



McGill University  
School of Computer Science  
Sable Research Group



---

## How to use McOSR to support OSR in LLVM

Sable Technical Report No. sable-2012-04

Nurudeen A. Lameed

21 November 2012

---

[www.sable.mcgill.ca](http://www.sable.mcgill.ca)

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Obtaining McOSR</b>	<b>3</b>
<b>3</b>	<b>Using McOSR</b>	<b>3</b>
3.1	Generating LLVM code for <i>compute</i> . . . . .	4
3.2	Generating LLVM code for <i>bsqrt</i> with no OSR . . . . .	5
3.3	Generating LLVM code for <i>bsqrt</i> with One OSR point . . . . .	6
3.4	Inserting OSR Points . . . . .	7
3.5	Running the OSR Pass . . . . .	7
3.6	Initializing and Finalizing the OSR subsystem . . . . .	7
3.7	Inlining Transformation . . . . .	7
<b>4</b>	<b>More Information</b>	<b>7</b>

## List of Figures

1	Generating LLVM Code for <i>compute</i> . . . . .	4
2	LLVM Code for <i>compute</i> . . . . .	4
3	(a) Generating LLVM Code for <i>bsqrt</i> with no OSR point. . . . .	5
4	(b) Generating LLVM Code for <i>bsqrt</i> with no OSR point. . . . .	6
5	LLVM Code for <i>bsqrt</i> with no OSR point. . . . .	6
6	LLVM Code generation for <i>bsqrt</i> with an OSR point . . . . .	8
7	Continuation of <i>createBSqrtOSR</i> in Figure 6 . . . . .	9
8	LLVM code for <i>bsqrtOSR</i> with an OSR point inserted . . . . .	10
9	The JIT's main function . . . . .	11
10	LLVM Code for <i>bsqrtOSR</i> after running the OSR pass . . . . .	12
11	The transformed code after OSR . . . . .	13

## List of Tables

## 1 Introduction

This document provides the instructions on how to obtain, install, and use McOSR to support on-stack replacement(OSR) in LLVM code. The McOSR project is part of the McLAB [1] project. We used LLVM version 3.0 for our implementation.

## 2 Obtaining McOSR

Instructions on how to download and install McOSR can be found at: <https://sable.mcgill.ca/mclab/mcosr>.

## 3 Using McOSR

All the LLVM code shown in this document work with LLVM version 3.0 and have not been tested with other versions of LLVM. The file *HowToUseOSR.cpp* (located in directory *example* in the source code for McOSR) contains all the code (with useful comments) used in this example. I illustrate the usage of McOSR with an example.

The following MATLAB function computes an approximate square root of the six-digit number, 123456, using a naive implementation of the Babylonian algorithm.

```
1 function r = compute(x, n)
2     x = (x + n/x)/2;
3 end
4
5 function r = bsqrt ()
6     % computes:  $x_{n+1} = 1/2(x_n + n/x_n)$ ;  $\lim_{n \rightarrow \infty} (x_n)$ 
7     n = 123456;
8
9     % approx infinity : long iteration
10    m = 1000000;
11    % x_0;
12    x = 600;
13    for i=1:m
14        x = compute(x, n);
15    end
16 end
```

### 3.1 Generating LLVM code for *compute*

Let us begin with the code generation for function *compute*. Using the LLVM *IRBuilder* library, I show how a user can generate LLVM code for *compute*. The C++ code is straightforward and is shown in Figure 1.

```
1 llvm::Function* createCompute(llvm::Module* M, llvm::LLVMContext& context) {
2 llvm::Constant* C =
3     M->getOrInsertFunction("compute", llvm::Type::getDoubleTy(context),
4                             llvm::Type::getDoubleTy(context),
5                             llvm::Type::getDoubleTy(context),
6                             (llvm::Type*)0);
7 llvm::Function* compute = llvm::cast<llvm::Function>(C);
8 llvm::BasicBlock* entry =
9     llvm::BasicBlock::Create(context, "entry", compute);
10 llvm::IRBuilder<> builder(entry);
11 llvm::Argument* n = compute->arg_begin();
12 n->setName("n");
13 llvm::Argument* x = ++(compute->arg_begin());
14 x->setName("x");
15 llvm::Value* nDIVx = builder.CreateFDiv(n, x);
16 llvm::Value* s = builder.CreateFAdd(x, nDIVx);
17 llvm::Value* half = llvm::ConstantFP::get(builder.getDoubleTy(), 0.5);
18 llvm::Value* result = builder.CreateFMul(s, half);
19     builder.CreateRet(result);
20 llvm::verifyFunction(*compute);
21     return compute;
22 }
```

