CS526 Final Report

David Zmick
Matthew Plant

May 2016

1 Introduction

On Stack Replacement is a mechanism to replace (often at run time) a function F with a modified version of F upon successful evaluation of some arbitrary conditional at any point P in F. The modified version has some extra behavior (or extra optimizations) but still behaves exactly as F would. OSR is often used in virtual machines to avoid expensive optimizations at compile time or load time, but allow them to occur when the virtual machine detects a hot loop or some other hot code path[5]. This technique is often used to improve performance of languages with many high cost/high level features[6]. For example, a virtual function call in a hot loop (Figure 1) requires a `vtable` lookup every iteration through the loop[4]. At runtime, it is simple to discover which function we need to use, then inline all calls to this function (if the loop is hot enough to warrant the additional optimization). Languages like Java, JavaScript[8], and Python can take advantage these optimizations to improve performance.

```
class Animal { virtual void talk() = 0; }
class Cat : public Animal { void talk() override { /* ... */ }
class Dog : public Animal { void talk() override { /* ... */ }

int foo(Animal &a) {
  for (size_t i = 0; i < BIG_ITERS; i++)
    a.talk();
}
```

Figure 1: A high cost virtual function call

One simple example of OSR (the one we discuss in this report) is function pointer inlining. We demonstrate the technique on a simple C library utility, the “is ordered” function (Figure 2) [1]. In this example, we cannot easily (or cheaply) inline the call to the comparator function at compile time, but, if we interrupt the loop, we can determine which comparator is being used, and inline it. Inlining the comparator will expose additional opportunities for optimization.

```
bool isord(int* values, size_t len, bool (*comp)(int, int)) {
  for (size_t i = 1; i < len; i++) {
    if (!comp(values[i-1], values[i])) return false;
  }

  return true;
}
```

Figure 2: “is ordered” example

2 Related Work

The state of the art on stack replacement implementation exists in the Java HotSpot virtual machine.[5] The technique is also used in the V8 JavaScript virtual machine[8]. In LLVM, there is no officially supported OSR implementation, but there is an implementation[9] which accompanies the Ph.D. thesis of D. Elia[2] and some of the same author's earlier papers[1]. One of the first implementations of on stack replacement existed in the Self programming language[7].

3 Implementation - High Level

Our OSR function pointer inlining implementation is implemented as an LLVM Module Pass. From a very high level, we first run an LLVM IR module through a very small number of optimizations (mem2reg, instruction combining, and CFG simplification). Then, we pass the module through the OSR pass. The OSR pass will instrument the code to call an OSR continuation generator function. This generator function is defined in C++, manipulating LLVM objects. The generator returns a new function which is then called by the OSR code.

4 Implementation - Details

Inserting OSR points can be broken down into two cases; closed OSR points, where the replacement function is known upon insertion: and open OSR points, where the function is to be compiled upon successful evaluation of the conditional. Inserting an open OSR point in LLVM is as simple as inserting a conditional branch to a function call. The optimizer will do the rest of the heavy lifting. In the case of inserting open OSR points, the function we would have inserted in the closed cases becomes a generating function: a function that takes as input the LLVM representation of the current functions and LLVM values that are Live at that point and returns an address to a JIT compiled function that is somehow a modified version of the original. After the generator function returns we can call the function with the live variables and continue the function. This function is then called and execution of the user program procceds.

There is further disambiguation between the generator function we've described above and the the general concept of continuation function generation. We expect continuation function generation to provide a more malleable LLVM function type, which the generator then JITs. The generator function may also choose to provide some sort of optimization to the original function, or a copy thereof before passing to continuation function generation or the JIT phase. Due to the variability in the concept the generator function is a disparate component.

For function inlining, we generate a continuation function, then run some optimizations on the generated function. We run Aggressive Dead Code elimination to propagate constants forward and remove the dead code. After running ADCE, most function calls to function pointers which are live at the OSR point will be made using the value passed as an argument to the continuation function. We perform a reverse look up of the llvm `Function*` using the runtime value of the function pointer, then copy the function into the module which we are generating. This function is copied with the attribute `always inline`. Then, we run the `always inline` pass on the module, followed by all of the `-O3` optimization passes.

