# Implementing Balanced Polling for OCaml

Dhruv Makwana, Modern Compiler Design (L25) Project Report

```
(* Small arrays are always allocated
   in the young generation *)
let newa = [| 0 |] in
(* Large arrays are always allocated
   in the old generation *)
let olda = Array.make 1000 0 in
let arr = Array.make 1000 olda in
while true do
  (* make an old-gen field point to
     an old-gen object *)
  arr.(0) <- olda;
  (* now make it point to a young
     object, a ref is recorded *)
  arr.(0) <- newa
done
```

Fig. 1. No safepoints in loop and growing ref-table.

***Abstract*—Native OCaml compilation does not guarantee the existence of safepoints on all execution paths. This means valid programs can crash or experience delayed thread communication. I fix this by inserting safepoints during code generation following Feeley's algorithm [1]. As a result, more OCaml programs can run natively and there now exists an upper bound on thread communication latency for Multi-core OCaml. However, this comes with a potential increase in their execution times.**

***Index Terms*—OCaml, polling, signal, multi-core, garbage-collection, threads, synchronisation**

## I. Introduction

A *safepoint* in an OCaml program is a check on flags used for thread communication and garbage-collection. In its current implementation, allocating an object is deemed a safepoint, since, in most OCaml programs, allocations occur frequently. Unfortunately, there exist pathological pure OCaml programs which, *when compiled to native code*, can spend large amounts of time executing sequence of instructions without an intervening safepoint (for bytecode-interpreted programs, the interpreter decides when to execute safepoints). Programs of this form could postpone signal handling and thread preemption/synchronisation indefinitely (this affects Multi-core OCaml as well, since the latter requires timely message-passing between threads). A commented example is given in Figure 1.

Similar pathological programs could occur inside numerical-library routines, if writing and copying large arrays/matrices. Therefore, handling them would provide immediate benefits to certain types of applications. For example, this would enable the implementation of engine-racing: if a flag used for synchronisation signifies an interrupt then we want a guarantee that all relevant threads notice and act on this signal in a reasonable amount of time and not run until completion – this is not possible in Multi-core OCaml if one of the engines is running in a loop without safepoints.

Further problems arise when such natively compiled OCaml programs trigger the creation of pointers from an old generation object to a young generation one. These events are recorded in a *ref-table* that grows beyond bound on the assumption that it will be garbage collected. Hence, if we are executing such natively compiled programs without safepoints, this results in a *Fatal error: ref_table overflow*. Fixing this bug would allow the execution of more valid OCaml programs, such as the given example.

### A. Contributions

In this report:

- I describe the balanced polling algorithm and my implementation of it (Section II). I describe the mapping from the algorithm's control-flow constructs to those of OCaml's C-- intermediate representation.
- I evaluate my implementation by measuring the size of the executables it produces and the execution times of those executables for some benchmarks and show that the end result is tolerable for small programs, with a 0.6-1.2% increase in executable size but a potential increase in execution time (Section III).
- I discuss the further work required to merge this implementation into upstream OCaml and possible improvements to the algorithm, and why they may or may not make a significant difference (Section IV).

## II. Implementation

### A. Example

Before I go any further, I will illustrate the desired outcome with an example. The examples are presented in

```
(let
  ; Set up variables
  (newa/1002 (alloc block-hdr(1024) 1)
   olda/1003 (extcall "caml_make_vect" ..)
   arr/1004 (extcall "caml_make_vect" ..))
  (catch
    (loop
      ; Check array bounds and write
      (checkbound (load_mut int (..)) ..)
      (extcall "caml_modify" ..)
      ; Check array bounds and write
      (checkbound (load_mut int (..)) ..)
      (extcall "caml_modify" ..))
  with(1) []))
```

Fig. 2. Translation of Figure 1 to C--.

```
(let
  ; Set up variables
  (newa/1002 (alloc block-hdr(1024) 1)
   olda/1003 (extcall "caml_make_vect" ..)
   arr/1004 (extcall "caml_make_vect"
       (seq (alloc block-hdr(0)) 2001) ..))
  (catch
    (loop
      ; Check array bounds and write
      (checkbound (load_mut int (..)) ..)
      (extcall "caml_modify" ..)
      ; Poll, check array bounds...
      (checkbound
        (seq (alloc block-hdr(0))
             (load_mut int (..))) ..)
      ; and then write
      (extcall "caml_modify" ..))
  with(1) []))
```

Fig. 3. Figure 2 with a safepoint `(alloc block-hdr(0))` inserted in the desired place.

