Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1507–1514

Reconstruction of Instruction Idioms in a Retargetable Decompiler

Jakub Křoustek, Fridolín Pokorný
Faculty of Information Technology,
Brno University of Technology,
Božetěchova 1/2, 612 66 Brno, Czech Republic
Email: ikroustek@fit.vutbr.cz, xpokor32@stud.fit.vutbr.cz

***Abstract*—Machine-code decompilation is a reverse-engineering discipline focused on reverse compilation. It performs an application recovery from binary executable files back into the high level language (HLL) representation. One of its critical tasks is to produce an accurate and well-readable code. However, this is a challenging task since the executable code may be produced by one of the modern compilers that use advanced optimizations. One type of such an optimization is usage of so-called instruction idioms. These idioms are used to produce faster or even smaller executable files. On the other hand, decompilation of instruction idioms without any advanced analysis produces almost unreadable HLL code that may confuse the user of a decompiler. In this paper, we present a method of instruction-idioms detection and reconstruction back into a readable form with the same meaning. This approach is adapted in an existing retargetable decompiler developed within the Lissom project. The implementation has been tested on several modern compilers and target architectures. According to our experimental results, the proposed solution is highly accurate on the RISC (Reduced Instruction Set Computer) processor families, but it should be further improved on the CISC (Complex Instruction Set Computer) architectures.**

***Keywords*—compiler optimizations, reverse engineering, decompiler, Lissom, instruction idioms, bit twiddling hacks**

I. Introduction

REVERSE engineering is a process analyzing existing objects to discover knowledge about their functionality. Within the computer and information security, reverse engineering is often used for analysis of binary executable files. This is useful for vulnerability detection, malware analysis, compiler verification, code migration, etc. In present, this analysis is commonly done by using low-level tools such as disassemblers and dumpers.

Machine-code decompilers (i.e. reverse compilers) are more effective but not so wide-spread. Their task is to recover HLL representation (e.g. C source code) from executable files. In contrast to compilation, the process of decompilation is much more difficult because the decompiler must deal with incomplete information on its input (e.g. information used by the compiler but not stored within the executable file). Furthermore, the input machine code is often heavily optimized by one of the modern compilers (e.g. GCC, LLVM, MSVC); this makes decompilation even more challenging.

This work was supported by the BUT grant FEKT/FIT-J-13-2000 Validation of Executable Code for Industrial Automation Devices using Decompilation and BUT FIT grant FIT-S-11-2.

Code de-optimization is one of the necessary transformations used within decompilers. Its task is to properly detect the used optimization and to recover the original HLL code representation from the hard-to-read machine code. One example of this optimization type is the usage of *instruction idioms* [1]. An instruction idiom is a sequence of machine-code instructions representing a small HLL construction (e.g. arithmetic expression, assignment statement) that is highly-optimized for its execution speed and/or small size.

The instructions in such sequences are assembled together by using Boolean algebra, arbitrary-precision arithmetic, floating-point algebra, bitwise operations, etc. Therefore, the meaning of such sequence is usually hard to understand at the first sight. A notoriously known example is the usage of an exclusive or to clear the register content (i.e. `xor reg, reg`) instead of an instruction assigning zero to this register (i.e. `mov reg, 0`).

The goal of this paper is to present an approach how to deal with instruction-idioms detection and their reconstruction during decompilation. This approach has been successfully adapted within an existing decompiler developed within the Lissom project [2, 3]. Moreover, this decompiler is developed to be retargetable (i.e. independent on a particular target platform, operating system, file format, or a used compiler); therefore, the proposed analysis has to be retargetable too.

This paper is organized as follows. In Section II, we give an introduction to instruction idioms and their usage within compiler optimizations. Then, we briefly describe the retargetable decompiler developed within the Lissom project in Section III. Afterwards, we present our own approach in Section IV. The most common instruction idioms employed in the modern compilers are presented and illustrated in the same section. Section V discusses the related work of instruction idioms reconstruction. Experimental results are given in Section VI. Section VII closes the paper by discussing future research.

II. Instruction Idioms used in Compilers

In present, the modern compilers use dozens of optimization methods for generating fast and small executable files. Different optimizations are used based on the optimization level selected by the user. For example, the GNU GCC compiler supports these optimization levels[1]:

- `O0` – without optimizations;

[1] See http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html for details.

- `O1` – basic level of speed optimizations;
- `O2` – the common level of optimizations (the ones contained in `O1` together with basic function inlining, peephole optimizations, etc.);
- `O3` – the most aggressive level of optimizations;
- `Os` – optimize for size rather than speed.

