Automatic pipeline parallelism for event-loops

Etienne Brodu
etienne.brodu@insa-lyon.fr
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA,
F-69621, Villeurbanne, France

Stéphane Frénot
stephane.frenot@insa-lyon.fr
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA,
F-69621, Villeurbanne, France

Frédéric Oblé
frederic.oble@worldline.com
Worldline
53 avenue Paul Krüger - CS 60195
69624 Villeurbanne Cedex

Abstract

The popularity of Javascript recently exploded. Most implementations of Javascript present an event-loop design. Node.js is an example. We believe this design is more efficient to build web applications than the classical thread approach [2]. We argue that an event-loop is a pipeline executing on a single-core. It relieves the developer from the synchronization burden, and is linearly scalable. But it limits the implementation to a single machine. Physics currently prevents us from building CPU with faster clocks. So, when the desired throughput augments, the machine struggles and the average latency increases.

Eventually, the only solution to keep a reasonable latency is to use multiple machines. A particularly efficient design to leverage this parallelism is to slice an application into stages to form a parallel pipeline [3]. However, slicing an application into stages is a costly operation when done manually. Every stage must be balanced to fit on a machine so as to avoid bottlenecks leading to wasted resources. Moreover, for important changes in the application, the slicing must be reconsidered to keep the balance.

We propose to automatically isolate stages in the application logic. The resulting application can be distributed over a network of machines. We believe this work will later allow to automatically balance stages to maximize throughput with the available resources.

An event loop invokes handlers to react on events. Handlers end without return, they asynchronously trigger the next handlers. The call stacks of two handlers are disjoints. Thus, it is possible to parallelize their execution, as long as causality is preserved. Every event handlers becomes a stage in the pipeline.

In a mono-thread execution model like Node.js, the memory is global. The atomic execution of handlers removes the need for synchronization and isolation [1]. The memory holds both the communications between the handlers, and their state. To parallelize the handlers, we isolate the state of each handler to reproduce exclusivity ; and we identify their communications to reproduce a one way flow of messages.

We built a first incomplete compiler as a proof of concept. After successful results with custom applications, we are building a complete compiler to transform real applications.

Références

[1] A Adya, J Howell et M Theimer. “Cooperative Task Management Without Manual Stack Management.” In : *USENIX Annual Technical Conference* (2002).

[2] Kai Lei, Yining Ma et Zhi Tan. “Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js”. In : *2014 IEEE 17th International Conference on Computational Science and Engineering*. IEEE, déc. 2014, p. 661–668. doi : 10.1109/CSE.2014.142.

[3] M Welsh, SD Gribble, EA Brewer et D Culler. *A design framework for highly concurrent systems.* 2000.

Automatic pipeline parallelism for event-loops.

