On unifying the coroutines and resumable functions proposals

Revision 0

Document number:	P0073R0
Date:	2015-09-25

Author:	Torvald Riegel
Reply-to:	Torvald Riegel <triegel@redhat.com>

1 The goals of this paper

I have argued in the past that there are commonalities between the different coroutine proposals and that there are opportunities for a unified proposal. This paper is meant to provide more detail about this.

So far, the proposals have been presented as separate vertical solutions, and it has been claimed that there is no substantial common ground between them. However, I believe we should be looking for commonalities, or we will both make future extensions of the features presented in these proposals harder and will make all of them harder to learn for programmers.

To me, the design of coroutines and resumable functions is not just in the space of programming interfaces. A large part of it is also at the level of the abstract machine: In particular, how we deal with concurrency beyond `std::thread`? While it's easy for programmers to use `std::thread`, they often don't need a full-featured operating system thread nor would want one because of additional, unnecessary overhead that might be attached to it; a thread of execution from a thread pool, or something like a coroutine would often be sufficient and better. I believe there is consensus in SG1 that we need more kinds of threads of execution beyond those spawned by `std::thread` (see, for example, the parallelism TS or P0072R0).

The main points I'm arguing for in this paper are:

- Executing coroutines, generators, etc. should be understood as representing threads of execution, with certain execution properties. Note that I did *not* say that they should be OS threads or similar to what `std::thread` spawns.

- Resumable functions enable a certain implementation of threads of execution, in particular how the call stack is implemented. The so-called stackful coroutines use a more traditional stack implementation.

- The compiler can bridge the gap between those two stack implementations under certain conditions by providing the impression of a stackful implementation (e.g., calling through normal non-resumable functions) but generating code that is using resumable functions internally. The Transactional Memory TS is a precedent for very similar compiler support, showing that it can be both specified and implemented[1].

In this paper I will not discuss interfaces of programming abstractions. Personally, I am more concerned about the design at the conceptual level than about details of a particular interface; my focus is on the internals, both design-wise and how it affects implementations or enables certain ones.

2 Coroutines represent threads of execution

Let us first revisit what a thread of execution is according to the standard, and abbreviate it with *TOE* instead of "thread" to avoid the confusion over what a "thread" is (e.g., an operation system thread?). A TOE is defined in the standard as "a single flow of control within a program" (see §1.10p1). *sequenced-before* is defined exactly for a TOE (see §1.9p13). The standard seems not quite clear regarding whether a TOE is a static (e.g., a function) or dynamic entity (i.e., an instance of an execution of a single flow of control); in this paper, I will assume the latter. See N4321 for a more detailed explanation of this terminology (and of the existing terminology in the standard). N4321 and P0072R0 use the term "execution agent" as name for TOEs that have specific execution properties attached to them. The standard uses execution agents to describe lock ownership of TOEs. P0072 argues for lighter-weight modes of execution provided through different kinds of execution agents. "Execution context" was suggested as a different name instead of execution agent, which might make it clearer that the intent is to model execution properties and not primarily to describe the hardware or software resources used for execution. Nonetheless, for simplicity, I will just use TOE in this paper, and associate execution properties with TOEs.

I think that we should embrace parallelism and concurrency—not just by providing programming abstractions in this area but also by being honest when and where parallelism or concurrency exist in a computation. We should not shy away from having multiple TOEs exist at the same time in a running program. We do not need to be afraid of this because we can still have simple forms of interleavings between TOEs; not all forms of concurrency are of the hard-to-handle, low-level kind. Disjoint-access parallelism (e.g., partitioning an array and processing the disjoint partitions in parallel) is a good example of a case that has plenty of TOEs but is still easy to understand for

[1] While the TM TS has not been fully implemented yet, GCC already has support for the core functionality for a couple of years now.

programmers. Another example is that for all of us, having a dialogue with another person is a completely natural thing to do.

Applied to coroutines, this means in my opinion that we need to understand coroutines as TOEs. On the level of program logic, there is concurrency in the sense of that a coroutine is an *individual* flow of control in the program, so matching the standard's definition of a TOE. I do not think that it is helpful to try to hide this fact by taking the mutual exclusion property of a coroutine and its callers and use it as a reason to merge the coroutine and the caller into one TOE.

Thus, when creating a coroutine and calling it for the first time, we conceptually spawn a TOE that will execute the separate flow of control that the coroutine is meant to represent (or that even is the coroutine). Constructing a `std::thread` will also spawn a TOE, but a TOE of a different kind with different execution properties (e.g., it is guaranteed to execute steps eventually irrespective of what other TOEs do, which is not the case for coroutines).