4.1 Continuation Function Generation

Continuation function generation is non trivial. A continuation function needs to resume execution from any point in the original function with no change in state. At a low level a lot of bookkeeping is involved, especially that which is particular to LLVM. At the high level, generation proceeds first by copying the original function's instructions into the continuation function. We then must insert a header into the function that simply unconditionally jumps to the place in the function we want to resume at. From this point we need to fix SSA in various ways, such as by adding or extending phi functions in and after the block we resume control at.

LLVM requires us to perform various book keeping operations to properly create the continuation function. For example, we must maintain several mappings of LLVM Value types between the input, output, and temporary functions. These provide a logical mapping between program locations and values as we transform the original function into the continuation function.

We must also remove the OSR condition from the original function; in the replacement function, we assert that the condition has already been fulfilled. Our strategy for removal of OSR conditions is to simply make the conditional branch in the OSR condition an unconditional branch. The OSR block, where the OSR function is called or generated, can then be removed and will never be reached. Later we can perform liveness analysis to remove any instructions that were added simply for evaluating the OSR condition; the removal of the unconditional block will render them dead.

5 Results

Performance tests were run on an Ubuntu 15.10 machine with 12gigs of RAM and an Intel Intel(R) Core(TM) i7-5600U CPU. Each test was run once (to get these numbers), but the tests were run multiple times to check the sanity of the values. The total time values represent the total time taken to run an initial set of passes, add the module to MCJIT, compile the module with MCJIT, and call `main`. The execution time values represent only the time taken to run `main` (see the `osr_test_tool` for details). The overhead numbers represent the amount of time used by the initial passes and the initial compile with MCJIT.

Test	osr execution time	osr total time	w/o osr exec time	w/o osr total time
live_fp	20099	23120	5656814	5659438
isord	8949	11197	7307	9647
isord_ret1	4584	6746	6062	8970
malloc_test	1022	2555	6062	8970
twoloop	6089720	6092495	11391745	11394680
tree search	10846811	10851694	9415287	9421799
sort	222927	229209	4297	9203

Figure 3: Performace Values

The live_fp test results stand out as an exception. A fragment of the code for this test is given in Figure 6a. The code generated by the OSR process at runtime (for `f1`) is given in Figure 6b. As can be seen from the code fragments, inlining the `f1` allows the compiler to perform very aggressive loop optimizations.

Test Name	MCJIT overhead	OSR Overhead	Speedup Time	Execution Speedup
live_fp	0.05%	13.07%	244.79	281.45
isord	24.26%	20.08%	0.86	0.82
isord_ret1	32.42%	32.05%	1.33	1.32
malloc_test	32.42%	60.00%	3.51	5.93
twoloop	0.03%	0.05%	1.87	1.87
tree search	0.07%	0.04%	0.87	0.87
sort	53.31%	2.74%	0.04	0.02

Figure 4: Derived Values

In the other cases, we do not see the same kinds of performance gains; but, a few of them do show modest improvements in performance. Usually, these performance gains are only achieved when the body of the function being inlined performs a considerable amount work or exposes considerable opportunities for additional optimization (isord_ret1, malloc_test, and twoloop). When this is not the case, we tend to see slight slow downs in the OSR code (isord, tree search).

The search benchmark is a possible implementation of `qsort` from the standard library. This code does not perform well because the osr condition is triggered many, many times (the function is recursive), and the overhead for recursively triggering the osr continuation generation is very high.

6 Limitations

This approach has a few limitations, most of which stem from the inability of the LLVM IR to adequately express high level properties of the language it represents. For example, it is difficult to differentiate between a virtual function call and some other load followed by a call, looking only at LLVM IR. This limitation makes some optimizations more difficult to perform without some way to communicated high level language properties to the llvm optimizer. We also found that some languages (such as the ravi lua implementation[10]) never trigger an OSR condition, even with our OSR pass enabled. This is because the language does not call function pointers from a table directly. Instead, it calls some other language function directly, with a function identifier, which then calls the correct function (Figure 5). This means that a function pointer is never live in the body of the loop, so the OSR library will not be able to optimize. If we are to have any hope of optimization in cases like this, the optimizer needs a way to access properties of the higher level language.

```
call void @luaV_gettable(%struct.lua_State* %L, %struct.TValue* %102, %struct
    .TValue* %100, %struct.TValue* %99)
```