In the nowadays compilers, the emission of instruction idioms cannot be explicitly turned on by some switch or command line option. Instead, these compilers use selected sets of idioms within different optimization levels. Each set may have different purpose, but multiple sets may share the same (universal) idioms. The main reasons why to use instruction idioms are:

- The most straightforward reason is to exchange slower instructions with the faster ones. These optimizations are commonly used even on the lower optimization levels.
- The floating-point unit (FPU) might be missing, but a programmer still wants to use floating-point numbers and arithmetic. Compilers solve this task via floating-point emulation routines (also known as *software floating point* or *soft-float* in short). Such routines are generated instead of hardware floating-point instructions and they perform the same operation by using available (integer) instructions.
- Compilers often support an optimization-for-size option. This optimization is useful when the target machine is an embedded system with a limited memory size. An executable produced by a compiler should be as small as possible. In this case, the compiler substitutes a sequence of instructions encoded in more bits with a sequence of instructions encoded in less bits in general. This can save some space in instruction cache too.

Another type of optimization classification is to distinguish them based on the target architecture. Some of them depend on a particular target architecture. If a compiler uses platform-specific information about the generated instructions, these instructions can be classified as platform-specific. Otherwise, they are classified as platform-independent.

As an example of platform-independent idiom, we can mention the `div` instruction representing a fixed-point division. The fixed-point division (signed or unsigned) is one of the most expensive instruction in general. Optimizing the division leads to a platform-independent optimization.

On the other hand, clearing the content of the register by using the `xor` instruction (mentioned in the introduction) is a highly platform-specific optimization. Different platforms can use different approaches to clear the register content. As an example, consider the zero register on MIPS (`$zero` or `$0`), which always contains the value of 0. Using this register as a source of zero bits looks like a faster solution than using a `xor` instruction.

Furthermore, different compilers use different instruction idioms to fit their optimization strategies. For example, GNU GCC uses an interesting optimization when making signed comparison of a variable. When a number is represented on 32-bits and bit number 31 is representing the sign, logically shifting the variable right by 31 bits causes to set the zeroth bit equal to the original sign bit. The C programming language classifies 1 as *true* and 0 as a *false*, which is the expected result of the given less-than-zero comparison. This idiom is shown in Fig. 1. The Fig. 1a represents a part of a source code with this construction. The result of its compilation with optimizations enabled is depicted in Fig. 1b. We illustrate the generated code on the C level rather than machine-code level for better readability.

The compiler used the before-mentioned instructions idiom—replacing the comparison by the shift operation. The not-standardized `lshr()` function is used in the output listed in Fig. 1b. The C standard does not specify whether operator ">>" means logical or arithmetical right shift. Compilers deal with it in an implementation-defined manner. Usually, if the left-hand-side number used in the shift operation is signed, arithmetical right shift is used. Analogically, logical right shift is used for unsigned numbers.

III. Lissom Project Retargetable Decompiler

In this section, we briefly describe the concept of an automatically generated retargetable decompiler developed within the Lissom project [2]. This decompiler aims to be independent on any particular target architecture, operating system, object file format, or originally used compiler. The concept of the decompiler is depicted in Fig. 2. Its detailed description can be found in [3]. Currently, the decompiler supports decompilation of MIPS, ARM, and Intel x86 executable files stored in different file formats.

The input binary executable file is preprocessed at first. The preprocessing part tries to detect the used file format, compiler, and (optional) packer, see [4] for details. Afterwards, it unpacks and converts the examined platform-dependent application into an internal uniform Common-Object-File-Format (COFF)-based representation. Currently, we support conversions from UNIX ELF, Windows Portable Executable (WinPE), Apple Mach-O, Symbian E32, and Android DEX file formats. The conversion is done via our plugin-based converter described in [5, 6]. Afterwards, such COFF-file is processed in the decompilation core that consists of three basic parts—a *front-end*, a *middle-end*, and a *back-end*. The last two of them are built on top of the LLVM Compiler Infrastructure [7]. LLVM Intermediate Representation (LLVM IR) [8] is used as an internal code representation of the decompiled applications in all particular decompilation phases.

```
int main(void)
{
  int a, b;

  /* ... */

  b = a < 0;

  /* ... */
}
```

(a) Input.

```
int main(void)
{
  int a, b;

  /* ... */

  b = lshr(a, 31);

  /*... */
}
```

(b) Output (for better readability in C).

Fig. 1: Example of an instruction idiom (C code).

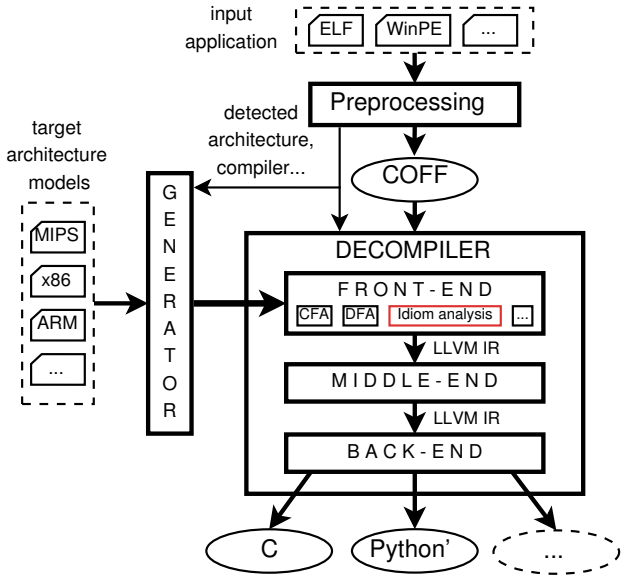

Fig. 2: The concept of the Lissom project retargetable decompiler.