Figure 1: Generating LLVM Code for *compute*

As shown in Figure 1, function *createCompute* returns the LLVM IR a typical LLVM front-end will generate. The LLVM IR generated by *createCompute* is shown in Figure 2.

```
1 define double @compute(double %n, double %x) {
2 entry :
3     %0 = fdiv double %n, %x
4     %1 = fadd double %x, %0
5     %2 = fmul double %1, 5.000000e-01
6     ret double %2
7 }
```

Figure 2: LLVM Code for *compute*

### 3.2 Generating LLVM code for *bsqrt* with no OSR

In Section 3.1, I describe how *IRBuilder* of LLVM can be used to generate LLVM IR for *compute*, here I show similar steps for generating LLVM code for *bsqrt*. Function *bsqrt* has a loop but I shall show only the steps to generate LLVM code without inserting any OSR point. In Section 3.3, I show how to generate LLVM code for *bsqrt* with an OSR point inserted at the beginning of the loop.

Figure 3 shows the C++ code to generate LLVM IR code for *bsqrt*.

```
1 llvm :: Function* createBSqrt (llvm :: Module* M, llvm :: LLVMContext& context,
2                               llvm :: Function* compute) {
3     // make bsqrt, the caller
4     llvm :: Constant* C =
5         M->getOrInsertFunction("bsqrt",
6                               llvm :: Type::getDoubleTy(context),
7                               (llvm :: Type*)0);
8     llvm :: Function* bsqrt = llvm :: cast <llvm :: Function>(C);
9     llvm :: BasicBlock* entry = llvm :: BasicBlock::Create (context, "entry", bsqrt);
10    llvm :: IRBuilder<> builder(entry);
11    llvm :: Value* n = llvm :: ConstantFP::get ( builder.getDoubleTy (),123456.0);
12    llvm :: Value* m = builder.getInt64 (100000000-1);
13    llvm :: Value* x = llvm :: ConstantFP::get ( builder.getDoubleTy (), 600.0);
14    llvm :: Value* one = builder.getInt64 (1);
15    llvm :: Value* i = builder.getInt64 (0);
16
17    // create a block for the loop
18    llvm :: BasicBlock* bb1 = llvm :: BasicBlock::Create (context, "bb1", bsqrt);
19    llvm :: BasicBlock* bbCont = llvm :: BasicBlock::Create (context, "cont", bsqrt);
20    llvm :: BasicBlock* bbExit = llvm :: BasicBlock::Create (context, "exit", bsqrt);
21    ...
22 }
```

Figure 3: (a) Generating LLVM Code for *bsqrt* with no OSR point.

The LLVM code generated by *createBSqrt* shown in Figure 3 is shown in Figure 5.

```

1  ...
2  builder.SetInsertPoint (bb1);
3  llvm::PHINode* iPHI = builder.CreatePHI(i->getType(), 2);
4  iPHI->addIncoming(i, entry);
5  llvm::PHINode* xPHI = builder.CreatePHI(x->getType(), 2);
6  xPHI->addIncoming(x, entry);
7  // if ii > 10000000 - 1 ... goto bbexit, else cont
8  llvm::Value* exitCond = builder.CreateICmpUGT(iPHI, m);
9  builder.CreateCondBr(exitCond, bbExit, bbCont);
10 builder.SetInsertPoint (bbCont);
11 llvm::Value* ii = builder.CreateAdd(iPHI, one);
12 iPHI->addIncoming(ii, bbCont);
13 llvm::CallInst* xx = builder.CreateCall2 (compute, n, xPHI, "xx");
14 xPHI->addIncoming(xx, bbCont);
15 builder.CreateBr (bb1);
16 builder.SetInsertPoint (entry);
17 builder.CreateBr (bb1);
18 // exit Block
19 builder.SetInsertPoint (bbExit);
20 builder.CreateRet (xPHI);
21 // verify that the generated code is valid LLVM IR code
22 llvm::verifyFunction (*bsqrt);
23 return bsqrt;
24 }