Figure 5: Example of a ravi lua function call

```
int f1(size_t a) {
  int f = 0;

  for (size_t i = 0; i < ITERS; i++) {
    f += a;
  }

  return f;
}

int func(int flag) {
  ft f = NULL;

  switch(flag) {
    case 1: f = f1; break;
    case 2: f = f2; break;
    default: f = f3; break;
  }

  int sum = 0;
  for (int i = 0; i < 1000000; i++) {
    sum += f(i);
  }

  return sum;
}
```

(a) live_fp fragment

```
define i32 @osrcont_func(i32 (i64)* %switch.select2_osr,
                         i32 %i.0_osr,
                         i32 %sum.0_osr,
                         i64 %loop_counter_osr) #0 {
osr_entry:
  %0 = icmp slt i32 %i.0_osr, 1000000
  br i1 %0, label %.preheader.preheader, label %loop.cont._crit_edge

.preheader.preheader:
  %1 = mul i32 %i.0_osr, 500
  %2 = sub i32 999999, %i.0_osr
  %3 = add i32 %1, 500
  %4 = mul i32 %2, %3
  %5 = add i32 %4, %sum.0_osr
  %6 = zext i32 %2 to i33
  %7 = sub i32 999998, %i.0_osr
  %8 = zext i32 %7 to i33
  %9 = mul i33 %6, %8
  %10 = lshr i33 %9, 1
  %11 = trunc i33 %10 to i32
  %12 = mul i32 %11, 500
  %13 = add i32 %5, %1
  %14 = add i32 %13, %12
  br label %loop.cont._crit_edge

loop.cont._crit_edge:
  %sum.0_fix.lcssa = phi i32 [ %sum.0_osr, %osr_entry ],
                             [ %14, %.preheader.preheader ]
  ret i32 %sum.0_fix.lcssa
}
```

(b) optimized live_fp llvm ir fragment

7 Appendix

7.1 Deliverables

The code is compiled and ready to go in `/class/cs526/zmick2/llvm` on the EWS machines. The code has been built in release mode into the directory `build/`

The OSR Pass is implemented in the following files:

- `include/llvm/Transforms/OSR/MCJITWrapper.h`
- `include/llvm/Transforms/OSR/OsrPass.h`
- `lib/Transforms/OSR/MCJITWrapper.cpp`
- `lib/Transforms/OSR/OsrPass.cpp`
- `tools/osr_test_tool/osr_test_tool.cpp`

We have also used portions of the code from tinyvm[9]:

- `include/llvm/Transforms/OSR/StateMap.hpp` - very small amount of code used
- `include/llvm/Transforms/OSR/Liveness.hpp`
- `lib/Transforms/OSR/Liveness.cpp`

There are a variety of testers implemented in the directory `osr_testers` as well as a makefile which will compile the testers to `.ll` files and run the tests. To run all of the tests, run `make`. It may be preferable to run tests individually to investigate their behavior, as a large number of files are created when running `make`.

7.1.1 osr_test_tool

The osr_test_tool program executes `.ll` files using the MCJIT virtual machine in a few different ways.

- A small number of optimization passes, then execution with the MCJIT. Timing are produced by this run.
- A small number of optimization passes, followed by the OSR pass. Then, the code is executed with MCJIT. In this phase, multiple output files are created.
- A small number of optimization passes, followed by the OSR pass. Then, the code is executed with MCJIT. Timings are produced by this run.

The tool produces a number of outputs files.

- The input module instrumented by the OSR pass. File will be named `<module_name>_after_pass.bc`
- The module created by the runtime OSR code generator. These files are created each time OSR fires. Files will be named `<module_name>_osr_module`n`.bc`

This is the tool we've used to test our implementation through the semester. Example usage in Figure 9.