C--: an intermediate representation used for native compilation in OCaml. It is the last hierarchical, expression-based IR and as such, expresses the minimal set of programming language constructs needed to compile to machine code. I defined a safepoint to be the allocation of zero-length float-array (`(alloc block-hdr(0))`), which is already supported by OCaml and does not actually allocate any memory.

First, we see a relatively straightforward mapping of Figure 1 to Figure 2. We note that there are no allocations in the loop and that by construction, the assignments (calls to 'caml_modify') add to the ref-table.

What we would like to do is transform the code in Figure 2 to that shown in Figure 3. Our safepoint `(alloc block-hdr(0))` is in the loop and guarantees that the garbage-collector is at least checked every iteration. Note that inserting one safepoint inside the loop incurs the least possible overhead whilst still being correct.

Although we could naively insert a safepoint at the start of every loop and at every procedure entry and exit, there is more general way to do this that results in a lower polling overhead.

### B. Balanced Polling

Feeley's *balanced polling* is a method of inserting safepoints into compiled code with low overhead and an upper-bound on interrupt latency. This method introduces less overhead and is more general than *call-return polling* which naively places safepoints at *every procedure entry and exit*.

Balanced polling is described with respect to a *tree* containing basic-blocks of instructions per procedure (one entry point, one return point). Branches are only allowed as the last instruction of a basic-block and are categorised into four types:

- *Local branches*, possibly conditional, to basic-blocks within the procedure
- Tail-calls to procedures, labelled *Reductions*
- Non-tail-calls to procedures, labelled *Sub-problems*
- *Returns* from procedures.

Further, the algorithm refers to one dynamic parameter ($\Delta$) and three static parameters ($L_{\text{max}}, E, R$) for determining when to insert a safepoint.

- $\Delta$: upper bound of distance to the most recent safepoint
- $L_{\text{max}}$: maximum number of instructions that can be traversed without inserting a safepoint (a proxy for *Latency*)
- $E$: grace at *Entry* to a procedure
- $R$: upper bound to the value of $\Delta$ just before a tail-call (or *Reduction*), also used for expressing loops

Parameter $E$ is a concession to avoid polling inside a short-lived procedure and is part of the invariant that $\Delta \leq L_{\text{max}} - E$ upon entry to a procedure. Conversely, if $\Delta$ exceeds $L_{\text{max}} - E$ we insert a safepoint. A consequence of this is that (ignoring tail-calls) up to $L_{\text{max}}/E$ sub-problems can occur in a sequence without any safepoints inserted.

To understand the role of parameter $R$, it helps to consider the value of $\Delta$ after a return from a sub-problem with no tail-calls: $\Delta_{\text{ret}} \leq \Delta_{\text{entry}} + E$. Assume the sub-problem has exactly $E$ instructions, so no safepoints are inserted. If there was a tail-call before the return with $E' \leq E$ instructions after the last safepoint was inserted, then $\Delta_{\text{ret}} = \Delta_{\text{entry}} + E + E'$. To enforce an invariant like $\Delta_{\text{ret}} \leq \Delta_{\text{entry}} + E$, we could insert a safepoint before every tail-call, but this would increase overhead considerably if there was a long sequence of tail-calls with little work done between each of them (for example, long chains of specialised functions calling more general

ones with specific arguments). Hence $R$ is a concession to avoid this. Under this, we add a lower bound $\Delta_{\text{ret}} \geq E + R$ and have that up to $(L_{\max} - R)/E$ sub-problems can occur in a sequence without any safepoints inserted.

