UDC 004.3+004.4+004.9

Jakub Křoustek

ON DECOMPILATION OF VLIW EXECUTABLE FILES

Machine-code decompilation (i.e. reverse program compilation) is a process often used in reverse engineering. Its task is to transform a platform-specific executable file into a high-level language representation, which is usually the C language. In present, we can find several such tools that support different target architectures (e.g. Intel x86, MIPS, ARM). These architectures can be classified either as RISC (reduced instruction set computing) or CISC (complex instruction set computing). However, none of the existing decompilers support another major architecture type – VLIW (very long instruction word).

In this paper, we briefly describe the VLIW architecture together with its unique features and we present several novel approaches how to handle these VLIW-specific features in the decompilation process. We focus on handling of instruction lengths, instruction bundling, and data hazards.

Introduction

Decompilation (i.e. reverse compilation) is a process of program transformation, which converts an input low-level program into a higher form of representation. This process can be used for dealing with several security-related issues (e.g. forensics, malware analysis) as well as re-engineering (e.g. migration of legacy code, source-code recovery), see [1–3] for more use-cases.

In this paper, we focus on machine-code decompilation, where the input is a binary executable file containing machine instructions for a particular processor architecture. This type of decompilation is much harder than any other type (e.g. byte-code decompilation) because it deals with a massive lack of information stored within executable files. A retargetable machine-code decompiler is even harder to implement because it tries to be independent of any particular target architecture, operating system, or used compiler.

Despite several attempts of retargetable decompilation, there still exists a family of processor architectures that is not supported by any existing decompiler. It is the VLIW (very long instruction word) family [4]. VLIW processors are used less frequently than RISC and CISC processors (which are well supported in decompilers), but they are very popular in several specific areas, e.g. digital signal processing (DSP).

In this paper, we discuss the most important caveats and pitfalls of the VLIW architecture from the decompilation point of view. Afterwards, we try to address these issues and propose several VLIW decompilation techniques. Those techniques will be used in the existing retargetable decompiler developed within the Lissom project[1] in the near future.

The paper is organized as follows. The next section briefly characterizes the VLIW processor architecture. Then, we discuss existing decompilers and their support of VLIW. Our retargetable decompiler is presented together with an example of its usage in the subsequent section. Afterwards, we depict the most important parts of the VLIW architecture that need to be addressed during decompilation. We also present several approaches how to handle these specific features during decompilation. A discussion of future research closes the paper.

VLIW Architecture Overview

The first reference about the VLIW processor architecture dates back to 1983 [4]. Since this time, all VLIW processors are characteristic by high performance and explicit instruction level parallelism (ILP). The performance speed-up (against RISC and CISC) is achieved via scheduling of a program execution at compilation time. Therefore, there is no need for run-time control mechanisms and hardware can be relatively simple. On the other hand, all constraints checks must be done by the compiler during compilation. These constrains will be described in the subsequent sections.

[1] http://www.fit.vutbr.cz/research/groups/lissom/

Each VLIW instruction specifies a set of operations that are executed in parallel. Each of these operations (also known as *syllables*) are issued and executed simultaneously. VLIW operations are minimal units of execution and are similar to RISC instructions [4]. Whenever the compiler is unable to fully utilize all operation slots, it must fill the gap with a `nop` (No OPeration) operation. This may lead to a rapid performance decrease because instruction cache will be full of inefficient `nop` instructions. Therefore, all the major VLIW processors use some kind of instruction encoding (i.e. compression). It basically packs each instruction into a so-called *bundle* that is smaller in size because the compression removes the `nop` instructions.

From the micro-architectural point of view, VLIW processors consist of clusters with register files and functional units [5]. Functional units are usually specialized. It means that every functional unit has its own task (adder, multiplier, unit for memory access, etc.), which is managed by operations. Therefore, this architecture contains several different decoders, while it usually contains only one fetch unit for fetching the whole long instruction words. Clusters can be interconnected, so data needed for a functional unit in one cluster can be transported from another cluster. This is done by special operations.

Most of the VLIW processors are used in DSP [6], e.g. SHARC by Analog Devices, the C6x DSP family by Texas Instruments (TI), ST2xx family from STMicroelectronics. The most well-known example is Itanium IA-64 by Intel.