After that, the unified COFF files are processed by the front-end part.Within this part, we use the ISAC architecture description language [9] for an automatic generation of the *instruction decoder*. The decoder translates the machine-code instructions into sequences of LLVM IR instructions. The resulting LLVM IR sequence characterizes behaviour of the original instruction independently on the target platform. This intermediate program representation is further analysed and transformed in the static-analysis phase of the front-end. This part is responsible for eliminating statically linked code, detecting the used ABI, recovery of functions, etc. [3]. When debugging information (e.g. DWARF, Microsoft PDB) or symbols are present in the input application, we may utilize them to get more accurate results, see [10].

The output of the front-end part (i.e. LLVM IR code representing input application) is sizable. The main reason is because it reflects a complete behavior of each machine-code instruction, which may not be necessary. For example, each side-effect of an instruction (e.g. setting a register flag based on instruction operands) is represented via the LLVM IR code, but results of these side-effects may not be used anywhere. Therefore, the front-end output is further processed within the middle-end phase, which is built on the top of the LLVM `opt` tool. This phase is responsible for reduction and optimization of this code by using many built-in optimizations available in LLVM as well as our own passes (e.g. optimizations of loops, constant propagation, control-flow graph simplifications).

Finally, the back-end part converts the optimized intermediate representation into the target high-level language (HLL). Currently, we support C and a Python-like language. The latter is very similar to Python, except a few differences—whenever there is no support in Python for a specific construction, we use C-like constructs. During the back-end conversion, high-level control-flow constructs, such as loops and conditional statements, are identified, reconstructed, and further optimized. Finally, it is emitted in the form of the target HLL.

The decompiler is also able to produce the call graph of the decompiled application, control-flow graphs for all functions, and an assembly representation of the application.

```
%a = load i32* @regs0
%b = xor i32 %a, %a
store i32 %b, i32* @regs0
```

(a) Optimized form of an instruction idiom in LLVM IR.

```
store i32 0, i32* @regs0
```

(b) De-optimized form of an instruction idiom in LLVM IR.

Fig. 3: Example of the bit-clear `xor` instruction-idiom transformation.

IV. Idiom Analysis and Reconstruction in the Retargetable Decompiler

The aim of the decompiler presented in the previous section is to allow retargetable decompilation independently on the particular target platform or the used compiler. Therefore, the methods of instruction idiom detection and reconstruction have to be retargetable too. For this reason, we implement these methods within the front-end phase because it uses the unified code representation in the LLVM IR format.

Detection of instruction idioms is based on a detection algorithm operating on the LLVM IR code. LLVM IR is a set of low-level instructions similar to assembly instructions. Moreover, LLVM IR is platform-independent and strongly typed, which meets our requirements. Therefore, machine instructions from different architectures can be easily mapped to sequences of LLVM IR instructions. This brings an ability to implement platform-independent instruction-idiom analysis.

The detection algorithm is similar to a peephole technique used in optimizing compilers [11]. It operates on a basic-block level, where each basic block contains a continuous sequence of instructions described via LLVM IR operations. A particular idiom is detected only if a basic block contains predefined LLVM IR instructions stored in a proper order and they must contain expected operand values (e.g. constant, register number). While examining instructions in an idiom sequence, we may found unrelated instructions (e.g. inserted by a code-motion compiler optimization). The detection algorithm may skip these instructions and continue the search on the following ones, but only if they do not modify the operands of already detected instructions. Otherwise, the search continues behind the current instruction from the beginning. Whenever an instruction idiom is detected, it is substituted by its more readable de-optimized version, once again in the LLVM IR form. The skipped instructions are left untouched on their original positions.

It should be noted that this algorithm does not search for a particular idiom over multiple basic blocks in present. The detection enhancement via usage of a control-flow analysis represents a future research. However, according to our experimental tests, the nowadays common compilers rarely scatter instructions related to idioms over multiple basic blocks. Therefore, the impact of this enhancement may be low.

An example demonstrating this substitution on the LLVM IR level is shown in Fig. 3. Fig. 3a represents the already mentioned `xor` bit-clear instruction idiom. To use register

TABLE I: Shortened list of instruction idioms found in compilers.