The complete algorithm is presented in Figure 4.

```
(* Entry *)
Delta = L_max - E

(* Non-branch instruction *)
if (Delta >= L_max - 1) then
    add_interrupt_check ()
    Delta := 0
Delta := Delta + 1

(* Sub-problem Call *)
if (Delta >= L_max - E) then
    add_interrupt_check ()
    Delta := 0
Delta := E + max(R, Delta)

(* Reduction Call *)
if (Delta >= R) then
    add_interrupt_check ()

(* Procedure_Return *)
(* added = true iff there are
   safepoints on path from entry *)
if Delta >= E + R && added then
    add_interrupt_check ()
```

Fig. 4. Compilation rules for balanced polling.

### C. Implementation for C--

A simplified grammar of C-- expressions, ignoring details irrelevant to the balanced polling algorithm, is presented in Figure 5. Its semantics are as follows – let-expressions are strict, they evaluate the bound-expression before the body; tuples (parentheses) are evaluated right-to-left; sequence (curly-braces) are evaluated left-to-right; exit-expressions are returns (with potentially multiple values); catch-expressions are a collection of (potentially) mutually-recursive continuations whose control-flow falls through to the subsequent expression; the apply operation is function application and all other constructs are similar in semantics to regular programming languages such as C or OCaml.

In implementing balanced-polling for C--, I mapped:

- the 'apply' operation to sub-problems;
- the sole looping construct to tail-calls;
- constants, tuples, sequences and all operations (except for 'apply', load and store) to non-branching instructions;
- the 'exit' construct to procedure return, assuming 'added' was always true (there is no feasible way to check this because safepoints are added recursively to the children of a node *before* determining whether a safepoint needs to be inserted for the root as a whole);
- constructs such as if-then-else- and switch-expressions directly onto join-points with the *maximum delta method*, that is, the delta at the join-point is the maximum of delta of all branches to it;
- try-with expressions by mapping the exception-handler as a new sub-problem and then mapping the expression subsequent to it as a join-point with the maximum-delta method;
- the 'catch' construct (which expresses a collection of continuations) as arbitrary control-flow and as such, mapped each continuation-body as a sub-problem, resetting $\Delta$ to $L_{\max} - E$. However, because of C--'s catch-expression semantic (continuations fall-through), I mapped the expression subsequent to them as a join-point with the maximum-delta method.

*expr* ::= *const*
| **let** *ident* **=** *expr* **in** *expr*
| *ident* **:=** *expr*
| (*expr*, **..**)
| **Op** *op expr list*
| {*expr*; **..**}
| **if** *expr* **then** *expr* **else** *expr*
| **switch** *expr* { *int* * *expr array* }
| **loop** *expr*
| **catch** *rec-flag continuation list expr*
| **exit** *expr list*
| **try** *expr* **with** *expr*

*op* ::= **load** *expr* | **store** *expr*
| **apply** *machtype* | **..**

Fig. 5. Simplified grammar of C--.

The implementation of the algorithm was complicated by the existence of critical sections: expressions within which it is illegal to insert a safepoint. Such expressions evaluate to an address (for example, into the middle of an array or a record) rather than a valid OCaml value and are not allowed to be in a live register across an allocation because they are invalid garbage-collection roots. They occur in three places: let-expressions, stores and loads, only when the argument value being computed is an address. In these cases, it is safe to increment $\Delta$, skip over the computation of the address and insert safepoints after it if necessary. In any other case, all three are mapped to non-branching instructions.

## III. Evaluation

First off, this implementation works: the program in Figure 1 no longer crashes. In fact, Figure 3 is extracted from a working implementation of the algorithm in the OCaml native compiler.

### A. Cost on Toy-Programs

However, adding balanced polling will affect OCaml programs that do not expand the ref-table indefinitely in two ways: the size of the native executable produced and the execution time of it. To investigate these effects, I conducted a small series of benchmarks on toy-programs. I calculated each timing as the mean over 20 executions and used Welch's T-Test to assess whether any differences (when compared to no polling) were statically significant. Error-bars represent $\pm\sigma$. These benchmarks are definitely *not* representable of real OCaml programs in the wild, but provide a useful way to effectively isolate some of the effects we wish to study.

Remembering that $L_{\text{max}}$ is a proxy for the maximum latency between safepoints, we can study the effect of varying this parameter on executable size and execution time. I followed the precedent of Feeley's paper and fixed $E$ (the grace at function entry) and $R$ (the largest admissible delta at a tail-call) to be the floor of $L_{\text{max}}/6$. Since the purpose of this report is to study balanced polling in OCaml, rather than showing balanced polling is not that much worse than minimal polling ($L_{\text{max}}$ is arbitrarily large) and better than call-return polling, I did not measure those strategies.

