Mono’s New Profiler API

Motivation: Problems with the old API

- Impossible to reconfigure profiling options at runtime
- Some interactions between multiple profilers were unclear
- ‘All or nothing’ instrumentations, e.g. enter/leave
- Certain aspects of the API were confusing to users
- Had events which were never fired
- Had many deprecated or outright broken features and events
- Adding new events took more effort than it should

Design Goals (I)

- All profiling features must be toggleable at runtime
  - Except for stuff that affects code generation
  - Reconfiguration of features should be async safe
  - Some features must still be enabled at startup to be used later, but are enabled in a latent (i.e. idle) state
- Disabled or idle features must be cheap
  - Examples:
    - We don’t want to gather GC roots if no roots callback is installed
    - Managed allocators must be usable regardless of profiling options
    - Sampling thread should be sleeping when sampling is in idle mode

Design Goals (II)

- Reasonably future proof
  - Adding a new event should basically be a 2-line change
  - For features that currently must be requested at startup, we must have the ability to make them available at runtime in the future without fundamentally changing the API
    - A future version of Mono might support recompiling methods for allocation profiling, enter/leave instrumentation, code coverage, etc
- Clear migration path for existing users
  - If you were using a function in the old API, there should be an obvious equivalent in the new API
    - Except for broken or deprecated stuff
  - We refuse to load a profiler using the old API so users will be aware that their module must be updated

Old vs New: Statistical sampling

```
void mono_profiler_startup (const char *desc)
{
    MonoProfiler *prof = g_new0 (MonoProfiler, 1);

    mono_profiler_install (prof, shutdown_cb);

    mono_profiler_set_statistical_mode (MONO_PROFILER_STAT_MODE_PROCESS, 100);
    mono_profiler_install_statistical (sample_cb);
    mono_profiler_set_events (MONO_PROFILE_STATISTICAL);
}
```

```
void mono_profiler_init_test (const char *desc)
{
    MonoProfiler *prof = g_new0 (MonoProfiler, 1);
    MonoProfilerHandle handle = mono_profiler_create (prof);

    mono_profiler_enable_sampling (handle);
    mono_profiler_set_runtime_shutdown_end_callback (handle, shutdown_cb);
    mono_profiler_set_sample_hit_callback (handle, sample_cb);

    if (!mono_profiler_set_sample_mode (handle, MONO_PROFILER_SAMPLE_MODE_PROCESS, 100))
        fprintf (stderr, “Warning: Another profiler controls sampling parameters.");
}
```

Old vs New: The key differences

- A handle is now used to alter settings for an installed profiler
- Each kind of event now has a function to set a callback
- User no longer has to set event flags; this is done internally
- The sampling feature must be enabled at startup
- The profiler will know if another profiler controls sampling
  - Whichever profiler enables sampling first gets control over parameters
- The entry point symbol name for profiler modules has changed

New Feature: Dynamic reconfiguration

- Almost all profiling features can now be reconfigured at runtime
  - Event callbacks can be set or unset at any point
    - If a callback was set at startup and is later unset, the runtime won't waste time collecting the data for that callback, just as if it wasn't set in the first place
    - Changing callbacks is inherently racy; thread B might still raise an event even though thread A just unset the callback
  - Sampling mode (none, process time, real time) and sampling frequency (Hz) can be changed by the controlling profiler

New Feature: Enter/leave filters

- Similar to code coverage filters
- Allows a profiler to decide which methods to instrument
  - Can decide whether to instrument the prologue, epilogue, or both
    - Can distinguish between normal method exits, exceptional exits, and tail calls
  - If any profiler wants a method instrumented, it will happen, even if others didn’t request it, so a profiler should be prepared to deal with this
- Can significantly reduce the performance impact of enter/leave instrumentation depending on the use case

New Feature: Call context introspection

- An extension of enter/leave instrumentation
- Allows enter/leave callbacks to access the stack frame of the instrumented method
  - Can access the `this` reference, arguments, locals, and the return value
    - Return value unavailable for tail calls (i.e. CIL `tail.` prefix and `jmp` opcode)
  - Stack frame accessed by using same debug info used by the debugger agent
- Enables arbitrary instrumentation of well-known methods in the base class library
  - Profiling of networking, thread pool, reflection, etc usage

New Feature: Instrumented managed allocators (I)

- Managed allocators can now be used when profiling
- Done with a profiler variant of the managed allocator
  - Checks the number of installed allocation event callbacks in an ‘unlikely’ branch at the end of the allocator
  - Calls into the profiler API with the allocated object if a non-zero amount of callbacks are installed
- Significantly improves performance when latent allocation profiling is enabled