Understanding coroutines as TOEs is important for three reasons. First, SG1 has discussed many parallelism and concurrency abstractions that create TOEs: Parallel algorithms from the Parallelism TS, executors, task regions, etc. Those TOEs have execution properties that are often lighter-weight than threads, for example regarding aspects such as forward progress guarantees, thread-specific state, or how call stacks are implemented (see below for more details on these). Those potentially lighter-weight properties can be common across TOEs spawned by very different high-level abstractions (e.g., same weaker-than-`std::thread` forward progress guarantees for parallel-vector loops and generators, see below for details).

Second, there often is no single right choice for what kind of TOE to spawn from a higher-level abstraction. For example, which stack implementation does one really need for a coroutine? Or which support for thread-local storage is required? This applies to coroutines and generators as well.

Third, TOEs can be considered the basic entity of computation, when looking at the implementation level and control over where something executes. The executor proposals try to provide programmers with such control, and while it is not always obvious from the programming interfaces, often this control happens by tuning or managing TOEs (e.g., through thread pools). Thus, if coroutines are TOEs, it becomes cleaner to apply executor features to them as well. Note that while often it may be most natural to resume a coroutine using the same compute resource as the caller, this isn't necessarily always best: Imagine a generator that needs to access lots of data to provide a result; this generator is most efficient to run on a CPU close to the accessed data in terms of the memory hierarchy. A similar example can be made in the space of accelerators. Also note that the basic scheme of interaction with the coroutine (e.g., the interleaving) is still the same in this case, so where the coroutine's TOE is executed is really an orthogonal property.

Defining different kinds of TOEs thus gives use a foundation upon which higher-level constructs can be specified and built, including constructs such as coroutines. Having this common base makes it easier for us and easier to grasp for users.

Of course, coroutines are not just TOEs in the sense of the TOE being independent of

anything else. Like other abstractions such as parallel loops that spawn TOEs, coroutines give additional guarantees such as mutual exclusion between the coroutine's TOE and the TOE represented by the caller of the coroutine.

2.1 Examples for specific properties of execution

Before looking at the stack implementation aspect in detail next, I want to give examples using some of the other properties.

Forward progress guarantees of TOEs (or of execution agents) are discussed in P0072R0. It is interesting to see that a lock-step execution of a vector-parallel loop and a typical coroutine execution can both be characterized as non-preemptive, collaborative scheduling. In terms of forward progress, the TOEs spawned by the loop and the TOE representing the coroutine all are weakly parallel TOEs as defined by P0072R0, with boost blocking by either the TOE calling the loop or the coroutine. Thus, one set of properties can describe the TOEs spawned from different abstractions.

Thread-specific state such as thread-local storage (TLS) or lock ownership are another example: SG1 has been discussing how (and whether at all) TOEs used for parallelism should support TLS (see P0072R0 for more background). There is no single right answer because while TLS might not be easily supportable on an accelerator, it sometimes is just required by code; yet seldomly does it seem necessary to run all TLS constructors and destructors for such TOEs, as would be required by `std::thread`. Right now, the coroutine proposals and the resumable function proposal make one specific choice, and they state this rather implicitly (e.g., through stating that a thread used to execute code could change, or just through leaving open which thread executes the code). It would be better in my opinion to provide choice to programmers, so that they can pick the TLS semantics they really need.

Lock ownership is a similar example to TLS. Should a generator inherit and modify the lock ownership of the TOE that called it most recently, or should it have it's own set of lock ownerships? There does not seem to be one right answer to this because it really depends on what the program tries to accomplish.

3 Call stack implementations

The major points that characterize a TOE with non-preemptive suspension are:

1. What happens before and after suspension,
2. How hand-off or hand-shake happen between this TOE and other TOEs on creation, suspension, and termination of this TOE, and
3. Whether there are any constraints on the code that is running in this TOE.

Such a TOE is then used by, say, a coroutine with a specific interface. In the discussions I had so far on this topic, I got the impression that for some people, the coroutine

interface is tied to a specific choice regarding the three points above and vice versa—which seems to match with the separate vertical coroutine proposals that we currently have on the table.

I do not think this has to be the case. While I can agree that some combinations might be easier to implement and allow one to optimize this or that slightly, all a coroutine basically needs is a non-preemptively suspendable TOE.

Furthermore, and as the discussions in the committee show, there are different use cases for coroutines that evaluate the choices made regarding the three points above differently. For example, in some use cases it is more important to not rely on inter-procedural compiler transforms whereas in others it could be vital to call functions without having to duplicate and modify them at the source code level.