Instruction idiom	GNU GCC	Visual Studio C++	Intel C/C++ Compiler	Open Watcom	Borland C Compiler
Less than zero test	✓	×	✓	×	×
Greater equal zero test	✓	×	×	×	×
Bit clear by using `xor`	✓	✓	✓	✓	✓
Bit shift multiplication	✓	✓	✓	✓	✓
Bit shift division	✓	✓	✓	✓	✓
Division by `-2`	✓	×	×	×	×
Expression `-x - 1`	✓	✓	×	×	×
Modulo power of two	✓	✓	✓	×	×
Negation of a float	✓	×	×	×	×
Assign `-1` by using `and`	×	✓	✓	×	×
Multiplication by an invariant	✓	✓	✓	✓	×
Signed modulo by an invariant	✓	×	✓	×	×
Unsigned modulo by an invariant	✓	✓	✓	×	×
Signed division by an invariant	✓	✓	✓	×	×
Unsigned division by an invariant	✓	✓	✓	×	×
Substitution by `copysignf()`	✓	×	×	×	×
Substitution by `fabsf()`	✓	×	×	×	×

content, a register value has to be loaded into a typed variable `%a`. Using the `xor` instruction, all bits are zeroed and the result (in variable `%b`) can be stored back into the same register. To transform this idiom into its de-optimized form, a proper zero assignment has to be done. This de-optimized LLVM IR code is shown in Fig. 3b. In this case, the typed variable `%b` holds zero, which can be directly stored in the register.

In Table I, we can see a shortened list of instruction idioms used in common compilers. This list was retrieved by studying the source codes responsible for code generation (this applies to open-source compilers—GNU GCC 4.7.1 and Open Watcom 1.9) and via reverse engineering of executable files generated by these compilers (this method was used for other compilers—Microsoft Visual Studio C++ Compiler 16 and 17, Borland C++ 5.5.1, and Intel C/C++ Compiler XE13). Some of these instruction idioms are widespread among modern compilers. We have also found out that actively developed compilers, such as GNU GCC, Visual Studio C++, and Intel C/C++ Compiler, are using these optimizations heavily. For example, they generate the `idiv` instruction (fixed signed division) only in rare cases on the Intel x86 architecture; they generate optimized division by using magic number multiplication instead. The decompiler currently supports all of these idioms, among others.

A decompilation of an executable file is a time consuming process. The decompilation time highly depends on the executable size. A good approach how to optimize instruction-idioms analysis is to use any available information to save decompilation time. This is especially important when we support many instruction idioms. Some of them are specific for a particular compiler and therefore, they can be omitted from the detection phase whenever another compiler is detected. On the other hand, detection of the used compiler (as described in [4]) may be inaccurate in some cases and the algorithm will not detect any used compiler. In that case, the idiom analysis tries to detect all the supported idioms. Another optimization approach is to detect only the platform-specific idioms based on the target architecture and omit idioms for other architectures.

Transformation of instruction idioms by using LLVM IR is a quite straightforward task and it is entirely platform independent. On the other hand, the detection of an instruction idiom is more challenging. For example, the expected operand values (e.g. values used for magic number multiplication) may not be stored as clearly as in a original HLL source code. For example, the original HLL constant may not be stored directly as a number (i.e. immediate value), but it may be computed through several machine-code instructions. These instructions fold the original value at run-time based on different resources (e.g. register value, memory content). For example, the MIPS instruction set does not allow direct load of 32-bit immediate value and it has to be done using more instructions (e.g. `lui` and `ori`). Therefore, the operand value is not stored directly within one instruction but it is assembled by an instruction sequence. This is quite complicated because the idiom-detection phase (as well as the rest of the decompiler) is done statically and run-time information is unavailable. To deal with this problem, we utilize a static-code interpreter, originally used for function reconstruction—see [3] for a detailed description of the static-code interpreter.

Using an interpreter to statically compute a value stored in a register is quite common task in instruction-idiom analysis. An example is shown in Fig. 4. An interpreter has to be run to statically compute a number stored in register `@regs3` by using the backtracking of previously used operations and their operands. The obtained result is in this case `680390859`. This number is used in optimized division by number `101` performed by the magic-number multiplication on ARM and the GNU GCC compiler. Another similar issue is accessing the data segment to load constants; the interpreter can solve this issue as well.

The last implementation issue is not related to idiom detection or transformation, but to the processing within the middle-end phase of the decompiler. This phase is responsible for optimization of the LLVM IR code generated by the front-end phase. We exploit the LLVM `opt` and its optimizations for this purpose. For example, it serves to reduce the code size, which has a positive impact on the decompilation results (as

```
%a = add i32 679477248, 0
store i32 %a, i32* @regs3

%b = load i32* @regs3
%b_1 = add i32 913408, 0
%b_2 = add i32 %b_1, %b
store i32 %b_2, i32* @regs3

%c = load i32* @regs3
%c_1 = add i32 203, 0
%c_2 = add i32 %c_1, %c
store i32 %c_2, i32* @regs3

; @regs3 contains value 680390859
; = 203 + 913408 + 679477248
```

Fig. 4: An example of a constant computation in LLVM IR.

has been discussed in Section III). Besides our decompilation project, `opt` is normally used as an optimization part of the LLVM compiler toolchain. However, LLVM is a modern compiler toolchain and it also uses instruction idioms for code optimizations. Therefore, the `opt` tool has tendencies to bring back idioms instead of the de-optimized code. Therefore, we had to disable these optimization passes used in `opt`.