State of the Art

Decompilation of RISC and CISC executable code is a well-known topic with history longer than three decades. Contrariwise, VLIW decompilation is mostly an untouched area of machine-code decompilation. Even the most modern decompilers do not support any VLIW architecture. A brief overview of these decompilers follows:

- *Boomerang*[2] is the only existing open-source machine-code decompiler. However, it is no longer developed;
- *REC Studio*[3] (also known as REC Decompiler) is freeware, but not an open-source decompiler. It has been actively developed for more than 25 years;
- *SmartDec*[4] is another closed-source decompiler specialising on the decompilation of C++ code;
- *Hex-Rays* decompiler[5] is a well-known plugin to the commercial IDA disassembler;
- The *dcc*[6] decompiler was the first of its kind, but it is unusable for modern real-world decompilation because it only supports decompilation of DOS executable files. It is also no longer developed;
- The *Decompile-it.com*[7] project looks promising, but the public beta version is probably still in an early version of development.

In table 1, we summarize the supported architectures of the decompilers. Architectures marked with an asterisk (*) are claimed by the authors, but are not included in any publicly available release. In conclusion, we can state that none of the nowadays decompilers supports decompilation of VLIW executable files.

Lissom Project Retargetable Decompiler

The Lissom project's retargetable decompiler aims to be independent of any particular target architecture, operating system, or object-file format. The decompiler is partially automatically generated based on the description of target architecture. For our decompiler, we have chosen the ISAC architecture description language (ADL) that is developed also within the Lissom project.

The ISAC processor model specifies resources (registers, memory, etc.) and the instruction set (i.e. assembler language syn-

[2] http://boomerang.sourceforge.net/

[3] http://www.backerstreet.com/rec/rec.htm

[4] http://decompilation.info/

[5] www.hex-rays.com/products/decompiler/

[6] http://itee.uq.edu.au/~cristina/dcc.html

[7] http://decompile-it.com/

tax, binary encoding, and behavior of each instruction). Furthermore, two decompilation phases (the middle-end and pack-end parts) are built on the top of the LLVM Compiler Infrastructure [7]. The LLVM assembly language (LLVM IR) is used as an internal code representation of decompiled applications in particular decompilation phases. A more detailed description can be found in [1, 8].

The decompiler consists of the preprocessing part and the decompilation core, see Figure 1.

At first, the input binary executable file is analyzed and transformed within the preprocessing part. This part tries to detect the used file format, compiler, and packer, see [8] for details. Afterwards, it unpacks and converts the input platform-dependent application into an internal uniform Common-Object-File-Format (COFF)-based representation. This COFF format is textual for better readability. The conversion is done via our plugin-based converter described in [9].

After the conversion, such a COFF file is processed in the decompilation core, which decodes machine-code instructions, analyses them, and tries to recover HLL constructions (functions, loops, etc.). Finally, it generates the target code in one of the supported languages. Currently, we support the C language and a Python-like language for his purpose. The decompiler is able to process MIPS, ARM, and x86 executables in UNIX ELF, Windows Portable Executable (WinPE), and Apple Mach-O file formats.

To give a brief demonstration of our solution, we present a decompilation of a simple program calculating the Fibonacci function for the Intel x86 architecture. The C source code for this program is given in Figure 2. It was compiled by using the GNU `gcc` compiler (v. 4.7.2) for the Linux/ELF file format. Debugging information and optimizations were disabled (`-O0`). The resulting HLL code generated by our decompiler is shown also in Figure 2. As can be seen, both codes have the same behavior. However, we can notice small differences, such as a usage of a `switch` statement instead of multiple if statements, or missing variables names.

It should be also noted that this decompiler is capable to decompile real-world RISC and CISC malware samples, see [10].

In conclusion, this decompiler is capable to produce a highly accurate code for the supported architectures. The decompilation can be also done online by using the web decompilation service [11].

Table 1. List of supported architectures in the common decompilers

Name	MIPS	SPARC	PPC	ARM	x86	VLIW
Boomerang	x	✓	✓	x	✓	x
REC Studio	✓	✓	x	x	✓	x
SmartDec	x	x	x	x	✓	x
Hex-Rays decompiler	x	x	x	✓	✓	x
dcc	x	x	x	x	✓	x
decompile-it.com	✓*	x	x	✓*	✓	x

