Continuous Monitoring of OCaml Applications using Runtime Events

Sadiq Jaffer Patrick Ferris

September 16, 2022

Abstract

The upcoming 5.0 release of OCaml includes a new runtime tracing system designed for continuous monitoring of OCaml applications called Runtime Events. It enables very low overhead, *programmatic* access to performance data emitted by the OCaml runtime and is designed to be extensible to application-generated events. This talk focuses on the implementation of Runtime Events and the user experience of writing applications exploiting this new feature.

Continuous Monitoring

Continuous monitoring is the continuous collecting of application performance data and using it for health monitoring and ad-hoc analysis. This can be very valuable for identifying performance and reliability issues in applications because when deployed they often behave very differently from development and testing environments. In order to make continuous monitoring practical and inexpensive, a low overhead means of gathering and consuming runtime and application events is required. Runtime Events aims to serve this purpose for OCaml.

Existing Monitoring Tools

Present day OCaml supports runtime tracing with *Eventlog*. Runtime tracing tools monitor key statistics about the OCaml runtime like execution time in various phases of garbage collection and the number of words allocated in the minor and major heaps. *Eventlog* has two caveats that make using it for continuous monitoring hard. First, it writes out events sequentially to disk which is problematic for long-running programs. Second, it is only available when a program is compiled and linked with the instrumented runtime. In contrast, Runtime Events does not require a different runtime to support most of the same events that *Eventlog* does. This means any OCaml program can be executed with the Runtime Events environment variable set and events will be continuously written to the Runtime Events shared ring buffers.

Runtime Events

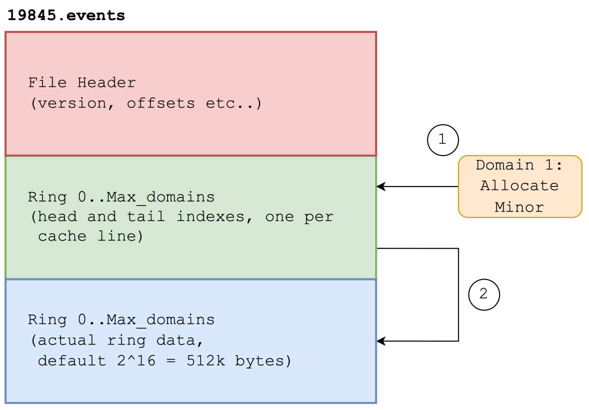


Figure 1: Runtime Events File Format

The new Runtime Events feature uses memory-mapped ring buffers to record per-domain events. A domain is OCaml's unit of parallelism. Each ring buffer is structured as a flight recorder, overwriting old data when there is insufficient space to write new events. Thorough benchmarking has shown the event probe overhead when not enabled is negligible.

The on-disk format of the ring buffer file for events, is relatively simple. A header stores general information such as a versioning and the offsets into the rest of the file. Following that is a metadata section that stores header and tail pointers and offsets for each ring buffer. Last are the ring buffers themselves, one per runnable domain. The format is laid out so it can be stored sparsely and the disk usage scales with the number of running domains. Whilst a program is executing, there are various points in the OCaml runtime that trigger events that are recorded. The domain is identified for the event and this is used to read the ring buffer's head and tail, label 1 in Figure **??**. The event is then added to the ring buffer itself with appropriate checks for sufficient capacity in the buffer, label 2 in Figure **??**. The head and tail entries are padded to ensure they do not share cache lines.

Recording runtime events is very efficient, in the order of tens of nanoseconds on a modern PC. In benchmarks with the upstream compiler, Runtime Events

Variant	Geomean
Eventring disabled	1.000133
Eventring enabled	1.000913

Figure 2: Geometric mean for change in the perf stat count of the number of retired instructions

resulted an 0.01% increase in instructions when not enabled and 0.09% when enabled.

Monitoring Applications

The Runtime Events library provides intuitive APIs for building applications in both OCaml and C. Applications could be simple, filtering and transporting events into other systems (such as Graphite, Prometheus or Jaeger). They may also be richer, offering OCaml-specific functionality not found in other observability systems and provide their own terminal or web browser-based interface for better data visualisation.

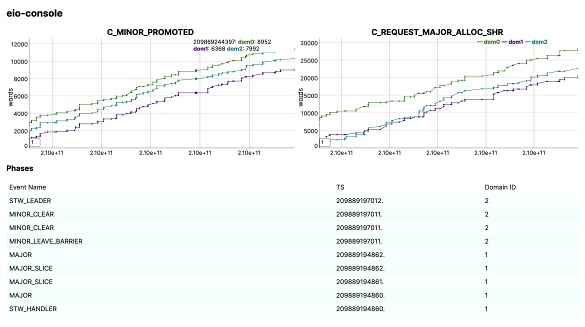
```ocaml
open Runtime_events

let tracing path_pid () =
  let count = ref 0 in
  let c = create_cursor path_pid in
  let rc _domain_id _ts counter value =
    match counter with
    | EV_C_REQUEST_MAJOR_ALLOC_SHR →
      count := !count + value;
      Printf.printf "Major: %i\r%!" !count
    | _ → ()
  in
  let cbs =
    Callbacks.create ~runtime_counter:rc ()
  in
  while true do
    ignore (read_poll c cbs None);
    Unix.sleepf 0.5
  done
```

This simple tracing function prints all of the requests to allocate using *alloc_shr* which allocates a block in the heap.

Richer applications may choose to dynamically visualise the data in a more informative way. *eio-console*[1] is an experimental, browser-based application that monitors live programs with runtime events being transported to the browser via an open WebSocket.

The screenshot of the live dashboard shows plotting the counters for multiple domains of a simple, parallel fibonacci program. Over time each domain allocates more memory as we would expect.

[1]https://github.com/patricoferris/eio-console



Event Name	TS	Domain ID
STW_LEADER	209889197012.	2
MINOR_CLEAR	209889197011.	2
MINOR_CLEAR	209889197011.	2
MINOR_LEAVE_BARRIER	209889197011.	2
MAJOR	209889194862.	1
MAJOR_SLICE	209889194862.	1
MAJOR_SLICE	209889194861.	1
MAJOR	209889194860.	1
STW_HANDLER	209889194860.	1

Figure 3: Eio-console screenshot

Future Work

At present only events from the OCaml runtime are recorded but the design allows for application-created events and an implementation[2] has been proposed for merging in to the runtime. This would allow end-users to build efficient logging infrastructure layered on top of the existing runtime events that is easily enabled for better debugging and monitoring.

[2]https://github.com/ocaml/ocaml/pull/11474