In Fig. 5, we demonstrate reconstruction of another idiom. In this figure, we can compare decompilation results with and without the instruction-idiom analysis. Fig. 5a illustrates a simple C program containing the division idiom. Decompilation result obtained without instruction-idiom analysis is depicted in Fig. 5c. It contains three shift operations and one multiplication by a magic value. Without the knowledge of fundamentals of this idiom, it is almost impossible to understand the resulting code. On the other hand, the decompilation result with instruction-idiom analysis enabled is well readable and a user can focus on a program sense, not on deciphering optimizations done by a compiler, see Fig. 5b.

```
int main(void)
{
  int a;

  /* ... */

  a = a / 10;

  /* ... */
}
```

(a) Input.

```
int main(void)
{
  int a;

  /* ... */

  a = a / 10;

  /* ... */
}
```

(b) Output with idiom analysis enabled.

```
int main(void)
{
  int a;

  /* ... */

  a = (lshr(a * 1717986919, 32) >> 2) -
      (a >> 31);

  /* ... */
}
```

(c) Output with idiom analysis disabled.

Fig. 5: C code example of decompilation with and without the idiom analysis.

```
#include <stdio.h>
int main(void)
{
  int a;

  /* ... */

  printf("1. Multiply: %d\n", a * 4);
  printf("2. Divide: %d\n", a / 8);
  printf("3. >= 0 idiom: %d\n", a >= 0);
  printf("4. Magic sign-div: %d\n", a / 10);
  printf("5. XOR by -1: %d\n", -a - 1);
  return a;
}
```

Fig. 6: C source code used for decompilers' testing.

V. Related Work

The fundamentals of instruction idioms and their usage within compiler optimizations are well documented, see [1, 12–16]. From these publications, we can gain insights into the principles behind instruction idioms as well as how and when to use them to obtain a more effective machine code.

Contrariwise, the detection and reconstruction of instruction idioms from a machine code is mostly an untouched area of machine-code decompilation. This topic is only briefly mentioned in [17–19]. Nevertheless, some of the existing (non-retargetable) decompilers support this feature. In order to observe the state of the art, we look closely on their approaches.

We used a test containing five idioms from a larger list listed in Table I. These idioms are the most common ones (e.g. multiplication via left shift) and the support of idiom reconstruction within a tested decompiler should be easily discovered via these idioms. A source code of this test is listed in Fig. 6. Each expression of the `printf` function represents one instruction idiom, whose meaning is described in Section IV. This source code was compiled for different target platforms (i.e. processor architecture, operating system, and file format) based on their support in each decompiler. Finally, each decompiler was tested by using this executable file and we analysed the decompiled results afterwards.

Boomerang is the only existing open-source machine-code decompiler [20]. However, it is no longer developed. According to our tests, it was able to reconstruct only the first instruction idiom.

REC Studio (also known as REC Decompiler) is freeware, but not an open-source decompiler. It has been actively developed for more than 25 years [21]. None of the instruction idioms was successfully reconstructed. We only noticed that REC Studio can reconstruct the register cleaning idiom (via the `xor` instruction) described in Section I.

SmartDec decompiler is another closed-source decompiler specialising on decompilation of C++ code, see [22] for details.

However, SmartDec was unable to reconstruct any instruction idiom from the machine-code.

Hex-Rays decompiler [23] achieved the best results—three successfully reconstructed idioms from five (it succeeded in the 1st, 2nd, and 4th test). Therefore, we have chosen this decompiler for a deeper comparison with our own solution as described in Section VI.

There are two other interesting projects. The *dcc* decompiler was the first of its kind, but it is unusable for modern real-world decompilation because it is no longer developed [17, 24]. On the other hand, the *Decompile-it.com* project looks promising, but the public beta version [25] is probably still in an early version of development and it cannot handle any of these instruction idioms.

In conclusion, we cannot compare our idiom-detection algorithm with approaches used in other tools because of two reasons. (1) They are either not distributed as open-source. (2) The open-source solutions do not support idiom recovery at all or they support only a very limited number of idioms. On the other hand, we can compare our results with the Hex-Rays Decompiler.

VI. Experimental Results

This section contains an evaluation of the proposed method of instruction-idiom analysis and reconstruction. The decompiled results are compared with the nowadays decompilation "standard"—the Hex-Rays Decompiler [23] that is a plugin to the IDA disassembler [26]. We used the latest versions of these tools, i.e. Hex-Rays Decompiler v1.8.0.130306 and IDA disassembler v6.4.130306. The Hex-Rays Decompiler is not an automatically generated retargetable decompiler, such as our solution, and it supports the Intel x86 and ARM target architectures. Our solution also supports the MIPS architecture at the moment.