7.2 Extended Example(s)

In this section we will explore two different examples with the “is ordered” function. Consider the implementation of “is ordered” given in Figure 7. We will provide two examples using two different comparison operations. The first example will demonstrate that the optimization performed by the OSR library at runtime can be impressive (Figure 8). The second is somewhat more realistic (Figure 10).

```
int isord(long* v, long n, int (*c)(void* a, void* b)) {
  for (long i=1; i<n; i++)
    if (!c(v+i-1,v + i)) return 0;

  return 1;
}
```

Figure 7: osr_test_tool running the `isord.ll` file

First we prepare an input C program containing the `isord` function in Figure 7 and the code in Figure 8. This program is in the file `isord_ret1.c`.

```
int comp1(void *a, void *b) {
  return 1;
}

int main() {
  size_t max = 1000000;
  long *arr = malloc(sizeof(long) * max);
  memset(arr, 0, max*sizeof(long));

  void* a = malloc(1); free(a);
  printf("running␣isord␣test\n");

  assert(isord(arr, 100, comp1) == 1); // should not do osr
  assert(isord(arr, max, comp1) == 1); // should osr and comp -> ret 1
  printf("passed\n");

  free(arr);
}
```

Figure 8: osr_test_tool running the `isord.ll` file

Then, we compile this code to totally unoptimized LLVM IR using the provided makefile `make isord_ret1.ll`. Next, we run this program with the `osr_test_tool` (Figure 9).

We now investigate the two `.bc` files which are created (excerpts given here). First, take a look at the `isord` function in the module that is initially provided to MCJIT. This is in the file `isord_ret1.ll_after_pass.bc`. We’ve provided a shortened (for space) version of this code in Figure 11. The OSR instrumentation has been added to this function (as can be seen in the Figure). If we take a look at the new `osrcont_isord` function (contained in `isord_ret1.ll_osr_module1.bc`) which is generated at runtime, we see that the generated function has successfully inlined the call to `comp1` and reoptimized the function (Figure 12).

Next, we will try the same thing with more realistic comparator. The code for this is given in Figure 10. Following the same procedure, we can see that the code is optimized at runtime into the code in Figure 13. Obviously, we don't get the same kind of simplification, but we can see that the function call has been inlined. We notice that the order of the arguments in both of the `osrcont_isord` functions is different. This is because we have used a `std::set` of `Value*` objects to store live variables, so the order or these values in the set depends on where the `Value*` happen to be pointing.

```
$ ../build/bin/osr_test_tool isord_ret1.ll
running isord test
passed
without osr time: 7435
without osr total time: 9643
running isord test
passed
running isord test
passed
osr execution time: 2665
osr total time: 4719
$ ls *.bc
isord_ret1.ll_after_pass.bc  isord_ret1.ll_osr_module1.bc
```

Figure 9: osr_test_tool running the `isord.ll` file

```
int comp2(void *a, void *b) {
  long ia = *(long*)a;
  long ib = *(long*)b;

  return ia <= ib;
}

int main() {
  size_t max = 1000000;
  long *arr = malloc(sizeof(long) * max);
  memset(arr, 0, max*sizeof(long));

  printf("running␣isord␣test\n");

  assert(isord(arr, max, comp2) == 1); // should osr
  printf("passed\n");

  free(arr);
}
```

