

# 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]. An event-loop uses a single-core execution, and a global memory. It relieves the developer from the synchronization burden, but limits the implementation to a single machine. When the desired throughput augments, latency increases. Physics currently prevents us from building CPU with faster clocks. 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 is reconsidered to keep the balance.

We argue that an event-loop is similar to a parallel pipeline executing on a single-core. 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 disjoint. 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. But 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.