All the three mentioned architectures are described as instruction-accurate models in the ISAC language in order to automatically generate our retargetable decompiler. MIPS is a 32-bit processor architecture, which belongs to the RISC processor family. The processor description is based on the MIPS32 Release 2 specification [27]. ARM is also a 32-bit RISC architecture. The ISAC model is based on the ARMv7-A specification with the ARM instruction set [28]. The last architecture used for the comparison is Intel x86 (also known as IA-32) that belongs in the CISC processor family. The model is based on the 32-bit processor core specified in [29] without extensions (e.g. x86-64).

We created 21 test applications in the C language. Each test is focused on a detection and reconstruction of a different instruction idiom. The Minimalist PSPSDK compiler (version 4.3.5) [30] was used for compiling MIPS binaries into the ELF file format, the GNU ARM toolchain (version 4.1.1) [31] for ARM-ELF binaries, and the GNU compiler GCC version 4.7.2 [32] for x86-ELF executables (the 32-bit mode was forced by the `-m32` option).

As can be observed, we used the ELF file format in each test case; however, the same results can be achieved by using the WinPE file format [33, 34]. All three compilers are based on GNU GCC. The reason for its selection is the fact that it allows retargetable compilation to all three target architectures and it also supports most of the idioms specified in Sections II and IV.

Different optimization levels were used in each particular test case. Because of different optimization strategies used in compilers, not every combination of source code, compiler, and its optimization level leads to the production of an instruction idiom within the generated executable file. Therefore, we count only the tests that contain instruction idioms. Furthermore, it is tricky to create a minimal test containing an instruction idiom without its removal by compiler during compilation.

An example of this problem is depicted by using a C code with multiplication idiom in Fig. 7a. The result of this code can be computed during compilation; therefore, the compiler emits directly the result without the code representing its computation (see the example in Fig. 7b). Therefore, we use functions from the standard C library for initialization of variables used in idioms. For example, this can be done by using statements `a = rand();` or `scanf("%d", &a);`. Example of an enhanced test is depicted in Fig. 7c. Such code cannot be eliminated during compilation and the instruction idiom is successfully generated in the executable file, see Fig. 7d.

The testing was performed on Intel Core i5 (3.3 GHz), 16 GB RAM running a Linux-based 64-bit operating system. The GCC compiler (v4.7.2) with optimizations enabled (`O2`) was used for creation of the decompiler.

Finally, we enabled the emission of debugging information in the DWARF standard [35] by using the `g` option because both decompilers exploit this information to produce a more accurate code, see [10] for details. The debugging information help to eliminate inaccuracy of decompilation (e.g. entry-point detection, function reconstruction) that may influence testing. However, the debugging information does not contain information about usage of idioms and therefore, its usage does not affect the idiom-detection accuracy.

All test cases are listed in Table II. The first column represents description of a particular idiom used within the test. The maximal number of points for each test on each architecture is

```c
int main(void)
{
  int a = 1;
  a = a * 8;
  return a;
}
```

(a) Test C code.

```c
int main(void)
{
  return 8;
}
```

(b) Compiler optimized code without instruction idiom.

```c
#include <stdlib.h>
int main(void)
{
  int a = rand();
  a = a * 8;
  return a;
}
```

(c) Enhanced test C code.

```c
#include <stdlib.h>
int main(void)
{
  int a = rand();
  a = a << 3;
  return a;
}
```

(d) Compiler optimized code with an instruction idiom.

Fig. 7: Problem of idiom removal by compiler.

TABLE II: Experimental results—number of successfully detected and reconstructed instruction idioms. Note: several tests differ only in the used numeric constant; however, different instruction idioms are emitted based on this value.

Tested instruction idiom	MIPS			ARM					Intel x86				
		Lissom			Lissom		Hex-Rays			Lissom		Hex-Rays	
	tests	✓	(%)	tests	✓	(%)	✓	(%)	tests	✓	(%)	✓	(%)
`intA = intB < 0`	5	5	100.0	5	5	100.0	0	0.0	5	0	0.0	0	0.0
`intA = intB >= 0`	5	5	100.0	5	5	100.0	0	0.0	5	5	100.0	0	0.0
`intA = 0`	0	-	-	0	-	-	-	-	1	1	100.0	1	100.0
`intA = intB * 4`	5	5	100.0	5	5	100.0	5	100.0	5	5	100.0	5	100.0
`intA = intB / -2`	0	-	-	5	5	100.0	5	100.0	4	0	0.0	4	0.0
`intA = intB / 4`	1	0	0.0	5	5	100.0	5	100.0	4	0	0.0	4	0.0
`intA = intB / 10`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`intA = intB / 120`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`uintA = uintB / 7`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`uintA = uintB / 9`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`intA = -intB - 1`	5	5	100.0	5	5	100.0	0	0.0	5	5	100.0	0	100.0
`intA = intB % 2`	0	-	-	5	4	80.0	2	40.0	4	0	0.0	0	0.0
`intA = intB % 3`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`intA = intB % 5`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`intA = intB % 8`	0	-	-	4	4	100.0	3	75.0	4	0	0.0	1	25.0
`uintA = uintB % 3`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`uintA = uintB % 5`	0	-	-	4	4	100.0	4	100.0	4	0	0.0	4	100.0
`uintA = uintB % 8`	5	5	100.0	5	5	100.0	1	20.0	5	0	0.0	0	0.0
`floatA = -floatB`	5	5	100.0	5	5	100.0	0	0.0	0	-	-	-	-
`floatA = copysign(floatB, floatC)`	5	5	100.0	5	5	100.0	0	0.0	0	-	-	-	-
`floatA = fabs(floatB)`	5	5	100.0	5	5	100.0	0	0.0	0	-	-	-	-
Total	**41**	**40**	**97.6**	**91**	**90**	**98.9**	**53**	**58.2**	**74**	**16**	**21.6**	**47**	**63.5**

