

CPC

Compiling threads to events efficiently

Gabriel Kerneis*

* *Laboratoire PPS, Université Denis Diderot, Paris, France*
<kerneis@pps.jussieu.fr>

ABSTRACT

We present Continuation Passing C (CPC), a programming language designed for writing concurrent systems. It features very lightweight threads, both cooperatively or preemptively scheduled, which are compiled to highly-efficient event-loop code. Its compilation passes have been proven correct. CPC has been used to write Hekate, a BitTorrent seeder able to sustain thousands of simultaneously connected peers.

KEYWORDS: source-to-source compilation; concurrency; continuation passing style

1 Introduction

Continuation Passing C (CPC) is a programming language designed for writing concurrent systems, developed by Juliusz Chroboczek and Gabriel Kerneis. The CPC programmer manipulates very lightweight threads, choosing whether they should be cooperatively or preemptively scheduled at any given point; the CPC program is processed by the CPC translator, which produces highly efficient event-loop code. This approach gives the best of both worlds: the relative convenience of programming with threads, and the low memory usage of event-loop code.

To translate threads into events, CPC uses a technique known as *conversion into Continuation Passing Style* (CPS) and several other transformations, including *lambda-lifting*. We have proved their correctness[Ker08], which had never been studied earlier in the context of an imperative language like C.

The current implementation of CPC has been used to write Hekate, a BitTorrent seeder designed to handle millions of simultaneous torrents and tens of thousands of simultaneously connected peers. Benchmarking a number of webservers shows that CPC code can be as efficient as the fastest thread libraries and the most carefully hand-written event-driven code available.

2 The CPC language

CPC extends the C language with *CPC threads*. A CPC thread represents a thread of control operating in the shared memory environment of the current process, like threads provided by other programming languages and by common thread libraries for C.

Unlike threads in most other programming languages, CPC threads can be realised either as lightweight data structures cooperatively scheduled by the (user-space) CPC scheduler, or as native system threads, and therefore preemptively scheduled by the operating system kernel. Transition from one scheduling mode to the other is controlled by the programmer, and can happen at any time during the life-time of a thread.

We consider the cooperative mode as the “normal” state of a CPC thread. Using cooperative threads by default makes programs simpler to write and debug: there is no need for locking mechanisms in critical sections and the scheduling is deterministic. Preemptive threads are useful for calling I/O functions that have no non-blocking equivalent (for example `gethostbyname` under Unix), or for implementing compute-bound code that wants to use multiple processors or processor cores. However, they are more expensive than cooperative threads, and, being preemptively scheduled, are more difficult to use correctly.

The CPC programmer must explicitly mark interruptible functions with the `cps` keyword; each call to a *cps* function introduces a cooperation point. Since calling an interruptible function introduces a cooperation point, the translator enforces that every function calling a *cps* function is itself *cps*. In effect, native C functions are directly callable by *cps* functions but not the opposite.

3 Outline of the compilation scheme

The CPC translator is structured as a series of source-to-source transformations:

- making the flow of control explicit, which introduces `goto` statements[NMRW02];
- converting `goto` to tail calls, which introduces local functions[SS76];
- lambda-lifting, to remove local functions[Joh85];
- CPS conversion, to introduce continuations[Plo75];
- linking to the runtime, to schedule threads using continuations[HF84].

The lambda-lifting and CPS conversion passes are well-known compilation techniques for functional languages but raise a number of issues when applied to an imperative language, like C.

In the general case, they require to box mutated variables, hindering performance significantly since constant variables barely exist in C.

We have proved, however, that lambda-lifting tail-called functions – such as those introduced by the `goto` elimination pass – avoids the need for boxing. Only some variables, whose address is retained across a cooperation point, need to be boxed at all. It is our intuition that such variables are rare enough in typical programs to keep boxing at a reasonable level.

In Hekate, the most substantial program written with CPC so far, less than 5% of the local variables are effectively boxed by the current (very naive) implementation of the translator.

This experimental result confirms the intuition about efficiency underlying the compilation scheme. It is to be noted, however, that Hekate is not necessarily representative of code that would be written by a programmer not familiar with the internals of CPC, its developers being the authors of CPC and two of their students.

4 Performance evaluation

We have measured the efficiency of code produced by CPC with both micro and macrobenchmarks.

Efficiency of the runtime library Our set of microbenchmarks consists in spawning a large number of threads and measuring how well basic operations scale. It confirms that CPC threads are very lightweight: a million of them can be spawned in 110 ms on a machine with a 3.16GHz CPU, allocating 75MB of memory. Synchronisation is very cheap too: 50 ms are necessary to suspend them all on a condition variable, and another 50 to awake them.