Figure 6 shows that we need not worry too much about executables bloating too much: less than 1.2% for most values of $L_{\text{max}}$ is tolerable for these small programs.

However, the impact on execution times is less clear. One would expect that increasing the maximum allowable latency should decrease the overhead monotonically, but the reality is not that simple. From Figure 7, we can surmise, that *if* there is a change at all, *then* it is typically an increase (differences that were not statistically significant ($p < 0.05$) are marked as 0% overhead without error-bars). *How much* overhead is incurred depends on the value of $L_{\text{max}}$ and the program being measured. For example: all programs perform poorly at $L_{\text{max}} = 144$. Even for the best case values of $L_{\text{max}}$, between 102 and 138, the overhead ranges from 0-20%, except for 'tight': a program designed to exhibit the worst-case effects (65-90% overhead) of polling (it is simply a loop that decrements a positive integer until it equals zero).

The reality is most likely that the *extent* of any overhead of balanced polling is highly-dependent on the application but in its current state, tolerable for small applications. A lot of the measured programs were written in a rather imperative style, used no libraries except for those provided by the compiler distribution and made frequent use of several nested, tight loops – a challenging case for polling. This shortcoming of balanced polling was mentioned in Feeley's paper where a tight loop was shown to have an 80% overhead. Its OCaml translation (the green line in Figure 7), has overheads of 65-90% for values of $L_{\text{max}}$ between 102 and 138.

### B. Cost on Owl: a Numerical Library

To address the shortcomings of the previous benchmarks, I (after hours grappling with Opam 2's undocumented support of compiler switches for local, in-development compilers) installed *Owl* (and its several dependencies) with my modified OCaml compiler. Owl is a new numerical library, written in OCaml that provides an excellent opportunity to study the effects of balanced polling (with $L_{\text{max}} = 90$) on real-world, loop-intensive applications. Figure 8 shows the initially surprising results: none of of the problems exhibited a statistically significant change in execution time. However, in retrospect, given that Owl makes heavy-use of external calls to C/Fortran libraries (OpenBLAS) for low-level operations, this should not be too surprising.

### C. Cost of Safepoint

I attempted to understand how much time a safepoint took to execute, in the best case. To do so, I ran a tight-loop program that simply count backwards from 17,179,869,183 to 0 five times, compiled with and without safepoints. Clearly this is a best case scenario because branch-prediction would predict the path very well and the program and its data are small enough to fit in L1-cache. With safepoints, each loop iteration took $(2.9 \pm 0.1)$ns compared to $(1.75 \pm 0.02)$ns without ($p = 2 \times 10^{-5}$). This suggests the safepoint took, *in the best case* 1.16 ns to execute, which is a little difficult to believe given that a safepoint is translated to a function call, to the code in Figure 9 (I even checked the assembly to see if the call was being inlined; it was not).

## IV. Further Work

There are several ways of *potentially* improving the current implementation. I say 'potentially' because as Figures 7 and 8 show, it is difficult to be sure about performance before measuring. Nonetheless, as it stands currently, my implementation is a C-- to C-- transformation and inserts another traversal of every top-level

