Flexible On-Stack Replacement in LLVM (Artifact)

Daniele Cono D'Elia   Camil Demetrescu

Dept. of Computer, Control, and Management Engineering
Sapienza University of Rome, Italy
{delia,demetres}@dis.uniroma1.it

A. Artifact Description

A.1 Abstract

OSRKit is a library that enables On-Stack Replacement (OSR) at arbitrary places in LLVM IR code. The artifact is designed to explore how OSRKit can instrument IR code to support OSR transitions in the LLVM MCJIT runtime environment. A running example is presented based on the `isord` case study discussed in Section 3. We also support repeating all the experiments presented in Section 5. The artifact includes an interactive VM called TinyVM for loading, inspecting, instrumenting, and executing IR code. The package is a preconfigured Oracle VirtualBox VM.

A.2 Description

The main component of the artifact is an interactive VM called TinyVM built on top of the LLVM MCJIT runtime environment and the OSRKit library. The VM provides an interactive environment for IR manipulation, JIT-compilation, and execution of functions either generated at run-time or loaded from disk: for instance, it allows the user to insert OSR points in loaded functions, run optimization passes on them, display their CFGs, and repeatedly invoke a function for a specified amount of times. TinyVM supports dynamic library loading and linking, and includes a helper component for MCJIT that simplifies tasks such as handling multiple IR modules, symbol resolution in presence of multiple versions of a function, and tracking machine-level generated code and data objects.

TinyVM is located in `/home/osrkit/Desktop/tinyvm/` and runs a case-insensitive command-line interpreter:

```
osrkit@osrkit-AE:~/Desktop/tinyvm$ tinyvm
Welcome! Enter 'HELP' to show the list of commands.
TinyVM>
```

Use `HELP` to print basic documentation on how to use the shell. Usage scenarios are discussed in A.5.

A.2.1 Check-list (artifact meta information)

- **Program:** `shootout` C benchmarks (included, Sep 2015).
- **Compilation:** LLVM 3.6.2 (release build).
- **Run-time environment:** Linux (version 3.x), no root password required.
- **Hardware:** x86-64 CPU.
- **Run-time state:** Cache-sensitive (performance measurements only).
- **Output:** Measurements are output to console.
- **Experiment workflow:** Invoke scripts and perform a few manual steps.
- **Publicly available?** Yes.

A.2.2 How Delivered

The artifact ships as an Oracle VirtualBox 5 Appliance. The latest version of the code is available at `https://github.com/dcdelia/tinyvm`.

A.2.3 Hardware Dependencies

An x86-64 platform is required.

A.2.4 Software Dependencies

The artifact was tested in Oracle VirtualBox 5.0.10.

A.3 Installation

To install the artifact, just import the appliance in Oracle VirtualBox, which installs Linux LXLE. Open the `README` file on the `Desktop` folder for further info on the artifact and the Linux distribution. To install TinyVM on bare hardware, check out the `cgo2016` branch from the GitHub repository and run `make` to compile it (LLVM 3.6.2 is required).

A.4 Experiment Workflow

We propose three usage sessions. In the first session, we show how to generate and instrument an LLVM IR code based on the `isord` example presented in Section 3. The second session focuses on how to run the scripts used to generate the performance tables of Section 5 related to questions Q1, Q2, and Q3. The third session addresses question Q4, using third-party software (the MATLAB McVM runtime [21]) that we ported to LLVM 3.6+ and extended with the `feval` optimization technique discussed in Section 4.2.

A.5 Evaluation and Expected Result

A.5.1 Session 1: OSR instrumentation in OSRKit

TinyVM implements a code generator for open OSR points that can dynamically inline function calls to targets that cannot be statically determined. In the example from Figure 4, a comparator function c is passed as argument to function isord, which checks whether an array v of numbers is ordered according to the criterion encoded in c.