New Feature: Instrumented managed allocators (II)

```
$ cat alloc.cs
class Program {
        static void Main () {
                for (var i = 0; i < 500000000; i++)
                        new object ();
        }
}
$ mcs alloc.cs
$ time MONO_GC_DEBUG=no-managed-allocator mono --profile=log alloc.exe
real    0m30.507s
user    0m30.141s
sys     0m0.266s
$ time mono --profile=log alloc.exe
real    0m2.939s
user    0m2.688s
sys     0m0.234s
```

- Running the log profiler with allocation profiling in latent state is much faster for allocation-heavy programs

Removed Feature: Old code coverage mode

- Implemented in `mono_arch_output_basic_block` within each backend
  - Not implemented for most architectures as nobody did the porting effort
- Entirely superseded by the IR-based code coverage mode
- If one profiler requested this mode and another profiler requested the IR-based mode, things would break horribly

Removed Feature: Call chain profiling

- A variant of statistical sampling
- Used various fragile strategies to collect a native backtrace on a sample hit signal
  - Native: Manually unwind the stack based on a `MonoContext`
    - Used heuristics and was not portable at all; only worked on x86
  - glibc: Use the `backtrace` function
    - Not guaranteed to be signal safe, or even produce useful results in a signal handler
  - Managed: Unwind the stack based on a `MonoContext` and `mono_find_jit_info`
    - Less fragile but also less useful; could simply use `mono_stack_walk_async_safe` instead
- Will probably use libunwind if we explore this feature again

Implementation: Event definitions (I)

- An event is now defined with a single line in profiler-events.h:

```
...
MONO_PROFILER_EVENT_2(gc_event, GCEvent, MonoProfilerGCEvent, event, uint32_t, generation)
MONO_PROFILER_EVENT_1(gc_allocation, GCAllocation, MonoObject *, object)
MONO_PROFILER_EVENT_2(gc_moves, GCMoves, MonoObject *const *, objects, uint64_t, count)
...
```

- profiler-events.h is a parameterized header
  - Similar to mini/mini-ops.h, metadata/icall-def.h
  - Included in profiler.h, profiler.c, profiler-private.h to generate the entire event callback API
- Glorified macro hack, but pretty convenient

Implementation: Event definitions (II)

- Example code generated for the `gc_allocation` event:

```
typedef void (*MonoProfilerGCAllocationCallback) (MonoProfiler *prof, MonoObject *object);

MONO_API void mono_profiler_set_gc_allocation_callback (MonoProfilerHandle handle,
    MonoProfilerGCAllocationCallback cb)
{
    update_callback (&handle->gc_allocation_cb, cb, &mono_profiler_state.gc_allocation_count);
}

void mono_profiler_raise_gc_allocation (MonoObject *object)
{
    for (MonoProfilerHandle h = mono_profiler_state.profilers; h; h = h->next) {
        MonoProfilerGCAllocationCallback cb = h->gc_allocation_cb;
        if (cb)
            cb (h->prof, object);
    }
}
```

Implementation: Callback registration (I)

- The runtime needs a fast way to determine whether it should raise an event, or for more expensive events, gather the needed data
  - We basically need a replacement for event flags in the old API
    - But users should not have to bother with this stuff
- Solution: An atomic counter for each event
  - Incremented when we install a non-`NULL` callback, decremented when we install a `NULL` callback
    - Not actually that simple: Could lead to unbalanced increments/decrements if a profiler installs more non-`NULL` callbacks than `NULL` callbacks

Implementation: Callback registration (II)

```
static void
update_callback (volatile void **location, void *new, volatile uint32_t *counter)
{
    void *old;

    do {
        old = InterlockedReadPointer (location);
    } while (InterlockedCompareExchangePointer (location, new, old) != old);

    if (old)
        InterlockedDecrement (counter);

    if (new)
        InterlockedIncrement (counter);
}
```

- Ensures that the counter reflects the exact amount of installed callbacks
- Slightly racy: NULL callback can be installed while counter is non-zero

Implementation: Raising events (I)

```
#define MONO_PROFILER_ENABLED(name) \
    G_UNLIKELY (mono_profiler_state.name ## _count)

#define MONO_PROFILER_RAISE(name, args) \
    do { \
        if (MONO_PROFILER_ENABLED (name)) \
            mono_profiler_raise_ ## name args; \
    } while (0)
```

