Erasm++ User's Guide

Makoto Nishiura
nishiuramakoto@gmail.com

February 1, 2012

Contents

1 Introduction

Erasm++ is a collection of libraries for runtime code generation, instruction decoding, and metaprogramming. At the core of these libraries is MetaPrelude, our Haskell-like lazy metaprogramming library for C++. The emphasis has been on ease-of-use, runtime performance, and general adaptability at the cost of potentially increased compile-time resource usage. Our driving motto is this: *C++ programs can be both much faster and more user-friendly at the same time than the functionally equivalent C programs.* As we will show later in this

manual, they can sometimes be an order of magnitude faster without obscure hand-coded optimizations than some popular C-based libraries.

2 ERASM++

2.1 Overview

ERASM++, the Embedded Runtime Assembler in C++, is an Embedded Domain Specific Language (EDSL) in C++ for runtime code generation. Its main strengths are:

- Close syntactical similarity with the real assembly language.
- Complete compile-time syntax checking.
- Very fast and small code generators.
- When inlined and optimized by recent compilers, the code generator functions compile to just a series of `MOV` instructions that contain no external references. In particular, they do not require any runtime support.
- Near full conformance to the C++03 standard subject to the limitations concerning the platform's data representations (see Section 2.5).

Only the Intel 64/IA-32 architectures are supported at this moment.

2.2 Basic Examples

2.2.1 32-bit case

First, create a file named `demo86.cpp`[1] with the following contents:

```
#include <erasm/x86_addr32_data32.hpp>
#include <iostream>

typedef int (*func_type) (int);

int main(int argc,char**argv)
{
    using namespace std;
    using namespace erasm::x86;
    using namespace erasm::x86::addr32;
    using namespace erasm::x86::addr32::data32;

    byte_t    buf[100];
    code_ptr  p = buf;
    imm32_t   x = 2;
```

[1] This program is included in `PACKAGEROOT/demo` directory.

```
    p += mov(p,eax,dword_ptr[esp+4]);
    p += add(p,eax,x);
    p += ret(p);

    func_type f = (func_type)buf;
    int len = p - buf;

    cout << "result="       << f(argc) << endl
         << "code_length=" << len     << endl;

    return 0;
}
```

Then, run the following commands on your shell:

```
$ g++ -DNDEBUG demo86.cpp -lerasm -o demo86
$ ./demo86
$ ./demo86 a b
```

Provided the stack is executable in your operating system by default, it will print the numbers 3 and 5.

2.2.2 64-bit case

First, create a file named `demo64.cpp`[2] with the following contents:

```
#include <erasm/x64_addr64_data32.hpp>
#include <iostream>

typedef int (*pf_t) (int);

int main(int argc,char**argv)
{
    using namespace std;
    using namespace erasm::x64;
    using namespace erasm::x64::addr64;
    using namespace erasm::x64::addr64::data32;

    byte_t    buf[100];
    code_ptr p = buf;
    imm32_t  x = 2;

    p += mov(p,rax,qword_ptr[rsp+8]);
    p += add(p,rax,x);
    p += ret(p);
```

[2]This program is included in `PACKAGEROOT/demo` directory.

```
    pf_t f = (pf_t)buf;
    int len = p - buf;

    cout  << "result="       << f(argc) << endl
          << "code_length=" << len     << endl;

    return 0;
}
```

Then, run the following commands on your shell:

```
$ g++ -DNDEBUG demo64.cpp -lerasm -o demo64
$ ./demo64
$ ./demo64 a b
```

Provided the stack is executable in your operating system by default, it will print the numbers 3 and 5.

The namespace `erasm::x86::addr32::data32` contains the actual x86 instruction definitions whose default address and operand sizes are both 32-bit. Similarly, the namespace `erasm::x64::addr64::data32` should be used for IA-64 instructions whose default address and operand sizes are respectively 64-bit and 32-bit. Currently other default address or operand size modes are not supported.