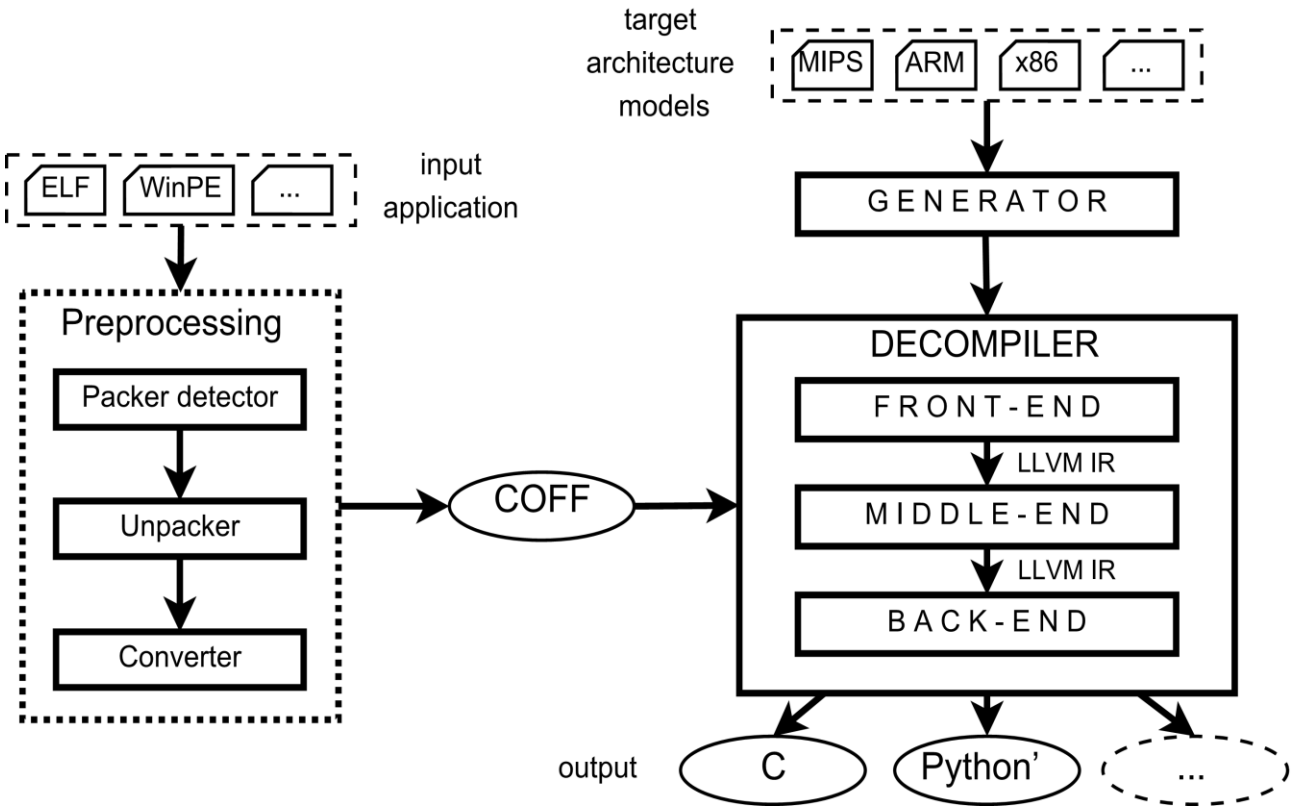

Figure 1. The concept of the Lissom project's retargetable decompiler

```c
int fib(int n) {
  int f;
  if (n == 1)
  {
    return 0;
  }

  if (n == 2)
  {
    return 1;
  }

  f = fib(n - 1) + fib(n - 2);
  return f;
}

int main()
{
  int x = 25;
  return (fib(x) != 46368);
}
```

```c
int32_t fib(int32_t a1) {
  int32_t v1, v2;
  switch (a1) {
    case 1:
      v1 = 0;
      break;
    case 2:
      v1 = 1;
      break;
    default:
      v2 = fib(a1 - 1);
      v1 = v2 + fib(a1 - 2);
      break;
  }
  return v1;
}

int main(int a1, char **a2) {
  return fib(25) != 46368;
}
```

Figure 2. Example of a decompilation process – Fibonacci number computation
(left – input C code, right – decompiled C code)

Decompilation of VLIW Executable Files

According to our analyses, the executable code of VLIW applications differs from the other architectures in several aspects. Those differences are described in the following text and we propose methods how to handle them in a decompilation process.

Instruction Length As the VLIW abbreviation indicates, the VLIW instructions are much larger than instructions on any other architecture (especially RISC). A short comparison of the common VLIW architectures and their instruction (i.e. bundle) lengths is depicted in table 2. It is usual to issue a 256-bit or larger instruction for VLIW architectures, while on RISC it is usually only 16/32/64-bit (based on architecture) instructions [4]. In past, the VLIW architecture allowed even larger lengths, such as 512-bit or 1024-bit [12].