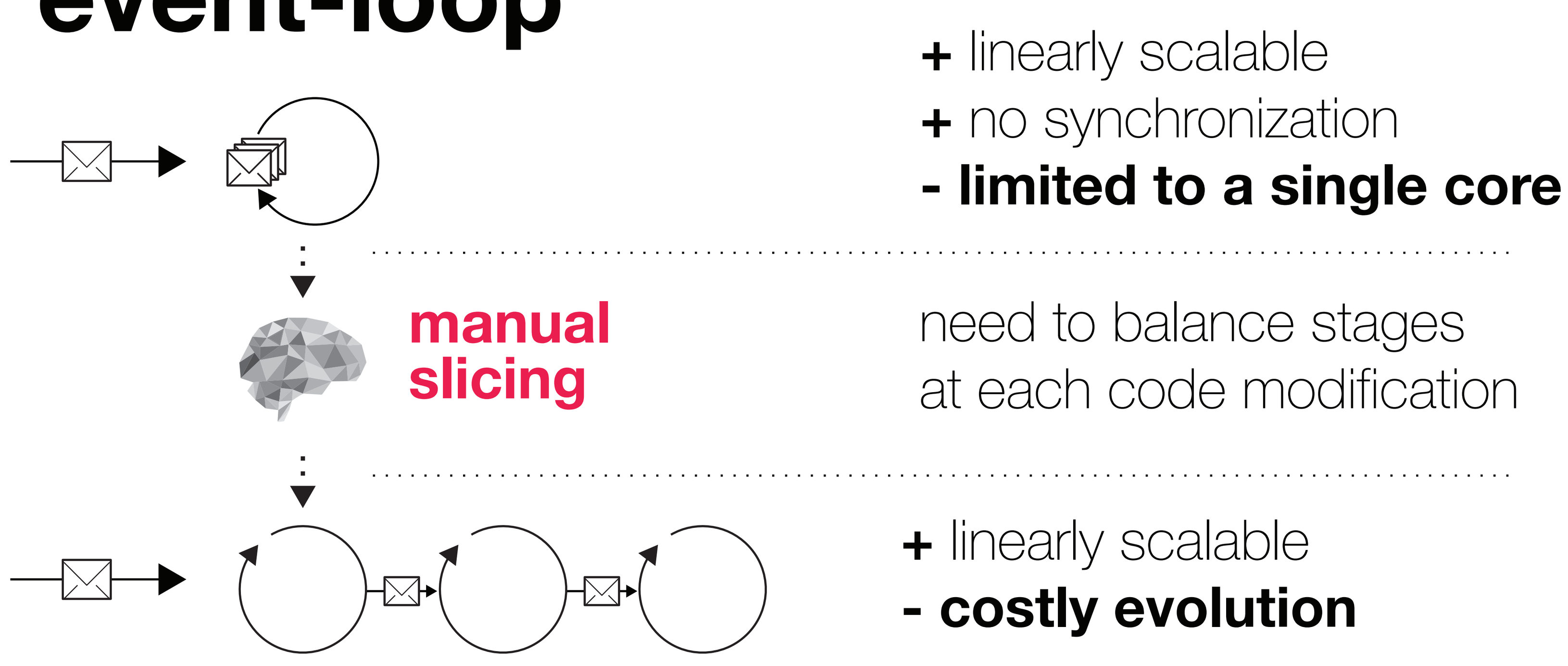
Etienne Brodu etienne.brodu@insa-lyon.fr

Stéphane Frénot stephane.frenot@insa-lyon.fr

Frédéric Oblé frederic.oble@worldline.com

An event-loop is a pipeline on a single core.

event-loop



pipeline parallelism

problematic

an event-loop is limited, and manually switching to pipeline parallelism is costly.