five (i.e. one point for each optimization level – `O0`, `O1`, `O2`, `O3`, `Os`). Some idioms are not used by compilers based on the optimization level or target architecture; therefore, the number of total points can be lower than five. For example the MIPS and ARM architectures lack a floating-point unit (FPU) and the essential FPU operations are emulated via *soft-float* idioms. On the other hand, the Intel x86 architecture implements these operations via the x87 floating-point instruction extension; therefore, the instruction idioms are not used in this case.

The decompilation results are depicted in Fig. 8. We can observe four facts based on the results. (1) The Hex-Rays decompiler does not support the MIPS architecture; therefore, we are unable to compare our results on this architecture. (2) Results of the Hex-Rays decompiler on ARM and Intel x86 are very similar (approximately 60%). Its authors covered the most common idioms for both architectures (multiplication via bit shift, division by using magic-number multiplication, etc.). However, the non-traditional idioms are covered only partially or not at all (e.g. integer comparison to zero, floating-point idioms). (3) Our solution achieved almost perfect results on MIPS and ARM; only one test for each architecture failed.

(4) The concept of idiom detection within the front-end phase reaches its limits on the Intel x86 architecture, where the accuracy drops to 20%. The difference between the same tests for ARM (or MIPS) and x86 lies in the complexity of the instruction-semantics description. In general, RISC instructions have only a few side effects (modification of registers, flags, or memory) and their behavioural description in LLVM IR is compact. Therefore, detection of instruction idioms on such smaller pieces of code is quite easy. Contrariwise, almost every CISC instruction has several side-effects and its description in LLVM IR is much more longer. In such long code sequences of LLVM IR, we are unable to detect idioms with higher accuracy. The solution of this problem represents a future research and it is described in Section VII.

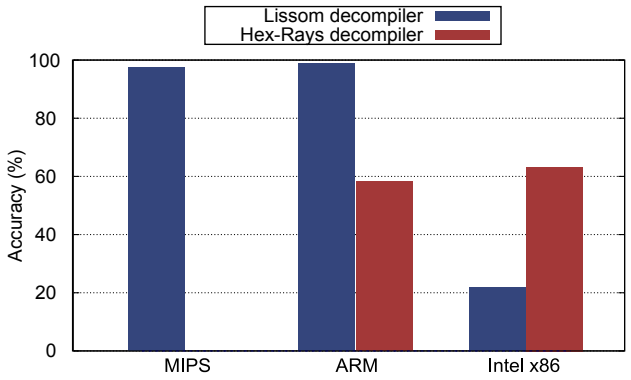

Fig. 8: Results of idiom analysis and reconstruction architecture.

VII. Conclusion

In this paper, we have focused on the problem of instruction idiom detection and reconstruction during the decompilation process of an existing retargetable decompiler. We proposed a new concept of this analysis that has been successfully tested on the MIPS, ARM, and x86 architectures within the Lissom project [2] retargetable decompiler.

In conclusion of the experimental results, our solution is capable to detect and reconstruct instruction idioms for the common RISC architectures with a very good accuracy (i.e. more than 97%), which is better than existing non-retargetable decompilers (some of them lacks this analysis as we demonstrated in Section V). However, the accuracy drops down significantly with the increasing instruction-set complexity. This issue has to be solved in the future research.

The major problem is the code complexity of CISC instructions during the early phases of decompilation (i.e. the front-end phase). On this level, it is problematic to properly detect the idiom without an increase of the false-positive ratio (that

is kept to be zero). However, it should be possible to perform this analysis after the optimization phase (i.e. the middle-end phase) that is based on the LLVM `opt` tool. This optimization phase will simplify the analysed code and it should be possible to detect idioms more easily. Moreover, changing order of this phase is supposed to have a minimal effect on the other phases.