The main pitfall of this difference is related to implementation because not all programming languages and compilers have a proper data type to hold and effectively operate with such large integral numbers. Roughly speaking, in order to decompress and decode such instructions, we must be able to store them in memory. For example, C/C++ does not implicitly support integers larger than 64-bits. Some of its compilers support language extensions (e.g. `__int128` in the GNU `gcc` compiler) however, it is still not enough for all VLIW processors.

The easiest solution is to implement a decompiler in a language supporting arbitrary precision integers (e.g. Python, Haskell). Whenever this solution is not applicable, it is often possible to use some existing library for manipulation of these numbers, e.g. GMP (The GNU Multiple Precision Arithmetic Library) [13], LLVM APInt (Arbitrary Precision Integers) [7], MPIR (Multiple Precision Integers and Rationals) [14]. In general, this solution is slower than usage of native data types. Another approach is to think of instruction as a sequence of bits rather than a large integer. In this case, one can use arrays or strings of bits. However, this approach is even slower.

The last approach suits best to our retargetable decompiler because the input instructions are stored in a textual COFF representation where each bit is stored as a single symbol. Therefore, we can manipulate them as a string of bits.

Instruction Bundling As has been said in the previous sections, VLIW instructions are in most cases stored in an encoded

Table 2. Comparison of common VLIW processors: number of operations, operation lengths, and maximal instruction length

name	manufacturer	ops	op length	instruction length
VEX	J. A. Fisher (HP)	4	32	128
ST2xx	STMicroelectronics	4	32	128
TigerSHARC	Analog Devices	4	32	128
Itanium IA-64	Intel	3	41	128
CHILI	OnDemand	4	40	160
Efficeon	Transmeta	8	32	256
C6x	Texas Instruments	8	32	256

form as bundles. Each architecture uses different method of `nop` compression; however, we can find four basic encoding types, see Figure 3.

Therefore, the very first step of VLIW decompilation is a decompression of operations from a bundle (process so-called *debundling*). Within this step, it is necessary to (1) properly decompress each operation from a bundle and (2) associate the operation to a functional unit. The second part is important because each functional unit (e.g. adder, multiplier) may support different set of operations and an improper association may lead to wrong decoding of such operation.

We have already made a preliminary step for the decompression of VLIW bundles via an enhancement of our ISAC ADL [15]. By using a new `DEBUNDLE` construction, we are able to describe a debundling process, see Figure 4. Based on this description, the decompression routine will be automatically generated in the same way as the current decoder.

During the execution of a VLIW instruction, all of its operations are executed in parallel. VLIW compilers are always responsible for the elimination of dependencies between operations issued in the same instruction because the VLIW architecture lacks of any run-time protection (e.g. out-of-order execution). Those dependencies are called hazards. We will focus on the data hazards.

The data hazard occurs when an operation modifies the same data (e.g. register, memory) as another operation reads/writes. We can find three types of this hazard (hazards are marked bold) [5]:

- Read after Write (RAW), e.g.
  operation1: `reg1 = reg2 + reg3`
  operation2: `reg4 = reg1 + reg2`
- Write after Read (WAR), e.g.
  operation1: `reg1 = reg2 + reg3`
  operation2: `reg3 = reg1 + reg2`
- Write after Write (WAW), e.g.
  operation1: `reg1 = reg2 + reg3`
  operation2: `reg1 = reg4 + reg5`