```

Figure 4: (b) Generating LLVM Code for *bsqrt* with no OSR point.

```

1 define double @bsqrt() {
2 entry :
3   br label %bb1
4
5 bb1:                                ; preds = %entry, %cont
6   %0 = phi i64 [ 0, %entry ], [ %3, %cont ]
7   %1 = phi double [ 6.000000e+02, %entry ], [ %xx, %cont ]
8   %2 = icmp ugt i64 %0, 99999999
9   br i1 %2, label %exit, label %cont
10
11 cont:                                ; preds = %bb1
12   %3 = add i64 %0, 1
13   %xx = call double @compute(double 1.234560e+05, double %1)
14   br label %bb1
15
16 exit :                                ; preds = %bb1
17   ret double %1
18 }

```

Figure 5: LLVM Code for *bsqrt* with no OSR point.

### 3.3 Generating LLVM code for *bsqrt* with One OSR point

In the last section, I showed the code to generate typical LLVM code that implements the square root function. In this section, I will show how to use OSR to support a dynamic program transformation. The program

transformation considered is the dynamic inlining of *hot* call sites.

The call instruction in line 13 of Figure 3.3 is a potentially hot call site. I will show how to annotate this call site so that the McJIT's dynamic inliner can consider the call site for inlining at runtime. McJIT's dynamic inliner is based on the LLVM's basic inliner library and is not part of the OSR library.

To use OSR functionality, we must first include two header files: *src/osr.h* and *src/OSRInfoPass.h*. To use McJIT's dynamic inliner, we need to include *src/callsite\_analyzer.h* and *src/McJITInliner.h*.

The key objective of this example is to show a user of McOSR how to use the OSR library to support dynamic program transformation and optimization in LLVM. This section will proceed as follows:

- In Section 3.4, I show how a front-end that generates LLVM IR can insert an OSR point at the beginning of a loop.
- In Section 3.5, I show how to run the OSR pass on a function.
- In Section 3.6, I show how to initialize and finalize the OSR subsystem.
- At the end, I highlight the transformation performed after the OSR event.

As in Figure 3, Figure 6 and Figure 7, show the code to generate LLVM code for function *bsqrt* with an OSR point inserted at the begin of the loop in *bsqrt*.

The LLVM code generated by C++ function *createBSqrtOSR* is shown in Figure 8.

**The JIT's main function** Figure 9 shows the *main* function for the JIT used in this example.

### 3.4 Inserting OSR Points

Lines 3–5 of Figure 7 shows how to insert an OSR point through a call to function *genOSRSignal* of the McOSR. Function *genOSRSignal* requires three arguments: (1) a pointer to the JIT's execution engine, (2) a pointer to the module containing the code, and (3) a pointer to the loop's initialization basic block. Please note that the loop header block must have exactly two predecessors.

The LLVM IR code generated by this call is shown in line 14 of Figure 8.

### 3.5 Running the OSR Pass

In lines 39 to 41 of Figure 9, I show the code to create and run the OSR pass on a function. The LLVM IR code generated after running the OSR pass is shown in Figure 10.

### 3.6 Initializing and Finalizing the OSR subsystem

Line 16 of Figure 9 shows the code to initialize the OSR subsystem. While line 44 shows the code to free up the memory used by the OSR subsystem.