2.3 Syntax

In general, the prototype of an ERASM++ instruction is in one of the following forms:

```
int MNEMONIC(code_ptr p);

int MNEMONIC(code_ptr p,
             OP_TYPE op1);

int MNEMONIC(code_ptr p,
             OP_TYPE op1,
             OP_TYPE op2);

int MNEMONIC(code_ptr p,
             OP_TYPE op1,
             OP_TYPE op2,
             OP_TYPE op3);
```

where `code_ptr` is an alias for the type of pointer to 8-bit unsigned integers on IA-64/IA-32 architectures.

Listing 1: Examples of ERASM++ instructions.

```
p += adc  (p,word_ptr  [edx + esi * _2],ax);
p += and_ (p,dword_ptr [rip + 0x1000],r12d);
p += jmp  (p, 0x10);
p += mov  (p,byte_ptr  [eax*_5],dl);
p += rep_movs (p,dword_ptr.es [di],dword_ptr.gs [si]);
```

Listing 2: The corresponding MASM instructions.

```
adc  WORD PTR  [edx + esi * 2],ax
and DWORD PTR [rip + 0x1000],r12d
jmp  0x10
mov  BYTE PTR  [eax*5],dl
rep movs DWORD PTR es:[di],DWORD PTR gs:[si]
```

The operational meaning of each instruction is always the same: it encodes the instruction at the location specified by the first argument, and returns the number of bytes encoded. If the target address is writable, it is guaranteed that either it succeeds to encode the instruction, or the instruction is syntactically ill-formed and the compilation fails. If the destination address is not writable, the behavior is undefined. In particular, it never throws a (C++) exception.

MNEMONIC is generally one of the instruction mnemonics as defined in the Intel Instruction Set Reference [5], with each letter converted to lower-case. However, there are some exceptions:

- If the mnemonic is reserved as a keyword in C++ , an underscore is appended, as in `int_`,`xor_`, etc.
- If the instruction requires a mandatory prefix such as REP, REPZ etc., then the prefix and the mnemonic of the instruction are joined by an underscore, as in `repe_scas`.
- Some instructions are ambiguous and cannot be fully specified solely by the combination of the mnemonic and operand types. In such a case, we resort to rather ad-hoc case-by-case disambiguation schemes such as:

  ```
  p +=       rep_movs(p,byte_ptr[rdi],byte_ptr[rsi]);
  p += rexw_rep_movs(p,byte_ptr[rdi],byte_ptr[rsi]);
  ```

  Here, `rep_movs` uses `ecx` as the count register, while `rexw_rep_movs` uses `rcx`. We are searching for a less ad-hoc disambiguation method, but we would also like the form of these instructions to be easily guessable from the definitions in Intel Instruction Set Reference [5].

OP_TYPE is the type of an instruction operand which is one of the following objects:

Registers. `dh`,`ax`,`edx`,`xmm0`,`st3`,`rax`,`r8`,`r9b`,`sil`, etc.

Memory references. Examples:

- `byte_ptr [esp + ebp * _2 + 4]`
- `dword_ptr [ebx * _8 + _4]`
- `xmmword_ptr . gs[bx+di]`

Immediates. `(int32_t)0`,`(int8_t)1`,`_1`,`_2`, etc.

The static constants `_0`-`_9` can be used as a shorthand for small 8-bit integers. By nature, assembly instructions are quite picky about the exact operand types, and it is very often the case that either using static constants or casting explicitly is required to avoid the ambiguity.

Note that the scale coefficient of the index register (e.g. `_2` in the expression `[edx + _2 * ecx]`) must be specified with the static constants `_1` - `_9`. This inconvenience results from the need to compute the necessary data statically, and will not change.

Please see Intel Instruction Set Reference [5] for the details of instruction definitions.