Thus, I think it would be helpful to users if they would have choice regarding how the call stack and suspension implementation used for a coroutine looks like; the more flexibility we have in the coroutine building blocks, the better.

This would allow for *unifying* the coroutine proposals. To back up my claim that this is possible, I will next outline how the call stack implementation can provide the properties that the programmer is asking for and allow a programmer to combine them.

Stackful coroutines (e.g., N4232) allow the coroutine TOE to call arbitrary functions and support suspension points in functions called by arbitrary functions. Thus, they give the impression of a "normal" TOE that can just call any code. Suspension points are transparent as far as the calleable code is concerned.[2] Side stacks of various forms or normal OS threads are valid implementations.

Resumable functions (e.g., N4499) are a lower-level mechanism because they require suspension points to be treated specially in the calling code, including functions that call other functions that may suspend. Thus, suspension is not transparent but requires functions to be annotated and use the `await` keyword on *all* function calls that may result in a suspension. The goal of this is to enable a specific call stack implementation, namely one that compresses the space required for the stack by (1) keeping only the essential live-variable information for each stack frame on the side, (2) rolling back all stack frames before suspension (i.e., returning from those functions), and (3) reconstructing stack frames after suspension using the live-variable information. Compiler support is required, but only of the intra-procedural kind.

So, simplified, resumable functions have the advantage of a potential performance improvement (depending upon use, however, they are not always faster), while stackful coroutines have the advantage of minimizing impact on existing code and improving the likelihood of code reuse.

3.1 Bridging the gap between stackful and resumable

Given that both options have their uses and advantages, it would be good if programmers do not have to choose between those two extremes. We can make this happen through relying on a bit more compiler support. In particular, we want the compiler to generate

[2] But they are not transparent regarding forward progress guarantees and thread-specific state, depending on which choices the specific coroutine feature makes regarding these execution properties.

code that uses the approach of resumable functions internally while putting as little requirements on programmers as possible to actually change the source code.

Thus, what we need the compiler to do is to analyze the call graph starting at the outermost function of the coroutine (i.e., the one that is called initially), and look for suspension points (e.g., `await` keywords). This extends the intra-procedural analysis that is already required by the resumable function proposal[3] to an inter-procedural analysis. Then, for all non-resumable functions that may be suspended, the compiler needs to clone those functions and transform the clones into resumable ones. This basically is similar to the compiler transformation already required by resumable functions, only that the compiler needs to treat calls to analyzed-as-resumable functions like if they had an `await` attached to them; additionally, the compiler needs to adapt the return type of each clone.

Such compiler support needs to be implemented, but is not magic either. The Transactional Memory TS is a precedent for requiring such support, so if a compiler implements or wants to implement this TS, it does or will have to do something very similar already. For transactions, we require special code to be generated to enable the use of software TM implementations. Because we do not want the programmer to have to do this, and do not even want to require programmers to annotate all functions that may be called from a transaction, we require the compiler to do this job as good as it can. The `optimized_for_synchronized` attribute is an even closer analogy.

There are difficulties with inter-procedural analyses (e.g., if function bodies are not available), but the TM TS shows how they can be dealt with. Furthermore, working around these difficulties is even easier in our case because strictly speaking, using resumable functions internally is an optimization—the implementation can always use a side stack implementation or similar if it cannot analyze a part of the call graph. The side stack can either be used for the whole coroutine, or even for parts of it so that any runtime/space overheads due to side stacks only occur if the coroutine executes functions in the unanalyzable part of the call graph. If the latter, the resumable function can hook into the side stack suspension mechanism to also suspend the resumable function if the side stack part of the coroutine is suspended.

We can make it even easier for the compiler by requiring an annotation for functions that are meant to be used as resumable functions, so that the compiler doesn't need to do the inter-procedural analyses looking for await keywords. All it then has to do for functions with such annotations is to treat all calls to annotated-as-resumable functions as if they had an `await` attached to them.[4]

Besides the resumable-to-stackful calls already covered above, calling a resumable from a normal or stackfil function is also straightforward. If the calling TOE is not a coroutine TOE, it just blocks for completion of the resumable function. If it is a (non-resumable, stackful) coroutine TOE, it can instead detect when the resumable functions

[3] Note that this depends on whether functions that contain `await` need to be annotated or not; in the discussion of resumable functions in Lenexa, some people wanted the annotations whereas others thought they were unnecessary.

[4] The rules for annotations can be specified so that the compiler will always be aware of whether a function is annotated-as-resumable; see the TM TS for an example.

returns a future with an intermediate result after suspension, and suspend itself when that happens; when resumed, the side stack part would be resumed first, followed by the resumable part.