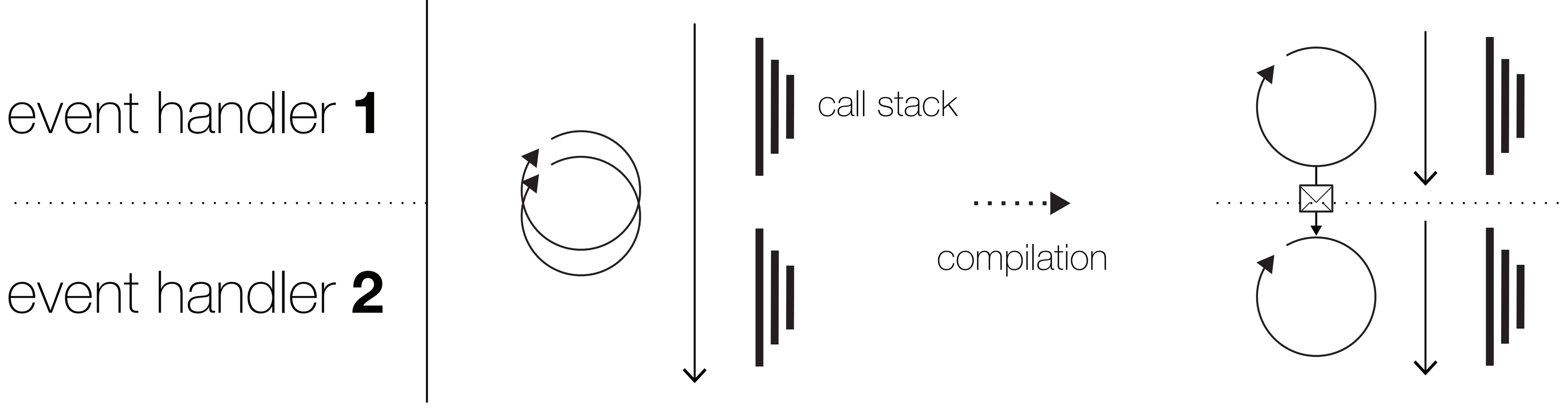
proposition

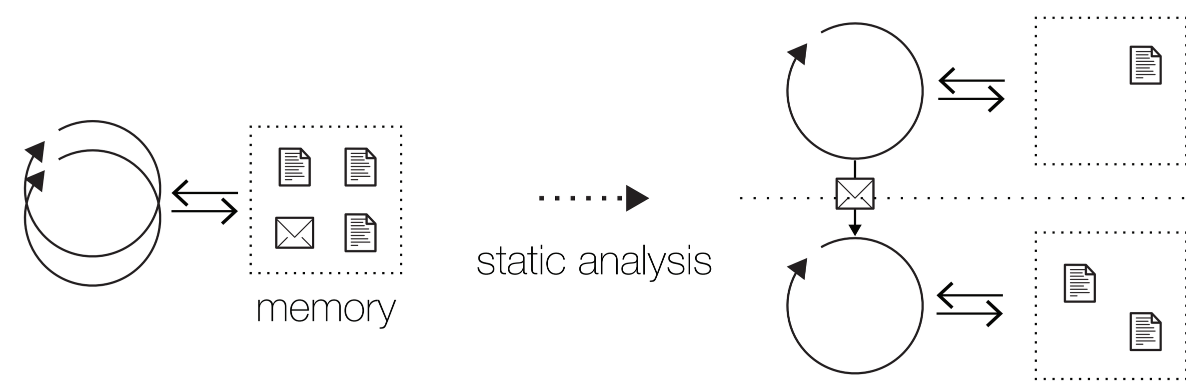
We propose to automate the slicing process at compile time.

event-loop assures
execution independence

&

static analysis provides
memory independence





compilation example

source[5]

```
var app = require('express')(),
    fs = require('fs'),
    count = 0;

app.get('/', function handler(req, res){
  fs.readFile(__filename, function reply(err, data){
    count += 1;
    var code = ('' + data)
                     .replace(/\n/g, '<br>')
                     .replace(/ /g, ' ');
    res.send(err
           || 'downloaded ' + count +
            ' times<br><br><code>' +
              code + '</code>');
  });
});

app.listen(8080);
console.log('>> listening 8080');
```

target[5]

```
flx source.js {}
>> handler-1000 [res]
  var app = require('express')(),
      fs = require('fs'),
      count = 0;
  app.get('/', >> handler-1000);
  app.listen(8080);
  console.log('>> listening 8080');
```

req, res

```
flx handler-1000 {fs}
-> reply-1001 [res]
  function handler(req, res) {
    fs.readFile(__filename, -> reply-1001);
  }
```

res, err, data

```
flx reply-1001 {count}
-> null
  function reply(err, data) {
    count += 1;
    var code = ('' + data)
                     .replace(/\n/g, '<br>')
                     .replace(/ /g, ' ');
    res.send(err
           || 'downloaded ' + count +
            ' times<br><br><code>' +
           code + '</code>');
  }
```


[1] A Adya, J Howell and M Theimer. "Cooperative Task Management Without Manual Stack Management." In : USENIX Annual Technical Conference (2002).

[2] JR von Behren, J Condit and EA Brewer. "Why Events Are a Bad Idea (for High-Concurrency Servers)." In : HotOS (2003).

[3] J Ousterhout. "Why threads are a bad idea (for most purposes)". Presentation given at the 1996 Usenix Annual Technical Conference (1996).

[4] M. Welsh, S. Gribble, E. Brewer, and D. Culler. A Design Framework for Highly Concurrent Systems. CS Technical Report UCB/CSD-00-1108, University of California, Berkeley, October 2000.

[5] flx-example: https://github.com/etnbrd/flx-example/tree/1.0. Accessed: 2014-08-22.