For the examples of syntactically valid instructions, see

- `src/erasm/x64_assembler_manual_test.cpp`
- `src/erasm/x64_assembler_auto_test.cpp`
- `src/erasm/x86_assembler_manual_test.cpp`
- `src/erasm/x86_assembler_auto_test.cpp`

For the examples of ill-formed instructions, see

- `src/erasm/x64_assembler_illegal_code.cpp`
- `src/erasm/x86_assembler_illegal_code.cpp`

2.4 Performance

If all the required functions are inlined(see below), ERASM++ instructions will be compiled to a series of simple memory copy instructions with no external references(except possibly the reference to the execution stack).

Let us give an example. On a 32-bit Pentium machine, g++ 4.6.1 compiles (with the option -O3 to ensure inlining) the following code:

```
inline int adc (code_ptr p,
                const ByteReg86 & op1,
                const BytePtr86 & op2)
{
  // details omitted
}

extern "C"  int code_test(code_ptr p)
{ return adc(p ,cl,byte_ptr [eax+ebp*_4+12345678]);  }
```

to:

```
_code_test:
sub     esp, 32
mov     eax, DWORD PTR [esp+36]
mov     BYTE PTR [esp+5], -88
mov     DWORD PTR [esp+6], 12345678
mov     BYTE PTR [esp+4], -116
mov     BYTE PTR [eax], 103
mov     BYTE PTR [eax+1], 18
mov     edx, DWORD PTR [esp+4]
mov     DWORD PTR [eax+2], edx
mov     edx, DWORD PTR [esp+8]
mov     WORD PTR [eax+6], dx
mov     eax, 8
add     esp, 32
ret
```

Although the above code may not be the fastest possible, we observe the following very nice properties:

No branches. Conditional or not, branches always consume precious processor resources [4], so their absence is always beneficial for performance reasons. Furthermore, in kernel spaces, there is a greater risk that the target address is not accessible at all (e.g. the corresponding section may have been swapped out and the code runs in a high priority mode.)

No external references. This is also a very good property for the similar reasons. See [4] for the details.

Although there are strong arguments against introducing C++ code into the operating system kernel spaces [1][6], it should be clear that ERASM++ *per se* does not possess any such weakness that makes it hard to adopt C++ code into the kernel space.

Currently the main ERASM++ encoder functions are all compiled in a seperate module for the sake of compiletime performance.

See `src/test_asm_performance.cpp` for how to inline the required functions.

2.5 Limitations

Compile-time Performance By far the biggest limitation of ERASM++ is its compile-time resource usage. Because of the heavy use of template metaprogramming techniques, the exact compile-time behaviour greatly varies from one compiler version to another.

For example, while g++-4.5.4 compiles a source file which consists of about 2500 instructions just fine with memory usage ∼500MB, g++-4.6.1 consumes

all available user-space (typically up to 2GB in 32-bit systems) and eventually gives up.

Error Messages Another disadvantage of using ERASM++ is its often cryptic error messages. This is inherently an implementation-dependant problem, and is impossible to solve generally. Current implementation only makes some effort to increase readability for g++.

Here's an example (manually formatted) error message from g++. Given the code

```
cvtpd2pi(p,mm1,xmmword_ptr[rax+ rsp*_2]);
```

g++-4.6.1 prints the following message:

```
./erasm/meta_prelude_core.hpp:829:128:
error: no type named 'result' in
'struct erasm::prelude::eval<
erasm::prelude::if_<
 erasm::prelude::error<
  erasm::x64::
  Error_ESP_and_RSP_cannot_be_used_as_the_index_register,
   erasm::prelude::Term<erasm::prelude::Int<4>,
    erasm::prelude::times<erasm::prelude::Int<2>,
                          erasm::prelude::Int<1> > >,
   erasm::prelude::Term<erasm::prelude::Int<0>,
                        erasm::prelude::Int<1> >,
   void, void>,
 erasm::x64::PtrRex<erasm::x64::XmmWord>,
 erasm::x64::PtrNoRex<erasm::x64::XmmWord> > >'
...
```

Note the underlined identifier above, which correctly spots the problem.

Unfortunately, Microsoft's `cl.exe` tries to outsmart our effort and emits more obscure error messages.

Labels We have not implemented labels yet. It is necessary to maintain code positions manually if back-patching or relocations are required.