Figure 10: Realistic is ordered comparator and main method

```
; Function Attrs: nounwind uwtable
define i32 @isord(i64* %v, i64 %n, i32 (i8*, i8*)* %c) #0 {
  ;; some code omitted before loop
; <label>:1
  %loop_counter = phi i64 [ %new_loop_counter, %11 ], [ 0, %0 ]
  %i.0 = phi i64 [ 1, %0 ], [ %12, %11 ]
  %osr.cond = icmp sge i64 %loop_counter, 1000
  br i1 %osr.cond, label %osr0, label %loop.cont

loop.cont:
  ;; some code omitted here
  %9 = call i32 %c(i8* %6, i8* %8) #1
  %10 = icmp eq i32 %9, 0
  br i1 %10, label %13, label %11
  ;; some code omitted here

; <label>:13
  %.0 = phi i32 [ 0, %3 ], [ 1, %loop.cont ]
  ret i32 %.0

osr0:
  %14 = alloca [1 x i8*]
  %15 = getelementptr [1 x i8*], [1 x i8*]* %14, i64 0, i64 0
  %16 = bitcast i32 (i8*, i8*)* %c to i8*
  store i8* %16, i8** %15

  ;; call to generator occurs here
  %17 =
   call i8*
    inttoptr (i64 8081136 to
              i8* (i8*, i8*, i8*, i8*, [1 x i8*]*, i1)*)
    (
      i8* inttoptr (i64 36186520 to i8*),
      i8* inttoptr (i64 140736536220912 to i8*),
      i8* inttoptr (i64 36301640 to i8*),
      i8* inttoptr (i64 36433728 to i8*),
      [1 x i8*]* %14, i1 true
    )

  ;; call the generated function
  %18 = bitcast i8* %17 to i32 (i64, i64, i32 (i8*, i8*)*, i64*, i64)*
  %19 = tail call i32 %18(
    i64 %loop_counter,
    i64 %n,
    i32 (i8*, i8*)* %c,
    i64* %v,
    i64 %i.0)

  ret i32 %19
}
```

Figure 11: The OSR instrumented code

```
define i32 @osrcont_isord(i64 %loop_counter_osr,
                          i64 %n_osr,
                          i32 (i8*, i8*)* %c_osr,
                          i64* %v_osr,
                          i64 %i.0_osr) #0
{
osr_entry:
  ret i32 1
}
```

Figure 12: The optimized code (generated at runtime) (some attributes removed for clarity)

```
define i32 @osrcont_isord(i32 (i8*, i8*)* %c_osr,
                          i64 %n_osr,
                          i64 %i.0_osr,
                          i64* %v_osr,
                          i64 %loop_counter_osr) #0
{
osr_entry:
  %0 = icmp slt i64 %i.0_osr, %n_osr
  br i1 %0, label %.lr.ph, label %._crit_edge

loop.cont:
  %1 = icmp slt i64 %8, %n_osr
  br i1 %1, label %.lr.ph, label %._crit_edge

.lr.ph:
  %i.0_fix1 = phi i64 [ %8, %loop.cont ], [ %i.0_osr, %osr_entry ]
  %2 = add nsw i64 %i.0_fix1, -1
  %3 = getelementptr inbounds i64, i64* %v_osr, i64 %2
  %4 = getelementptr inbounds i64, i64* %v_osr, i64 %i.0_fix1
  %5 = load i64, i64* %3, align 8
  %6 = load i64, i64* %4, align 8
  %7 = icmp sgt i64 %5, %6
  %8 = add nsw i64 %i.0_fix1, 1
  br i1 %7, label %._crit_edge, label %loop.cont

._crit_edge:
  %.0 = phi i32 [ 1, %osr_entry ], [ 0, %.lr.ph ], [ 1, %loop.cont ]
  ret i32 %.0
}
```

Figure 13: The optimized code (generated at runtime) (some attributes removed for clarity)

References

[1] Elia, Daniele Cono D; Demetrescu, Camil *Flexible On-Stack Replacement in LLVM.*

[2] Elia, Daniele Cono D *New Techniques for Adaptive Program Optimization* April 2016

[3] Lameed, Nurudeen; Hendren, Laurie *A modular approach to on-stack replacement in LLVM.* VEE 2013

[4] David Detlefs; Ole Ageson *Inlining of Virtual Methods.* ECOOP' 99 — Object-Oriented Programming

[5] Soman, Sunil; Krintz, Chandra *Efficient and General On-Stack Replacement for Aggressive Program Specialization.* VEE 2013

[6] David Detlefs; Ole Ageson *Making Pure ObjectOriented Languages Practical* Conference on ObjectOriented Programming, Systems, Languages, and Applications (OOPSLA), Oct. 1991.

[7] Holzle Urs; Ungar David *A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance* OOPSLA '94 Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications

[8] on-stack replacement in v8
`https://wingolog.org/archives/2011/06/20/on-stack-replacement-in-v8`

[9] tinyvm: A simple VM to play with OSR in LLVM
`https://github.com/dcdelia/tinyvm`

[10] ravi: Lua on llvm
`https://github.com/dibyendumajumdar/ravi`