### 3.7 Inlining Transformation

The transformed LLVM IR after the only OSR event has been triggered and completed is shown in Figure 11. Notice that *compute* has been inlined into *bsqrt* (lines 22 – 27). This transformation was done during the OSR event that occurred when *bsqrtOSR* was compiled and its machine code executed.

## 4 More Information

For more information on McOSR, please refer to: [www.sable.mcgill.ca/mclab/mcosr](http://www.sable.mcgill.ca/mclab/mcosr).



```

1 llvm :: Function* createBSqrtOSR(llvm::Module* M, llvm::LLVMContext& context,
2                               llvm :: Function* compute) {
3   // make bsqrt, the caller
4   llvm :: Constant* C =
5     M->getOrInsertFunction("bsqrtOSR",
6                           llvm :: Type::getDoubleTy(context),
7                           (llvm :: Type*)0);
8   llvm :: Function* bsqrt = llvm :: cast<llvm::Function>(C);
9   llvm :: BasicBlock* entry = llvm :: BasicBlock:: Create( context, "entry", bsqrt );
10  llvm :: IRBuilder<> builder(entry);
11  llvm :: Value* n = llvm :: ConstantFP:: get( builder.getDoubleTy (),123456.0 );
12  llvm :: Value* m = builder.getInt64 (100000000-1);
13  llvm :: Value* x = llvm :: ConstantFP:: get( builder.getDoubleTy (), 600.0);
14  llvm :: Value* one = builder.getInt64 (1);
15  llvm :: Value* i = builder.getInt64 (0);
16
17  // create a value for OSR_THRESHOLD: 2
18  llvm :: Value* OSR_THRESHOLD = builder.getInt64(2);
19  // create a block for the loop
20  llvm :: BasicBlock* bb1 = llvm :: BasicBlock:: Create( context, "bb1", bsqrt );
21  builder.CreateBr (bb1);
22
23  // bb1 is the loop header
24  builder.SetInsertPoint (bb1);
25  llvm :: PHINode* iPHI = builder.CreatePHI(i->getType(), 2);
26  llvm :: PHINode* xPHI = builder.CreatePHI(x->getType(), 2);
27  iPHI->addIncoming(i, entry);
28  xPHI->addIncoming(x, entry);
29
30  // let us insert an osr condition here ... and an OSR point
31  // -----
32  llvm :: BasicBlock* osrBB = llvm :: BasicBlock:: Create( context, "osr", bsqrt );
33  // create basic blocks for T/F paths ...
34  llvm :: BasicBlock* fallThru =
35  llvm :: BasicBlock:: Create( context, " o.fallthru ", bsqrt );
36  llvm :: BasicBlock* osrExit =
37  llvm :: BasicBlock:: Create( context, " o.exit ", bsqrt );
38  llvm :: BasicBlock* bbCont = llvm :: BasicBlock:: Create( context, "cont", bsqrt );
39  llvm :: BasicBlock* bbExit = llvm :: BasicBlock:: Create( context, "exit", bsqrt );
40  // generate the OSR conditional inst in the header
41  // to branch into osrBB, if the counter > osr_threshold
42  llvm :: Value* osrCond = builder.CreateICmpUGT(iPHI, OSR_THRESHOLD, "ocond");
43  builder.CreateCondBr(osrCond, osrBB, fallThru );
44  // terminate osrBB
45  builder.SetInsertPoint (osrBB);
46  builder.CreateBr ( osrExit );
47  builder.SetInsertPoint ( fallThru );
48  builder.CreateBr ( osrExit );
49  // ... cont'd