To interactively run a dynamic inlining experiment (see Section 3), we provide under the folder tinyvm/isord a C module inline.c with an LLVM IR counterpart inline.ll (generated with clang -S -emit-llvm -O1 inline.c).

We can load the IR module in TinyVM and show the code generated for method isord with:

```
osrkit@osrkit-AE:~/Desktop/tinyvm$ tinyvm
Welcome! Enter 'HELP' to show the list of commands.
TinyVM> LOAD_IR isord/inline.ll
[LOAD] Opening "isord/inline.ll" as IR source file...
TinyVM> DUMP isord
[...]
```

Displayed virtual register names and basic block labels will often differ from those reported in Figure 5, which have been refactored for the sake of readability. In particular, the loop body of isord will look like:

```
.lr.ph:                               ; preds = %2, %0
  %i.01 = phi i64 [ %10, %2 ], [ 1, %0 ]
  %4 = getelementptr inbounds i64* %v, i64 %i.01
  %.sum = add nsw i64 %i.01, -1
  %5 = getelementptr inbounds i64* %v, i64 %.sum
  %6 = bitcast i64* %5 to i8*
  %7 = bitcast i64* %4 to i8*
  %8 = tail call i32 %c(i8* %6, i8* %7) #3
  [...]
```

A ϕ-node %i.01 is used to represent the index of the for loop from the C code, and is set to %10 when reached from the loop header (basic block %2) *after* a loop iteration. In fact, as a result of -O1 optimizations, when n>1 execution jumps from the function entrypoint %0 directly into the loop body, initializing the ϕ-node with 1. Comparator c is invoked with a tail call, storing its return value into virtual register %8.

OSR points can be inserted with the INSERT_OSR command, which allows several combinations of features (see HELP for an exhaustive list). In this session we will modify isord so that when the loop body is entered for the first time, an OSR is fired right away[1]:

```
TinyVM> INSERT_OSR 100 ALWAYS OPEN UPDATE IN isord
                   AT %4 DYN_INLINE %c
```

TinyVM will UPDATE the function as follows: an ALWAYS-true OSR condition is checked before executing instruction %4 to fire an OPEN OSR transition to the DYN_INLINE code generator, which will inline any indirect function call to the function pointer %c. We choose %4 as location for the OSR as it is the first non-ϕ instruction in the loop body; we also hint the LLVM back-end through IR profiling metadata that taking the branch to the %OSR_fire block is 100%-likely.

The IR will now look like:

```
TinyVM> DUMP isord
...
.lr.ph:                               ; preds = %2, %0
  %i.01 = phi i64 [ %10, %2 ], [ 1, %0 ]
  %alwaysOSR = fcmp true double 0.000000e+00,
                                0.000000e+00
  br i1 %alwaysOSR, label %OSR_fire,
                    label %OSR_split, !prof !1

OSR_split:                            ; preds = %.lr.ph
  %4 = getelementptr inbounds i64* %v, i64 %i.01
  %.sum = add nsw i64 %i.01, -1
  [...]

OSR_fire:                             ; preds = %.lr.ph
  %OSRCast = bitcast i32 (i8*, i8*)* %c to i8*
  %OSRRet = call i32 @isord_stub(i8* %OSRCast,
                i64* %v, i64 %n,
                i32 (i8*, i8*)* %c,
                i64 %i.01)
  ret i32 %OSRRet
```

OSRKit has split the %.lr.ph block for the OSR condition, also adding an OSR_fire block to transfer the execution state to isord_stub and eventually return the OSRRet value.

We can now let isord run on an array dynamically initialized by the driver method, which takes as argument the array length to use. The method will also populate it with elements ordered for the comparator in use (see inline.c). For instance, we can ask driver to set up an array of 100000 elements and run isord on it:

```
TinyVM> driver(100000)
Time spent in creating continuation function:
                          0.000252396 seconds
Address of invoked function: 140652750196768
Function being inlined: cmp
Elapsed CPU time: 0 m 0 s 3 ms 417 us 157 ns
               (that is: 0.003417157 seconds)
Evaluated to: 1
```

The method returns 1 as result, indicating that the vector is sorted. Compared to Figure 7, the IR code generated for the OSR continuation function isordto (DUMP isordto) is slightly different, as the MCJIT compiler detects that additional optimizations (e.g., loop strength reduction) are possible and performs them right away[2]. We expect code generated for isord_stub to be identical up to renaming to the IR reported in Figure 6.