- Runtime code can use `MONO_PROFILER_ENABLED` explicitly when gathering data for an event is expensive
- Most runtime code should use `MONO_PROFILER_RAISE` since, for the vast majority of events, the data is readily available

Implementation: Raising events (II)

```
void
sgen_client_collecting_minor (SgenPointerQueue *fin_ready_queue, SgenPointerQueue *critical_fin_queue)
{
    if (MONO_PROFILER_ENABLED (gc_roots))
        report_registered_roots ();

    if (MONO_PROFILER_ENABLED (gc_roots))
        report_finalizer_roots (fin_ready_queue, critical_fin_queue);
}
```

```
static MonoObject *
do_runtime_invoke (MonoMethod *method, void *obj, void **params, MonoObject **exc, MonoError *error)
{
    ...

    MONO_PROFILER_RAISE (method_begin_invoke, (method));

    result = callbacks.runtime_invoke (method, obj, params, exc, error);

    MONO_PROFILER_RAISE (method_end_invoke, (method));

    ...
}
```

Implementation: Statistical sampling (I)

- Sampler thread sends signals to all live threads at a configured frequency (Hz) based on a configured clock (process or real time)
  - Process time not supported on all OSs; falls back to real time
  - Uses real time signals when the OS supports them
  - Uses real time scheduling when available (usually requires root)
  - Skips special threads (e.g. tools threads, SGen worker threads)
- Signal handler raises the sample hit profiler event with instruction pointer and signal context
  - Profiler can then use e.g. `mono_stack_walk_async_safe`

Implementation: Statistical sampling (II)

- Room for improvement
  - We should explore making real time scheduling work without root
    - OS X seems to allow it, but it's complicated
  - OSs without real time signals drop many of our sampling signals
    - E.g. OS X has no support for signal queuing; just uses a bit mask
      - Only one instance of each signal can be in flight at any given time
    - This results in most signals going to the main thread
    - Possible solutions:
      - Use Mach thread suspend/resume APIs to do sampling (Apple platforms only)
      - Figure out some kind of reliable round-robin signal strategy (portable)

Implementation: Instrumented managed allocators

- Basically a regular allocator but with the CIL equivalent of:

```
MONO_PROFILER_RAISE (gc_allocation, (obj));
```

- The event is raised after we exit the critical region and just before we return to normal managed code
  - Event callbacks can execute arbitrary code which might e.g. take a lock
- When we load an AOT image, all allocator wrappers will be redirected to the instrumented variant if allocation profiling was requested at startup
  - This way, we don’t have to disable managed allocators when we want to do allocation profiling on a FullAOT target such as iOS

Implementation: Enter/leave instrumentation

- Implemented as JIT internal calls
  - Enter: Emitted in the entry basic block
  - Leave: Emitted before `ret`, `jmp`, and `tail.call` instructions
- Call contexts are allocated on the stack and passed to the callback
  - In the JIT: Filled out by a special `fill_prof_call_ctx` IR instruction
    - Copies stack pointer, frame pointer, and callee-saved registers to the context
    - Also stores an address to the return value for an epilogue context
    - Values are retrieved from the state stored in the call context based on debug info
    - Not supported in the LLVM backend (so no watchOS support)
  - In the interpreter: We store a pointer to the `InterpFrame` in the call context
    - Values are retrieved from the `InterpFrame`; no need for debug info

Implementation: Code coverage (I)

- A coverage info structure is allocated for each method
  - Contains a series of coverage entries, where each entry consists of a CIL offset and a coverage counter
- Every basic block in the method is instrumented with a tiny piece of code that increments the counter in the coverage entry corresponding to the basic block's CIL offset
- A profiler requests the results of coverage instrumentation on shutdown, where we map each coverage entry to a source code location based on debug info

Implementation: Code coverage (II)

- Room for improvement
  - We embed the address of the counter directly in IR, which is incompatible with AOT
    - In particular, this means coverage analysis is unavailable on iOS, watchOS, etc
  - We allocate as many coverage entries as there are CIL instructions in a method, which is quite wasteful
  - The fact that a basic block has been entered does not necessarily mean that the entire basic block will be executed
    - This is because the JIT uses extended basic blocks, i.e. any instruction can throw an exception and thus exit the block

Conclusion

- The new profiler API has a more clearly defined API contract, gives more control to users, and has paved the way for new features that would be very difficult to implement in the old API
- Very easy to evolve the new API
- The API is close to having feature parity with the CLR API
- First and hopefully last time we break profiler API compatibility
- New API ships in Mono 5.6