```

Figure 6: LLVM Code generation for *bsqrt* with an OSR point

```

1 // Now insert an OSR point
2 // -----
3 llvm:: CallInst * osrMarker =
4 osr:: Osr:: genOSRSignal(osrBB, llvm::McJITInliner:: inlineAnnotatedCallSites ,
5                          entry /*the loop's entry block ( initialization BB)*/);
6 // end osr point insertion
7 // -----
8 // here we generate the loop termination condition ...
9 // if ii > 100000000 - 1 goto bbexit, else cont
10 builder.SetInsertPoint ( osrExit );
11 llvm:: Value* exitCond = builder.CreateICmpUGT(iPHI, m);
12 builder.CreateCondBr (exitCond, bbExit, bbCont);
13 // continuation block, the loop back-edge
14 builder.SetInsertPoint (bbCont);
15 // increment the counter ...
16 llvm:: Value* ii = builder.CreateAdd (iPHI, one);
17 // compute the updated value of x
18 llvm:: CallInst * xx = builder.CreateCall2 (compute, n, xPHI, "xx");
19 // set the values of the phis from the loop back-edge
20 iPHI->addIncoming(ii, bbCont);
21 xPHI->addIncoming(xx, bbCont);
22 // -----
23 // Annotate the last call site, so that the inliner can consider it
24 // when a corresponding OSR event is triggered at runtime.
25 CallSiteAnalyzer:: annotateCSWithOSRPoint(xx, osrMarker);
26 // -----
27 // complete the code
28 builder.CreateBr (bb1);
29 // exit Block
30 builder.SetInsertPoint (bbExit);
31 builder.CreateRet (xPHI);
32
33 // verify that the generated code is valid LLVM IR code
34 llvm:: verifyFunction (* bsqrt );
35 bsqrt ->dump();
36 return bsqrt ;
37 }

```

Figure 7: Continuation of *createBSqrtOSR* in Figure 6

## References

[1] McLab. <http://www.sable.mcgill.ca/mclab/>.

```

1 define double @bsqrtOSR() {
2 entry :
3   call void @disp()
4   call void @llvm.var.annotation (i8* bitcast (double *) @bsqrtOSR to i8*), i8* null , i8* null , i32 1)
5   br label %bb1
6
7 bb1:                                     ; preds = %cont, %entry
8   %0 = phi i64 [ 0, %entry ], [ %3, %cont ]
9   %1 = phi double [ 6.000000e+02, %entry ], [ %xx, %cont ]
10  %ocond = icmp ugt i64 %0, 2
11  br i1 %ocond, label %osr, label %o.fallthru
12
13 osr:                                     ; preds = %bb1
14  call void @_osrSignalFunc(i8* bitcast (double *) @bsqrtOSR to i8*), i64 1)
15  br label %o.exit
16
17 o.fallthru :                             ; preds = %bb1
18  br label %o.exit
19
20 o.exit :                                 ; preds = %o.fallthru , %osr
21  %2 = icmp ugt i64 %0, 99999999
22  br i1 %2, label %exit, label %cont
23
24 cont:                                    ; preds = %o.exit
25  %3 = add i64 %0, 1
26  %xx = call double @compute(double 1.234560e+05, double %1), !OSR1 !0
27  br label %bb1
28
29 exit :                                   ; preds = %o.exit
30  ret double %1
31 }

```

Figure 8: LLVM code for *bsqrtOSR* with an OSR point inserted

```

1 int main() {
2
3     InitializeNativeTarget ();
4     LLVMContext Context;
5
6     // Create some module to put our function into it.
7     Module *M = new Module("osrtest" , Context);
8
9     // Now we create the JIT.
10    EngineBuilder EB = EngineBuilder(M);
11    EB.setOptLevel(CodeGenOpt::None);
12    ExecutionEngine* EE = EB.create ();
13
14    // Initialize osr stuff here ...
15    // -----
16    osr :: Osr :: init (EE, M);
17    // -----
18
19    // Create Compute
20    Function* compute = createCompute(M, Context);
21
22    // version with no bsqrt with no OSR
23    Function* bsqrt = createBSqrt (M, Context, compute);
24
25    // Compile, execute and print out the result :
26    EE->getPointerToFunction(bsqrt);
27    outs () << "\n\nRunning Bsquare root Without OSR: \n" ;
28    std :: vector<GenericValue> noargs1;
29    GenericValue gv1 = EE->runFunction(bsqrt, noargs1);
30    outs () << "=====\n" ;
31    outs () << "Result: " << gv1.DoubleVal << "\n" ;
32    outs () << "=====\n" ;
33
34    // Now create a new version of bsqrt with an OSR point;
35    Function* bsqrtOSR = createBSqrtOSR(M, Context, compute);
36
37    // Here we create and run the osr pass on bsqrtOSR ...
38    // -----
39    FunctionPassManager fpm(M);
40    fpm.add(osr :: createOSRInfoPass ());
41    fpm.run(*bsqrtOSR);
42    // -----
43    ...
44    osr :: Osr :: releaseMemory();