Standard Conformance ERASM++ declares static constants `_1`,`_2`, etc. in one of its namespaces. This is potentially problematic as the standard specifies that any identifier that begins with an underscore is reserved for the implementation in the global namespace.

Finally, ERASM++ cannot be fully standard conformant because it must be dependent on the platform-specific data representations. For example, the Intel 64/IA-32 related code relies on the availability of 8-, 16-, 32-, and 64-bit integers, the availability of `#pragma pack` compiler directive, and the little-endianess of the platform. However, if the compiler does support the platform,

there should be no problem compiling ERASM++ as long as the compiler is reasonably standard conformant.

3 GenericDsm

3.1 Overview

GenericDsm is a fast and generic instruction decoder library that supports pattern-match like interface which resembles that of modern functional languages such as Haskell. Features include:

Genericity. Since the decoder function is a template function parametrized by the user-supplied set of actions, it can adapt to any kind of needs without sacrificing performance.

Speed. Genericity allows specializing to one's custom needs. In some cases, this leads to an instruction decoder which is more than an order of magnitude faster than non-generic instruction decoder libraries such as libopcodes (a library which is ubiquitous in GNU/Linux systems).

Pattern-matching against instructions. Not only does it lead to cleaner code, it also contributes to reducing common overheads in many existing disassembler libraries. For example, it allows using static compile-time predicates on a set of instructions and operands, avoiding the common runtime tests in C-based libraries.

No dependence on runtime system. Just like ERASM++, the main decoder function in GenericDsm itself does not depend on other libraries including STL or even the C runtime libraries, except some Boost's header-only libraries. Furthermore, it does not throw exceptions, rely on vtables, or allocate memory outside the stack. This allows GenericDsm to be used in most demanding circumstances.

Reentrant. Since the decoder function itself only uses local stack-allocated variables, it is fully reentrant and thread-safe as long as the user-supplied actions are. (Assuming the decoded instructions are constant, of course.)

Only Intel 64/IA-32 architectures are supported. Also, there is a considerable compile-time overhead using GenericDsm as is common in all C++ programs with heavy use of metaprogramming and inlining.

3.2 Basic Examples

3.2.1 32-bit case

First, create a file named `dsm86.cpp`[3] with the following contents:

[3]Included in PACKAGEROOT/demo directory.

```
#include <erasm/dsm_x86_util.hpp>
#include <erasm/dsm_x86.hpp>
#include <erasm/faststream.hpp>

using namespace std;
using namespace erasm::x86;
using namespace erasm::x86::addr32::data32;

struct MyDsm : public SimpleDsm<>
{
   MyDsm(const_code_ptr start,ostream& os = erasm::cout)
      : SimpleDsm(start,os)
      {}

   using SimpleDsm::action;

   action_result_type
   action(const Ret& insn)
      {
         os() << "Matched:" << Ret::mnemonic << endl;
         return finish(insn);
      }

   template<class Op>
   action_result_type
   action(const Push& insn,const Op& op)
      {
         os() << "Matched:" << Push::mnemonic
              << "␣" << op << endl;
         return next(insn);
      }

};

int f(int x)
{
   return x+1;
}

int main(int argc,char**argv)
{
   const_code_ptr p = (const_code_ptr) f;
   MyDsm   mydsm(p);

   const_code_ptr end = decode(p,mydsm);
```

```
    if (mydsm.failed()) {
        erasm::cout << "Disassembly failed at:"
                    << hex << (void*)end << endl;
    } else {
        erasm::cout << "Disassembly ended at:"
                    << hex << (void*)end << endl;
    }
    return 0;
}
```

Then, run the following commands on your shell:

```
$ g++ -DNDEBUG -O3 dsm86.cpp -lerasm_dsm -o dsm86
$ ./dsm86
```

On a 32-bit system, the output will look like the following:

```
Matched:push ebp
89e5                08048491   2 mov ebp,esp
8b4508              08048493   3 mov eax,DWORD PTR[ebp+0x8]
83c001              08048496   3 add eax,0x1
5d                  08048499   1 pop ebp
Matched:ret
Disassembly ended at:0x804849b
```

