system: slightly less than 1 % of the execution time is spent in coroutine-related functions, and the differences between backends end up being smaller than the variability of disk-access times.

However, `perf` allows for a fine-grained analysis of the time spent in each function, with some measurement uncertainty due to its event-sampling approach. Therefore, it is possible to isolate the most time-consuming coroutine function in `perf` results (for instance `qcow2_co_writev` for a write benchmark), and compare its ranking for the various coroutine backends.

This is a tedious process that we did not manage to automate: since the splitting pass of the CPC translator creates new coroutine functions, one needs to sum those to recover the global time spent in the original function. In the few runs that we analysed, the coroutine functions split by the CPC translator did not take significantly more time to execute. A likely explanation is that coroutine functions frequently yield in practice; the time wasted in the trampoline loop is negligible compared to the overhead of entering and switching coroutines for other backends.

Despite our efforts, these macro-benchmarks are frustratingly inconclusive. Profiling at least seems to indicate that CPS conversion does not generate a significant overhead in the case of QEMU. This is in line with the conclusions of our previous experiments on smaller programs [25].

8. Conclusions

We have applied a conversion to continuation-passing style to QEMU, a large open-source project written in C with heavy use of function pointers. Then, we have used these continuations to implement an alternative coroutine backend, both more portable and much faster than the existing ones. We have also developed CoroCheck, a tool for the static analysis of coroutine annotations, and used it to correct several hundreds of missing annotations in QEMU.

Our work demonstrates that static analysis of coroutines helps ruling out some actual bugs. It also shows that CPC is flexible and mature enough to scale efficiently to very large programs, and to a model (coroutines) that it was not initially designed to handle.

Beyond the scientific results, this is the story of a successful collaboration between an open-source project and academic research. Over the course of a few months, we have started a fruitful relationship, improving the state of coroutines in QEMU, developing a tool that they can use as part of their test suite, and fixing many bugs in CIL and CPC. We were certainly not the first, but we hope that many others will come after us, and discover the mutual benefits of scaling-up their techniques to real-world projects.

Acknowledgments

The authors wish to thank Alan Mycroft and Gabriel Scherer for their thoughtful suggestions on the structure of this article, as well as Matthieu Boutier, Marc Lasson, Raphaël Proust, and Peter Sewell for their comments. They are also beholden to the anonymous reviewers for their extremely detailed and valuable suggestions. This work has been supported by Google (Google Summer of Code 2013) and EPSRC (grants EP/H005633 and EP/K008528).

References

[1] C Intermediate Language. URL `http://cil.sf.net`.

[2] CoroCheck. URL `http://github.com/kerneis/corocheck`.

[3] Continuation-Passing C. URL `http://github.com/kerneis/cpc`.

[4] QEMU. URL `http://git.qemu-project.org/?p=qemu.git`.

[5] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In C. S. Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 289–302. USENIX, 2002.

[6] K. Anton and P. Thiemann. Towards deriving type systems and implementations for coroutines. In K. Ueda, editor, *APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2010.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[8] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5): 366–371, May 1966.

[9] B. Canet. block: repair the throttling code, July 2013. URL `http://git.qemu-project.org/?p=qemu.git;h=b681a1c73e15e08c70c10cccd9c9f5b65cca12e8`.

[10] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[11] S. Conchon, J.-C. Filliâtre, and J. Signoles. Designing a generic graph library using ml functors. In M. T. Morazán, editor, *Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 124–140. Intellect, 2007.

[12] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In R. Cytron and R. Gupta, editors, *PLDI*, pages 232–244. ACM, 2003.

[13] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - a software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[14] A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2), 2009.

[15] T. Duff. Duff's device, Nov. 1983. URL `http://www.lysator.liu.se/c/duffs-device.html`. Electronic mail to R. Gomes, D. M. Ritchie and R. Pike.

[16] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In A. T. Campbell, P. Bonnet, and J. S. Heidemann, editors, *SenSys*, pages 29–42. ACM, 2006.

[17] J. Fischer, R. Majumdar, and T. D. Millstein. Tasks: language support for event-driven programming. In G. Ramalingam and E. Visser, editors, *PEPM*, pages 134–143. ACM, 2007.

[18] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[19] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: composable asynchronous IO for native languages. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 903–920. ACM, 2011.

[20] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3/4):143–153, 1986.

[21] S. C. Johnson and D. M. Ritchie. The C language calling sequence. Computing Science Technical Report 102, Bell Laboratories, Sept. 1981.

[22] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[23] G. Kerneis. *Continuation-Passing C : program transformations for compiling concurrency in an imperative language*. PhD thesis, Université Paris Diderot, Paris, France, Nov. 2012.

[24] G. Kerneis and J. Chroboczek. Continuation-Passing C, compiling threads to events through continuations. *Higher-Order and Symbolic Computation*, 24:239–279, 2011.

[25] G. Kerneis and J. Chroboczek. CPC: programming with a massive number of lightweight threads. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*, pages 30–34, 2011.

[26] A. Key. Weave: translated threaded source (with annotations) to fibers with context passing, Nov. 2010. As used within RAID-1 code in IBM SSA RAID adapters (ca. 1995–2000). Personal communication.

[27] O. Kiselyov. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, 2012.

[28] M. N. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX Annual Technical Conference*, pages 87–100. USENIX, 2007.

[29] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 189–199. ACM, 2007.

[30] D. Mazières. A toolkit for user-level file systems. In Y. Park, editor, *USENIX Annual Technical Conference, General Track*, pages 261–274. USENIX, 2001.

[31] D. S. Myers, J. N. Carlisle, J. A. Cowling, and B. Liskov. MapJAX: Data structure abstractions for asynchronous web applications. In *USENIX Annual Technical Conference*, pages 101–114. USENIX, 2007.

[32] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[33] J. Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX 1996 Technical Conference*, June 1996. Invited talk.

[34] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[35] J. H. Reppy. Concurrent ML: Design, application and semantics. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993.

[36] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 317–328. ACM, 2009.

[37] K. F. Sagonas and D. Luna. Gradual typing of Erlang programs: a Wrangler experience. In S. T. Teoh and Z. Horváth, editors, *Erlang Workshop*, pages 73–82. ACM, 2008.

[38] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer, 2008.

[39] G. L. Steele Jr. and G. J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA, Mar. 1976.

[40] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.

[41] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In R. Rocha and J. Launchbury, editors, *PADL*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011.

[42] A. van Wijngaarden. Recursive definition of syntax and semantics. In *Formal Language Description Languages for Computer Programming*, pages 13–24, Amsterdam, Netherlands, 1966. North-Holland Publishing Company.

[43] J. Vouillon. Lwt: a cooperative thread library. In E. Sumii, editor, *ML*, pages 3–12. ACM, 2008.

[44] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243. ACM, 2001.

[45] K. Wolf and S. Hajnoczi. coroutine: introduce coroutines, Jan. 2011. URL `http://git.qemu-project.org/?p=qemu.git;h=00dccaf1f848290d979a4b1e6248281ce1b32aaa`.