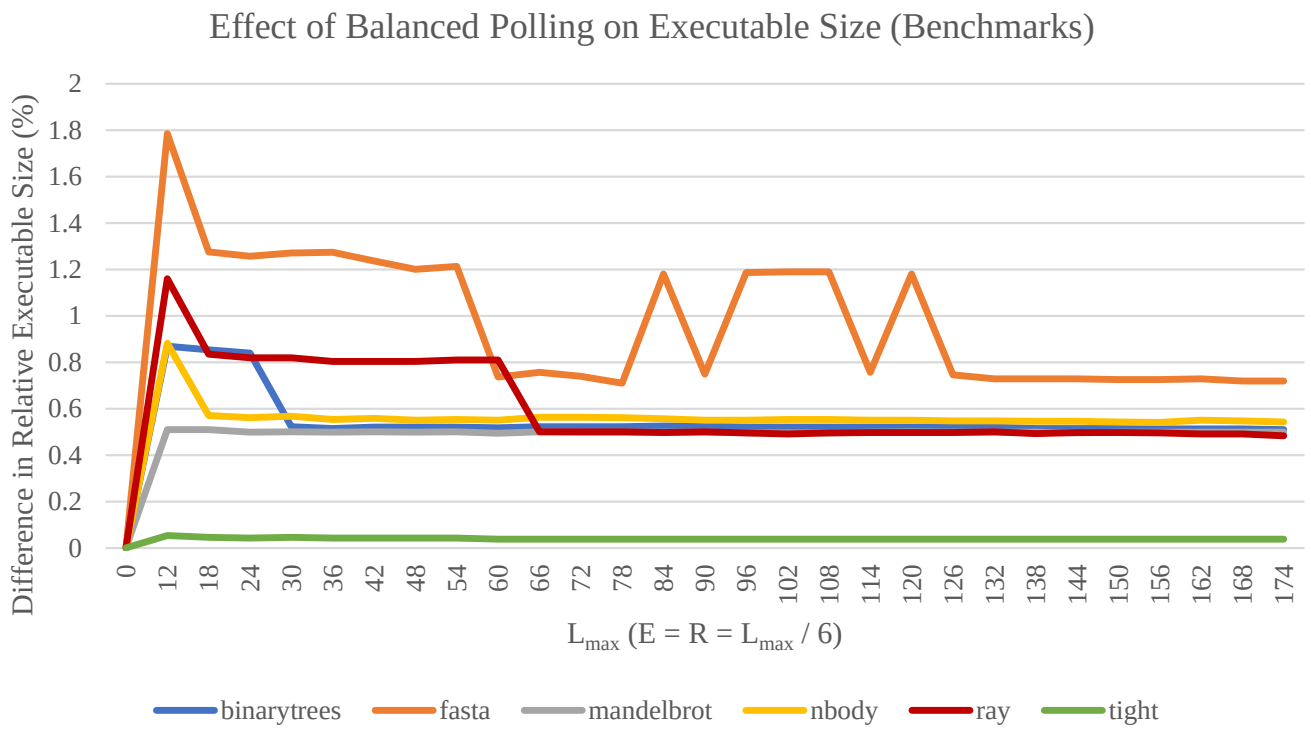


Fig. 6. I used toy-programs from benchmarksgame.alioth.debian.org/ and www.ffconsultancy.com/languages/ray_tracer to investigate the effect of varying $L_{\max}$ on executable size, relative to no balanced polling (marked on the graph as $L_{\max} = 0$). Overall, the effects are small, typically increasing executable size by 0.6-1.2%.