3.2.2 64-bit case

First, create a file named `dsm64.cpp` with the following contents:

```
#include <erasm/dsm_x64_util.hpp>
#include <erasm/dsm_x64.hpp>
#include <erasm/faststream.hpp>

using namespace std;
using namespace erasm::x64;
using namespace erasm::x64::addr64::data32;

struct MyDsm : public SimpleDsm<>
{
    MyDsm(const_code_ptr start,ostream& os = erasm::cout)
        : SimpleDsm(start,os)
        {}

    using SimpleDsm::action;

    action_result_type
```

```
    action(const Ret& insn)
       {
          os() << "Matched:" << Ret::mnemonic << endl;
          return finish(insn);
       }

    template<class Op>
    action_result_type
    action(const Push& insn,const Op& op)
       {
          os() << "Matched:" << Push::mnemonic
               << " " << op << endl;
          return next(insn);
       }

};

int f(int x)
{
    return x+1;
}

int main(int argc,char**argv)
{
    const_code_ptr p = (const_code_ptr) f;
    MyDsm    mydsm(p);

    const_code_ptr end = decode(p,mydsm);

    if (mydsm.failed()) {
       erasm::cout << "Disassembly failed at:"
                   << hex << (void*)end << endl;
    } else {
       erasm::cout << "Disassembly ended at:"
                   << hex << (void*)end << endl;
    }
    return 0;
}
```

Then, run the following commands on your shell:

```
$ g++ -DNDEBUG -O3 dsm64.cpp -lerasm_dsm -o dsm64
$ ./dsm64
```

On a 64-bit system, the output will look like the following:

```
Matched:push rbp
```

```
89e5                    08048491   2 mov ebp,esp
8b4508                  08048493   3 mov eax,DWORD PTR[rbp+0x8]
83c001                  08048496   3 add eax,0x1
5d                      08048499   1 pop rbp
Matched:ret
Disassembly ended at:0x804849b
```

Note that the directive `using SimpleDsm::action` in those two examples is mandatory unless your compiler employs a non-standard name resolution algorithm.

As we will discuss shortly, the above apparently quite simple examples already give *very fast* disassemblers. Indeed, they are more than four times as fast as libopcodes based disassemblers on one of our systems (Windows 7 32-bit, Pentium Dual E2220 2.40GHz) *including IO*.

3.3 Overview of the API

In this section, we outline the current library interface. Note that the current implementation is still in its alpha stage, and the interface *will* change as we receive feedbacks from users/testers.

Here is the prototypes of the two most important functions in the library:

```
namespace erasm { namespace x64 {
namespace addr64 { namespace data32 {

    template<class Action>
    const_code_ptr
    decode(const_code_ptr decode_start,
           Action & action);

}}}}

namespace erasm { namespace x86 {
namespace addr32 { namespace data32 {

    template<class Action>
    const_code_ptr
    decode(const_code_ptr decode_start,
           Action & action);

}}}}
```

The functions take two arguments, the first of which denotes the address at which the disassembly starts, and the second what action it should take for each decoded instruction. It returns the address the decoding has ended.

To describe the condition under which the class **Action** is to be a valid argument type, recall the prototype of an n-ary ($n = 0, 1, 2, 3$) ERASM++

instruction:

```
int mnemonic(code_ptr_type p,
             OP_TYPE_0 op_0,
             .. ,
             OP_TYPE_{n-1} op_{n-1});
```

For each *mnemonic*, we define the corresponding class *Mnemonic*, a class whose name is the same as *mnemonic* except the first letter is upper-case. The class *Mnemonic* is a non-virtual subclass of `InstructionData`, which we will explain later.