```

Figure 9: The JIT's main function

```

1 define double @bsqrtOSR() {
2 entry :
3   call void @disp()
4   call void @llvm.var.annotation(i8* bitcast (double ()* @bsqrtOSR to i8*), i8* null, i8* null, i32 1)
5   br label %bb1
6
7 bb1:                                     ; preds = %cont, %entry
8   %0 = phi i64 [ 0, %entry ], [ %3, %cont ]
9   %1 = phi double [ 6.000000e+02, %entry ], [ %xx, %cont ]
10  %ocond = icmp ugt i64 %0, 2
11  br i1 %ocond, label %osr, label %o.fallthru
12
13 osr:                                     ; preds = %bb1
14  call void @_osrSignalFunc(i8* bitcast (double ()* @bsqrtOSR to i8*), i64 1)
15  store i64 %0, i64* @live
16  store double %1, double* @live1
17  store i32 1, i32* @_osrPt
18  call noinline void @_recompile(i8* inttoptr (i64 37044736 to i8*), i32 1)
19  %_osr = call noinline double @bsqrtOSR()
20  ret double %_osr
21
22 o.fallthru :                             ; preds = %bb1
23  br label %o.exit
24
25 o.exit :                                  ; preds = %o.fallthru
26  %2 = icmp ugt i64 %0, 99999999
27  br i1 %2, label %exit, label %cont
28
29 cont:                                    ; preds = %o.exit
30  %3 = add i64 %0, 1
31  %xx = call double @compute(double 1.234560e+05, double %1), !OSR1 !0
32  br label %bb1
33
34 exit :                                    ; preds = %o.exit
35  ret double %1
36 }

```

Figure 10: LLVM Code for *bsqrtOSR* after running the OSR pass

```

1 define double @bsqrtOSR() {
2   prolog.entry :
3     %osrPt = load i32* @__osrPt
4     %cond = icmp eq i32 %osrPt, 0
5     br i1 %cond, label %entry, label %prolog
6
7   entry :                                     ; preds = %prolog.entry
8     call void @disp()
9     br label %bb1.preheader
10
11  bb1.preheader :                             ; preds = %prolog, %entry
12    %ph = phi i64 [ 0, %entry ], [ %_rst_, %prolog ]
13    %ph1 = phi double [ 6.000000e+02, %entry ], [ %_rst_xx, %prolog ]
14    br label %bb1
15
16  bb1 :                                       ; preds = %bb1.preheader, %cont
17    %0 = phi i64 [ %3, %cont ], [ %ph, %bb1.preheader ]
18    %1 = phi double [ %6, %cont ], [ %ph1, %bb1.preheader ]
19    %2 = icmp ugt i64 %0, 99999999
20    br i1 %2, label %exit, label %cont
21
22  cont :                                     ; preds = %bb1
23    %3 = add i64 %0, 1
24    %4 = fdiv double 1.234560e+05, %1
25    %5 = fadd double %1, %4
26    %6 = fmul double %5, 5.000000e-01
27    br label %bb1
28
29  exit :                                     ; preds = %bb1
30    ret double %1
31
32  prolog :                                   ; preds = %prolog.entry
33    store i32 0, i32* @__osrPt
34    %_rst_xx = load double* @live1
35    %_rst_ = load i64* @live
36    br label %bb1.preheader
37 }

```

Figure 11: The transformed code after OSR