To show native code generated by the MCJIT back-end, we can run TinyVM in a debugger with gdb tinyvm and leverage the debugging interface of MCJIT. For instance,

[1] The syntax for inserting an OSR point controlled by a profiling counter is slightly different. For an example, please refer to the script isord/inline-counter.tvm included in TinyVM.

[2] Notice that lowering to native code an IR function in MCJIT (which happens in TinyVM when first executing it) may alter its IR representation.

once driver has been invoked, we can switch to the debugger with CTRL-Z and display the x86-64 code for any JIT-compiled method with:

```
(gdb) disas isordto
Dump of assembler code for function isordto:
   [Base address: 0x00007ffff7ff2000]
   <+0>:   mov    -0x8(%rdi,%rcx,8),%edx
   <+4>:   sub    (%rdi,%rcx,8),%edx
   <+7>:   xor    %eax,%eax
   <+9>:   test   %edx,%edx
   <+11>:  jg     0x7ffff7ff201a <isordto+26>
   <+13>:  inc    %rcx
   <+16>:  mov    $0x1,%eax
   <+21>:  cmp    %rsi,%rcx
   <+24>:  jl     0x7ffff7ff2000 <isordto>
   <+26>:  retq
```

To return to TinyVM, we can use the signal 0 command in gdb (the prompt is not re-printed, but the shell is alive).

A.5.2 Session 2: Performance Figures

The experiments can be repeated by executing scripts on a selection of the shootout benchmarks [7]. Each benchmark was compiled to LLVM IR using clang as described in Section 5.1. For each benchmark X, tinyvm/shootout/X/ contains the unoptimized and optimized (opt -O1) IR code, each in two versions:

- bench and bench-O1: IR code of the benchmark;
- finalAlwaysFire and finalAlwaysFire-O1: IR code of the benchmark preprocessed by slicing the hottest loop into a separate function when needed (see Section 5.2).

Each experiment runs a warm-up phase followed by 10 identical trials. We manually collected the figures from the console output and analyzed them, computing confidence intervals. We show how to run the code using n-body as an example[3]. Times reported in this section have been measured in VirtualBox on an Intel Core i7-4980HQ CPU @ 2.80GHz, a different setup than the one discussed in Section 5.1.

Question Q1. The purpose of the experiment is assessing the impact on code quality due to the presence of OSR points. The first step consists in generating figures for the baseline (uninstrumented) benchmark version. Go to /home/osrkit/Desktop/tinyvm and type:

```
$ tinyvm shootout/scripts/bench/n-body
```

The script is as follows:

```
LOAD_IR shootout/n-body/bench.ll
bench(50000000)
REPEAT 10 bench(50000000)
```

which loads the IR code, performs a warm-up execution of the benchmark, and then 10 repetitions. The experiment duration was $\approx$ 1m, with a time per trial of ≈ 5.725s.

[3] For rev-comp, first run bootstrap.sh in tinyvm/shootout/.

The benchmark with the hottest loop instrumented with a never-firing OSR point can be run with:

```
$ tinyvm shootout/scripts/codeQuality/n-body
```

The script is as follows:

```
LOAD_IR shootout/n-body/bench.ll
INSERT_OSR 5 NEVER OPEN UPDATE IN bench AT %8 CLONE
bench(50000000)
REPEAT 10 bench(50000000)
```

Note that the second line inserts a never-firing open OSR point at basic block %8 labeled with <label>:8 in function bench of file shootout/n-body/bench.ll, using branch weight of 5% as a hint for the LLVM native code generation back-end that OSR firing is very unlikely.