Now, for an object `act` of type `Action` to be a valid argument for `decode`, the following expression must be valid for any object `insn` of type *Mnemonic*:

```
std::pair<const_code_ptr,ACTION_TYPE> ret
    = act.action(insn,op_0,..,op_{n-1});
```

Here, `ret.second` is either:

`ACTION_FINISH` , to instruct the function to return (with value `ret.first`), or

`ACTION_CONTINUE` , to instruct the function to continue decoding at the address specified by `ret.first`.

Furthermore, the following expression must be valid:

```
const_code_ptr ret = act.error(decode_data);
```

where `decode_data` is an object of class `InstructionData` and contains the location of decode error.

The following prototype illustrates the sufficient condition to be the type of a valid argument for `decode`.

```
typedef std::pair<const_code_ptr,ACTION_TYPE>
        action_result_type;

struct Action
{
   template<class Insn>
   action_result_type
   action(const Insn& insn);

   template<class Insn,class Op1>
   action_result_type
   action(const Insn& insn,
          const Op1& op1);

   template<class Insn,class Op1,class Op2>
   action_result_type
```

```
    action(const Insn& insn,
           const Op1& op1,
           const Op2& op2);

    template<class Insn, class Op1, class Op2, class Op3>
    action_result_type
    action(const Insn& insn,
           const Op1& op1,
           const Op2& op2,
           const Op3& op3);

    const_code_ptr_type
    error(const InstructionData& params);

};
```

For example, the following code defines a simple instruction counter that only considers `ret` and `jmp` instructions:

```
#include <erasm/dsm_x86_util.hpp>
#include <erasm/dsm_x86.hpp>

using namespace erasm::x86;
using namespace erasm::x86::addr32::data32;

struct MyCounter : public InstructionCounter
{
    InstructionCounter(const_code_ptr start)
       : InstructionCounter(start)
       {}

    // Let the base class handle the default cases
    using InstructionCounter::action;
    //
    using InstructionCounter::counter_;

    action_result_type
    action(const Ret& insn)
       {
          counter_++;
          return finish(insn);
       }

    template<class Op>
    action_result_type
    action(const Jmp& insn, const Op& op)
```

```
    {
        counter_++;
        return finish(insn);
    }
};
```

Note how both direct and indirect `jmp` instructions are matched by the following member declaration:

```
template <class Op>
action_result_type action(const Jmp& insn, const Op& op);
```

If the user wants to treat the 8-bit direct `jmp` instructions specially, it is just as simple as adding the following member definition:

```
action_result_type action(const Jmp& insn, rel8_t op1);
```

Each instruction class is derived from `InstructionData`, which is defined in `erasm/x64_instruction_definition_common.hpp`. (Caution: the interface of this class *will* change.) Also, for each instruction class a static member `mnemonic` of type `const char * const` is defined and holds the obvious value.

Finally, for any operand object `x` and for any ostream object `os` , the expression `os << x` is guaranteed to be valid and prints appropriate information.

3.4 Performance

We have compared the performance of GenericDsm and several free decoder libraries for various tasks. The tasks were conducted on the following two machines:

Machine 1 Pentium Dual E2220 2.40GHz, Windows 7 32-bit.

Machine 2 Celeron M 1.3GHz, Linux 32-bit.

The results are summarized in Table 1 and Table 2.

Libraries The libraries used for the comparison are the following:

GenericDsm (GD) version 0.10, compiled with:

- Mingw gcc 4.6.1 on Machine 1, option -O3
- Debian gcc 4.6.1-4 on Machine 2, option -O3

DynamoRIO [2] (DR). The following binary distributions were used:

- DynamoRIO-Windows-3.1.0-4 on Machine 1,
- DynamoRIO-Linux-3.1.0-4 on Machine 2.

udis86 [8] version 1.7, compiled with:

- Mingw gcc 4.6.1 on Machine 1, option -O3
- Debian gcc 4.6.1-4 on Machine 2, option -O3

libopcodes [7]. The following binary distributions were used:

- Mingw GNU binutils version 2.21.53.20110804 on Machine 1
- Debian GNU binutils version 2.21.52.20110606 on Machine 2

Tasks All the tasks were conducted ten times on the same contiguous block of instructions. This block consisted of the equal numbers of all supported instructions in IA-32 mode, totaling 2150000 instructions that consumed about 10M bytes of memory.