We discovered during these benchmarks the importance of optimising the runtime library. To this end, separating the translation passes and the runtime scheduling system allows to plug CPC into existing event-loops and threadpools, respectively for the cooperative and preemptive mode. Even though the current implementation focuses on a simple runtime, favoring readability over efficiency, we successfully integrated CPC into the libev event-loop[Leh10], yielding even faster results.

Translation overhead While testing the efficiency of various primitives is useful to optimise the runtime library, macrobenchmarks are necessary to measure the amount of overhead introduced by the translation from threaded to event-driven code.

We have written a set of toy web servers (less than 200 lines each) that share the exact same structure: a single thread or process waits for incoming connections, and spawns a new thread or process as soon as one is accepted. Because of this simple structure, these servers can be directly compared, and we are able to benchmark the underlying implementation of concurrency rather than the implementation of the web server. We wanted to stick to a simple, easily reproducible, setup which would show how well the studied web servers handle many simultaneous requests. Hence, we opted for a raw benchmark, consisting in downloading a small file many times and measuring the response time.

Figure 1 presents the results of our experiment. It plots the average serving time per request against the number of concurrent requests; a smaller slope indicates a faster server. It shows that CPC code is as efficient as the fastest thread libraries we tested. A more in-depth analysis of this benchmark is available in a technical report[KC09].

5 Hekate, BitTorrent seeder written in CPC

CPC has been used to write a substantial piece of software: Hekate, a BitTorrent seeder. Two B.Sc. students spent a few months, supervised by the authors of CPC, designing and implementing this peer-to-peer client targeted at delivering thousands of torrents to hundreds of clients.

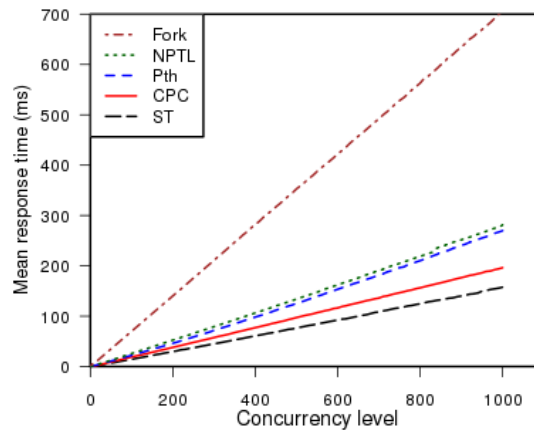


Figure 1: Web servers comparison. Featuring servers based, in increasing order of efficiency, on the `fork` system call, the Linux *Native Posix Threads Library*, the *Gnu Pth* library, CPC and the *State Threads* library.

Hekate consists in 3,000 lines of CPC code — one third of which is plain, sequential C code and the rest is threaded, cooperative, code. It uses two external libraries: a blocking one (`curl`) to perform HTTP requests and a cooperative one (Chroboczek’s DHT) for the distributed hashtable extension to the protocol. Interfacing with such code is made easy due to the ability to use preemptive as well as cooperative threads, respectively.

We use two threads per client: one for reading requests, the other one to push the chunks of files back to the client. Given the cheapness of CPC threads, this model scales to hundreds of clients without any problem.

When too many clients are connected, the BitTorrent protocol suggests to put some of them in a waiting, *choked*, state: the related threads are suspended and awoken in a round-robin fashion using condition variables, which turns out to be a perfectly suited primitive for such an architecture.

We have found that CPC allows a pleasant style of concurrent programming which scales well. Using Hekate to seed the updates of the World of Warcraft software, distributed by Blizzard via BitTorrent, we served hundred of clients for days, reaching up to 5 MB of data per second, without ever raising Hekate’s load above 1% of the CPU.

References

- [HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM.
- [Joh85] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, Nancy, France, 1985. Springer-Verlag New York, Inc.
- [KC09] Gabriel Kerneis and Juliusz Chroboczek. Are events fast? Technical report, 2009.
- [Ker08] Gabriel Kerneis. CPC, des threads coopératifs par passage de continuations. M.Sc. thesis, 2008.
- [Leh10] Marc Lehmann. The libev manual, 2010. Available at <http://cvs.schmorp.de/libev/ev.pod>.
- [NMRW02] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, pages 213–228, 2002.
- [Plo75] G. D. Plotkin. Call-by-name, call-by-value and the [lambda]-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.
- [SS76] Guy L Steele and Gerald J Sussman. Lambda: The ultimate imperative. Technical report, Massachusetts Institute of Technology, 1976.