The second propose of the future research lies in further testing of the retargetable idiom detection and reconstruction by using executables created by different compilers and for different target architectures. There is always a room for improvement by adding new instruction idioms into our database of supported idioms.

Finally, usage of control-flow analysis for instruction-idiom detection may be useful when dealing with more aggressive optimizations.

REFERENCES

[1] H. S. Warren, *Hacker's Delight*. Boston, US-MA: Addison-Wesley, 2003.

[2] Lissom, http://www.fit.vutbr.cz/research/groups/lissom/, 2013.

[3] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele, "Detection and recovery of functions and their arguments in a retargetable decompiler," in *19th Working Conference on Reverse Engineering (WCRE'12)*. Kingston, ON, CA: IEEE Computer Society, 2012, pp. 51–60.

[4] J. Křoustek and D. Kolář, "Preprocessing of binary executable files towards retargetable decompilation," in *8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13)*. Nice, FR: International Academy, Research, and Industry Association (IARIA), 2013, pp. 1–6.

[5] J. Křoustek, P. Matula, and L. Ďurfina, "Generic plugin-based convertor of executable file formats and its usage in retargetable decompilation," in *6th International Scientific and Technical Conference (CSIT'11)*, 2011, pp. 127–130.

[6] J. Křoustek and D. Kolář, "Object-file-format description language and its usage in retargetable decompilation," in *AIP Conference Proceedings (SCLIT'12)*, vol. 1479. American Institute of Physics (AIP), 2012, pp. 466–469.

[7] The LLVM Compiler Infrastructure, http://llvm.org/, 2013.

[8] LLVM Assembly Language Reference Manual, http://llvm.org/docs/LangRef.html, 2013.

[9] K. Masařík, *System for Hardware-Software Co-Design*, 1st ed., ser. VUTIUM. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2008.

[10] J. Křoustek, P. Matula, J. Končický, and D. Kolář, "Accurate retargetable decompilation using additional debugging information," in *6th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'12)*. International Academy, Research, and Industry Association (IARIA), 2012, pp. 79–84.

[11] J. W. Davidson and D. B. Whalley, "Quick compilers using peephole optimization," *Software: Practice and Experience*, vol. 19, no. 1, pp. 79–97, 1989.

[12] M. Beeler, R. W. Gosper, and R. Schroeppel, *HAKMEM*. Massechusetts Institute Of Technology, 1972.

[13] R. M. Stallman and the GCC Developer Community, "GNU Compiler Collection Internals," http://gcc.gnu.org/onlinedocs/gccint.pdf, 2010.

[14] W. von Hagen, *The Definitive Guide to GCC*. Apress, 2006.

[15] R. Hyde, *The Art of Assembly Language*. San Francisco, US-CA: No Starch Press, 2003.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge, UK: Cambridge University Press, 2007.

[17] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, School of Computing Science, Queensland University of Technology, Brisbane, QLD, AU, 1994.

[18] M. J. V. Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, University of Queensland, Brisbane, QLD, AU, 2007.

[19] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna, "Advanced static analysis for decompilation using scattered context grammars," in *Applied Computing Conference (ACC'11)*. World Scientific and Engineering Academy and Society (WSEAS), 2011, pp. 164–169.

[20] Boomerang, http://boomerang.sourceforge.net/, 2013.

[21] Reverse Engineering Compiler (REC), http://www.backerstreet.com/rec/rec.htm, 2013.

[22] SmartDec, http://decompilation.info/, 2013.

[23] Hex-Rays Decompiler, www.hex-rays.com/products/decompiler/, 2013.

[24] The dcc Decompiler, http://itee.uq.edu.au/~cristina/dcc.html, 2013.

[25] Decompile-It.com – Online C Decompiler, http://decompile-it.com/, 2013.

[26] IDA Disassembler, www.hex-rays.com/products/ida/, 2013.

[27] MIPS Technologies Inc., *MIPS32 Architecture for Programmers Volume II-A: The MIPS32 Instruction Set*, MIPS MD00086 ed., 2010, https://www.mips.com/products/architectures/mips32/.

[28] ARM Limited, *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, ARM DDI 0406C ed., 2011, https://silver.arm.com/download/download.tm?pv=1199569.

[29] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture," 2013, http://download.intel.com/products/processor/manual/253665.pdf.

[30] Minimalist PSPSDK, http://sourceforge.net/projects/minpspw/, 2013.

[31] GNU ARM Toolchain, http://www.gnuarm.com/, 2012.

[32] GCC: the GNU Compiler Collection, http://gcc.gnu.org/, 2013.

[33] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification," 1995, http://refspecs.freestandards.org/elf/elf.pdf.

[34] Microsoft Corporation, "Microsoft portable executable and common object file format specification," http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, 2013, version 8.3.

[35] *DWARF Debugging Information Format*, 4th ed., DWARF Debugging Information Committee, 2010, http://www.dwarfstd.org/doc/DWARF4.pdf.