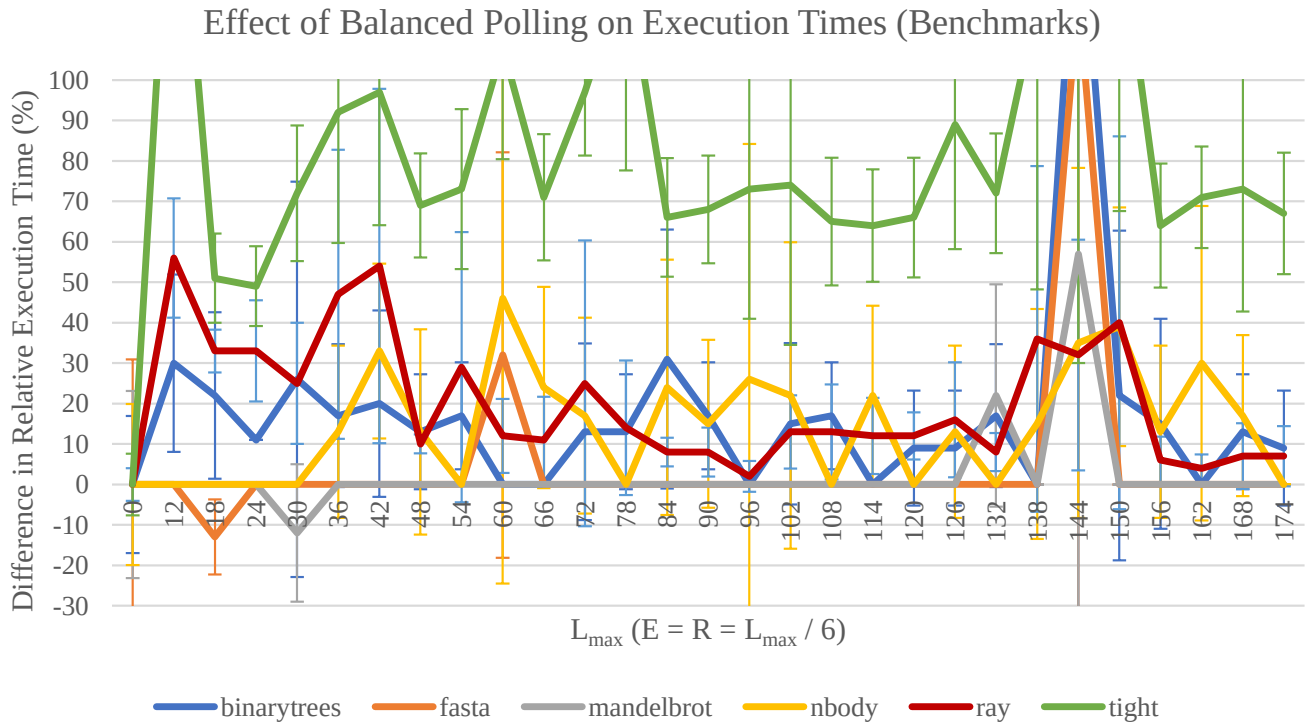


Fig. 7. Error-bars represent $\pm\sigma$. In contrast to Figure 6, here, the effect of varying $L_{\max}$ on execution time, relative to no balanced polling (marked on the graph as $L_{\max} = 0$) is erratic and pronounced (differences that were not statistically significant ($p < 0.05$) are marked as 0% overhead without error-bars).

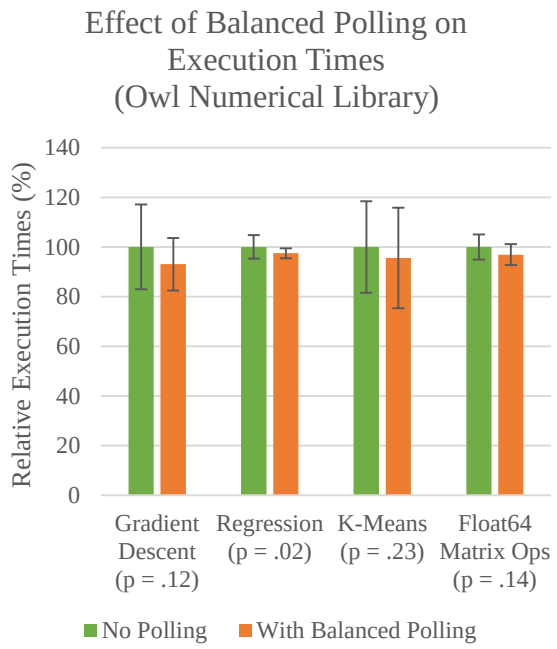


Fig. 8. Error-bars represent $\pm\sigma$. All types of problems I tested within the Owl numerical library showed no statistically significant change in execution times with balanced polling.

function declaration before it is linearised into machine code. I did not measure the impact of this additional phase on the performance of the compiler itself, but it could be significant.

As far as the efficiency of the generated code, there is a trade-off to be explored in complexity of the code to insert safepoints versus maximum allowable latency. If we continue to operate on C--, then there might be a gain in only adding safepoints *on paths* from root to leaves that have no other safepoints. However, this would require *two* traversals of the C-- expression: one full traversal to determine the paths and the second partial traversal, only along the paths calculated by the first, to execute the balanced polling algorithm on them. This approach would also sacrifice the guarantee of $L_{\max}$ as an upper bound for the maxim number of C-- expressions that can be evaluated without an intervening safepoint (a proxy for *latency*).