The tasks conducted are the following:

- Counting the number of instructions in the block.
- Counting the number of CTIs, the Control Transfer Instructions, which include JMP, CALL, Jcc, LOOPcc and other instructions that affect the instruction pointer register.
- Printing disassembly of the same block in Format A (see Table 3).
- Printing disassembly of the same block in Format B (see Table 3).
- Printing disassembly of the same block in Format C (see Table 3).

In order to give a rough estimate of the significance of IO overhead in these tasks, we have compared the performance of disassembly in three different formats. This was necessary because the formatted IO is hard-coded in the decoder functions of some of the libraries compared.

Conclusions

- DynamoRIO uses adaptive level-of-detail instruction representation [3] and provides specialized decoder functions for each representation. In particular, it provides a very fast instruction decoder `decode_next_pc` which calculates only the next instruction address. This was the only function that outperformed GenericDsm in this comparison.
- In all other tasks, GenericDsm was 2-70 times faster than other libraries.
- The three libraries we have compared against use `printf` or its variants for formatted output, whereas GenericDSM uses a custom ostream object `erasm::cout` which turned out be slightly faster in some cases than printf or its variants.

Task	GD	DR	udis86	libopcodes
Count Instructions	0.19	0.11	5.00	10.55
Count CTIs	0.19	1.34	N/A	N/A
Disassembly (Format A)	2.28	N/A	6.58	N/A
Disassembly (Format B)	2.89	14.01	7.08	N/A
Disassembly (Format C)	4.10	N/A	8.75	18.55

Table 1: Results on Machine 1 (Pentium Dual E2220 2.40GHz, Windows 7 32-bit). The numbers indicate the average elapsed times in seconds.

Task	GD	DR	udis86	libopcodes
Count instructions	0.33	0.21	9.34	4.99
Count CTIs	0.31	7.86	N/A	N/A
Disassembly (Format A)	3.99	N/A	10.56	N/A
Disassembly (Format B)	4.74	29.94	11.47	N/A
Disassembly (Format C)	6.83	N/A	13.79	14.44

Table 2: Results on Machine 2 (Celeron M 1.3GHz, Linux 32-bit). The numbers indicate the average elapsed times in seconds.

	Opcode	Address	Length	Instruction
Format A:				adc al,0x12
Format B:		08159c18		adc al,0x12
Format C:	1412	08159c18	2	adc al,0x12

Table 3: Disassembly formats used for the comparison.

- In libopcodes and udis86, the formatted IO code is tightly coupled in the decoder function and there seems to be no easy way to measure overheads related to formatted IO separately. This means that even mere instruction counting involves some overhead related to formatted IO in these libraries. In contrast, GenericDsm could easily adapt to all the tasks with no unnecessary overheads.

3.5 Limitations

- Slow compilation.
- Can cause code bloat if instantiated to too many actions.
- No support for symbol resolution.
- No support for detecting the instruction's effects on the `eflags` register.
- Only relatively recent, reasonably standard conformant compilers can be used.
- Compilation errors can cause long and unfriendly error messages.
- The library interface is still unstable.

4 License

References

[1] Jeremy Andrews. Linux: C++ In The Kernel? `http://kerneltrap.org/node/2067`, 2004.

[2] Derek Bruening and Qin Zhao. DynamoRIO. `http://dynamorio.org/home.html`.

[3] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Department of Electrical Engineering and Computer Science,Massachusetts Institute of Technology, 2004.

[4] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2011.

[5] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z, 2011.

[6] Microsoft Corporation. C++ for Kernel Mode Drivers: Pros and Cons. http://msdn.microsoft.com/en-us/windows/hardware/gg487420, 2007.

[7] Free Software Foundation. GNU Binutils. http://www.gnu.org/software/binutils/.

[8] Vivek Thampi. udis86. http://udis86.sourceforge.net/.