The experiment duration was $\approx$ 1m with a time per trial of $\approx$ 5.673s. The ratio $5.673/5.725 = 0.990$ for n-body is slightly smaller than the one reported in Figure 10 on the Intel Xeon platform. The experiment for building Figure 11 uses scripts in bench-O1 and codeQuality-O1.

Question Q2. This experiment assesses the run-time overhead of an OSR transition by measuring the duration of an always-firing OSR execution and of a never-firing OSR execution, and reporting the difference averaged over the number of fired OSRs (Table 2). The always-firing OSR execution for n-body (unoptimized) is as follows:

```
$ tinyvm shootout/scripts/finalAlwaysFire/n-body
```

which runs:

```
LOAD_IR shootout/n-body/finalAlwaysFire.ll
INSERT_OSR 95 ALWAYS SLVD UPDATE IN advance AT
        %entry TO advance AT %entry AS advance_OSR
bench(50000000)
REPEAT 10 bench(50000000)
```

The second line inserts an always-firing resolved OSR point at the beginning of basic block %entry in function advance of file shootout/n-body/finalAlwaysFire.ll, generating a continuation function called advance_OSR. A branch weight of 95% is given as a hint to the LLVM native code generation back-end that OSR firing is a high-probability event. The time per trial was $\approx$ 5.876s.

The never-firing OSR execution used as baseline is as follows:

```
$ tinyvm shootout/scripts/finalAlwaysFire/
     baseline/n-body
```

with a time per trial of $\approx$ 5.669s. The average time per OSR transition is therefore $(5.876 - 5.669)/50000000 = 4.14 \cdot 10^{-9}$s. Compare this with the result of Table 2.

Question Q3. The third experiment measures the overhead of OSRKit for inserting OSR points and creating a stub or a continuation function. To perform one trial for the open OSR experiment of Table 3, run:

```
$ tinyvm shootout/scripts/instrTime/open/n-body
```

which yielded:

```
Time spent in stub generation: 0.000012835 sec
Time spent in OSR point insertion: 0.000013219 sec
```

One trial for the resolved OSR experiment can be run as follows:

```
$ tinyvm shootout/scripts/instrTime/final/n-body
```

obtaining, e.g.:

```
Time spent in creating cont. func.: 0.000075849 sec
Time spent in OSR point insert.: 0.000009409 sec
```

Notice that in a virtualized environment there may be significant fluctuations in the reported times across different trials, as we rely on a high-resolution timer for measurements[4].

A.5.3 Session 3: `feval` optimization in McVM

McVM is a virtual machine for MATLAB developed at McGill University. As a by-product of our project, we ported it from the LLVM legacy JIT to MCJIT, and later extended it with a new specialization mechanism for `feval` calls. The source code for this version along with the MATLAB benchmarks listed in Section 5.1 are publicly available at `https://github.com/dcdelia/mcvm`.

Experiments reported in Table 4 (Question Q4) can be repeated using a number of scripts provided along with a McVM build in `/home/osrkit/Desktop/mcvm/`.

For each benchmark `X`, `benchmarks/scripts/` contains three MATLAB scripts to use as input for `mcvm`:

- `base/X`: single run of original code (i.e., `feval`-based);
- `direct/X`: single run of code optimized by hand (i.e., with direct calls);
- `many/X`: multiple runs of original code (for code caching).

We manually collected figures from the console output and computed speedups for the different settings. We show how to run the code using `odeRK4` as an example. The platform used to obtain reported numbers is the same as in session 2.