Another approach would be to implement this algorithm on an IR lower than C--. After C--, OCaml's native compiler has both generic machine-code and linearised machine code IRs and I suspect it might be possible to modify either the *selectgen* (C-- to Mach) or the *linearise* (Mach to LinearMach) pass to emit safepoints alongside the instructions following this algorithm. I think this approach would be precarious but feasible for anyone familiar with those passes.

```
FUNCTION(caml_allocN)
    CFI_STARTPROC
    PROFILE_CAML
    subl    G(caml_young_ptr), %eax
    /* eax = size - caml_young_ptr */
    negl    %eax
    /* eax = caml_young_ptr - size */
    cmpl    G(caml_young_limit), %eax
    jb      LBL(103)
    movl    %eax, G(caml_young_ptr)
    ret
LBL(103):
    subl    G(caml_young_ptr), %eax
    /* eax = - size */
    negl    %eax
    /* eax = size */
    pushl   %eax; CFI_ADJUST(4)
    /* save desired size */
    subl    %eax, G(caml_young_ptr)
    /* must update young_ptr */
    movl    4(%esp), %eax
    movl    %eax, G(caml_last_return_address)
    leal    8(%esp), %eax
    movl    %eax, G(caml_bottom_of_stack)
    ALIGN_STACK(8)
    call    LBL(105)
    UNDO_ALIGN_STACK(8)
    /* recover desired size */
    popl    %eax; CFI_ADJUST(-4)
    jmp     G(caml_allocN)
    CFI_ENDPROC
```

Fig. 9. x86 assembler (before pre-processing) for caml_allocN, called with N = EAX = 0.

On a more fundamental level, it may be worthwhile adding an actual polling construct to insert as a safepoint instead of relying on a zero-length float array. Somewhat surprisingly, there are assembly-routines, similar to that of Figure 9, hardcoded for one-, two-, and three-word allocations (indicating constant propagation is *not* used). Although allocations in OCaml are cheap (bump-the-pointer), a dedicated safepoint could be implemented more cheaply and clearly, perhaps even having two-kinds of polls: one for frequently visited locations (using a conditional branch, as is the case currently) and one for infrequently used ones (dereferencing a null-pointer, catching the resulting SIGSEGV and then executing the safepoint code).

## V. Conclusion

Overall, implementing balanced polling for OCaml is a conceptually simple task that turned out to be a challenging and educational investigation into what would be required to do it *well*. Feeley's balanced polling

algorithm and my implementation of it as a C-- to C-- pass is a good starting point, but requires more work on the efficiency of both the implementation and the generated code before it can reasonably be submitted as a patch to upstream OCaml.

Regardless of the implementation, there do exist tight-loop OCaml programs that will experience the worst-case of polling overheads. However, right now, the benefits of an upper-bound on thread-communication latency, especially for Multi-core OCaml, outweigh its costs.

## Acknowledgements

I would like to thank Stephen Dolan for suggesting this project, the example program in Figure 1 and his patient help in explaining C-- and 'bad GC root tagged'; Mark Shinwell for explaining the semantics of C--'s 'catch' construct; and David Chisnall for feedback on a draft of this report. Any errors, are of course, mine.

## Opam 2 and Compiler Switching

The usual business of opam and compiler-conf no longer works. For posterity, here are some instructions from github.com/ocaml/opam/issues/2531#issuecomment-235377458, thanks to Gabriel Radanne:

```
opam switch install --no-switch --empty safepoints
eval $(opam config env --switch=safepoints)
cd /to/my/ocaml
./configure --prefix $(opam config var prefix)
make -j world.opt
# Ensure cat VERSION == 4.06.0+safepoints
opam pin add ocaml-variants.4.06.0+safepoints -k path .
# In the editor:
opam-version: "2.0"
name: "ocaml-variants"
version: "4.06.0+safepoints"
synopsis: "4.06 with safepoints"
depends: ["base-unix" "base-bigarray" "base-threads"]
flags: compiler
setenv: CAML_LD_LIBRARY_PATH = "%{lib}%/stublibs"
install: [make "install"]
url { src: "file:///to/my/ocaml" }
```

## References

[1] M. Feeley. *Polling efficiently on stock hardware*. Proceedings of the conference on Functional programming languages and computer architecture. ACM, 1993.