To determine a baseline for speedup computation, we let `mcvm` perform a single run of the original code with no `feval` optimization. Note that we can selectively enable or disable `feval` optimization using the `-jit_feval_opt` flag:

```
$ cd ~/Desktop/mcvm
$ ./mcvm -jit_feval_opt false <
         benchmarks/scripts/base/odeRK4
**********************************************
     McVM - The McLab Virtual Machine v1.0
Visit http://www.sable.mcgill.ca for more info.
**********************************************
>: >: Compiling function: "testSH"
Compiling function: "odeRK4"
Compiling function: "testSHfun"
Compiling function: "rhsSteelHeat"
Compiling function: "testSHfun"
Compiling function: "rhsSteelHeat"
[TOC] Elapsed time: 20.141959 seconds
 t y_RK4
   0.0000  1.000000
  20.0000 227.364633
```

To measure the performance of McVM's code caching mechanism, we let the benchmark run multiple times in the same instance of the VM:

```
$ ./mcvm -jit_feval_opt false <
         benchmarks/scripts/many/odeRK4
```

The experiment duration on our platform was $\approx$ 2m, with an average time per trial of ≈ 19.836s (manually computed by averaging the elapsed time figures from the console, after discarding the warm-up run). The resulting speedup for the base code caching mechanism was thus $20.142/19.836 = 1.015\times$, slightly different than the one reported in column *Base* of Table 4 for the Intel Xeon platform, for which we repeated each experiment 10 times.

We can now set an upper bound for speedups by measuring the running time when the code has been optimized by hand inserting direct calls in place of `feval` instructions:

```
$ ./mcvm < benchmarks/scripts/direct/odeRK4
[...]
>: >: Compiling function: "testSH_direct"
Compiling function: "odeRK4_testSHfun"
Compiling function: "testSHfun"
Compiling function: "rhsSteelHeat"
[TOC] Elapsed time: 7.977169 seconds
 t y_RK4
   0.0000  1.000000
  20.0000 227.364633
```

In this scenario McVM can compile the whole program ahead of time, as `rhsSteelHeat` is not invoked through an `feval` instruction anymore. A comparison of the running times suggests a rough $20.142/7.977 = 2.525\times$ speedup for by-hand optimization w.r.t. the baseline version (compare to column *Direct* in Table 4).

We can now try to assess the speedup from our `feval` optimization technique on `odeRK4`:

```
$ cd ~/Desktop/mcvm
$ ./mcvm -jit_feval_opt true <
         benchmarks/scripts/base/odeRK4
[...]
>: >: Compiling function: "testSH"
Compiling function: "odeRK4"
Compiling and tracking a feval instruction...
Compiling and tracking a feval instruction...
Compiling and tracking a feval instruction...
Compiling and tracking a feval instruction...
Function contains annotated feval instructions!
Compiling function: "testSHfun"
Compiling function: "rhsSteelHeat"
Type conversion required for variable y
Type conversion required for variable $t10
```

[4] `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4503740/`.

```
[TOC] Elapsed time: 8.450570 seconds
 t y_RK4
   0.0000  1.000000
  20.0000 227.364633
```

The execution time ratio between the base version and the optimized code that we JIT-compile is thus $20.142/8.451 = 2.383$ (compare to column *Opt. JIT* in Table 4). Notice that compensation code is generated to perform unboxing of IIR variables `y` and `$t10` ("Type conversion required...") so that execution can correctly resume from the optimized code.

We can finally evaluate the speedup enabled by our code caching mechanism (Section 4.2) for the compilation of continuation functions by running:

```
$ ./mcvm -jit_feval_opt true <
         benchmarks/scripts/many/odeRK4
```

The experiment duration was $\approx$ 1m, with a time per trial of $\approx$ 11.817s (discarding the warm-up run). The resulting speedup w.r.t. is thus $20.142/8.006 = 2.516\times$ (compare to column *Opt. cached* in Table 4).

A.6 Notes

We encourage the reader to experiment with TinyVM, creating IR programs with `clang -S -emit-llvm -O1`, instrumenting them with OSR points, and exploring the generated code. Please bear in mind that TinyVM is a prototype implementation that does not support exercising all the features for VM builders provided by OSRKit. Also, unexpected results may arise: we will be glad to hear about your experience and grateful to receive any bug reports.