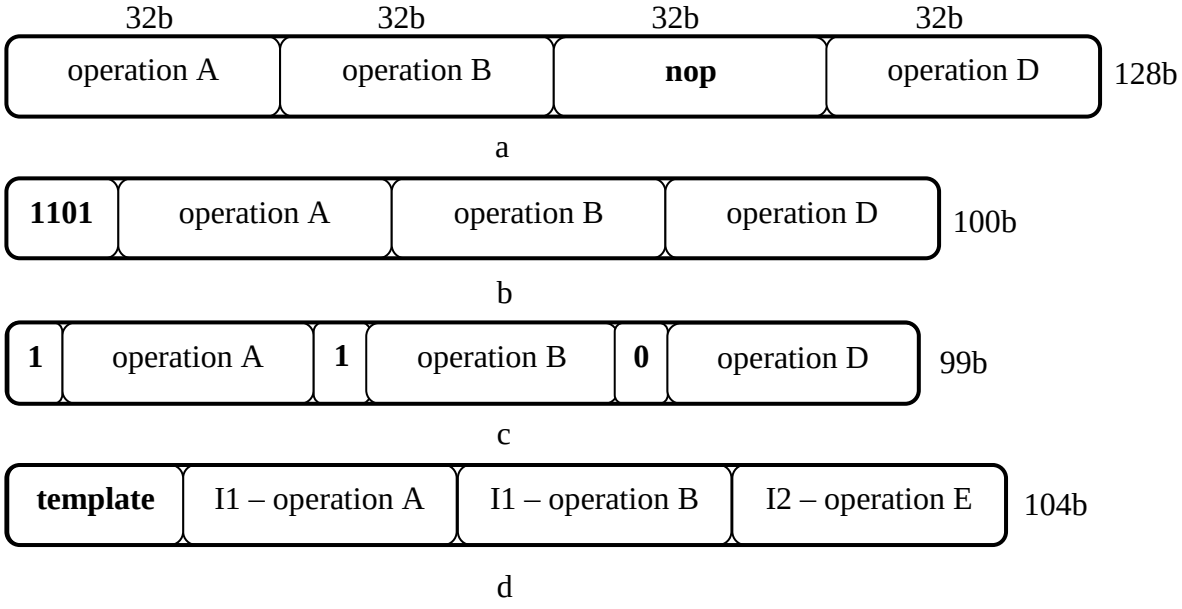

Figure 3. Typical instruction encodings used in VLIW processors.
a) Simple encoding without compression, which is not used in real-world processors.
b) Fixed-overhead encoding, e.g. the Multiflow TRACE architecture.
c) Distributed encoding, e.g. TI C6x, STMicroelectronics ST2xx, Fujitsu FR-V.
d) Template-based encoding, e.g. Intel Itanium, TI C64x+

```
DEBUNDLE
{
   IF (OPCODE_1 == NOP) {                  // 1st slot
      slot_1(NOP_CODING);                  // issue NOP to 1st decoder
   } ELSE {
      slot_1(OPCODE_1 OPERANDS_1);         // issue useful operation
   }
   IF (OPCODE_2 == NOP)     {              // 2nd slot
      slot_2(NOP_CODING);
   } ELSE {
      IF (OPCODE_1 == NOP) {               // control of 1st slot
         slot_2(OPCODE_2 OPERANDS_1);
      } ELSE {
         slot_2(OPCODE_2 OPERANDS_2);
      }
   }
};
```

Figure 4. Example of VLIW debundling description in the ISAC ADL
(a simplified CHILI processor with two operation slots)

Although it should not occur in theory, data hazards are common in practice. Compilers know how each particular architecture reacts on those situations and they can exploit it. For example, they know the order in which the results of operation slots are stored (e.g. the result of the last slot is stored lastly) and they can issue an instruction with such operations.

On the other hand, decompilers are processing instructions sequentially on RISC and CISC architectures – they are decoding and analyzing one instruction after another without their interference [16]. In order to decompile VLIW code, parallel execution of operations has to be supported. Therefore, the information about handling of hazards must be available to the decompiler for each target VLIW processor. It can be done either via a description of instruction semantics or microarchitecture (e.g. pipeline modelling). Both methods are available in ISAC. Afterwards, the decompiler may skip the conflicting effects of operations. For example, the decompiler can ignore the first assignment in the WAW example above whenever it knows that only the last assignment is stored into the same register.

Compilers

The final remark is related to compilers and file formats. According to our research, there is only a limited number of compilers supporting VLIW architectures. For example, the GNU compiler supports Itanium IA-64, TI C6x, and FR-V. Most of the VLIW-friendly compilers use only the ELF as a target file format of executable files. From a decompilation point-of-view, this is promising because it does not differ from other architectures and the same decompilation methods may be applied (e.g. ELF loader, de-optimizations for `gcc`).

However, many of VLIW-processor manufacturers supply their own compiler toolchain (e.g. VEX toolchain, Open64 for Itanium, `st200cc` for ST2xx). Some of these compilers are not publicly available or not distributed as source code. Therefore, it is harder for the decompiler developer to properly test all constructions that may arise in executable code. It should be also noted that any particular compiler may use its own VLIW-code optimizations. This may lead to the implementation of compiler-specific de-optimizations in the decompiler as described in [8].

Conclusion

This paper was focused on the decompilation of VLIW executable files. According to our research, this architecture is not supported by any existing decompiler. There are basically two reasons. Firstly, the VLIW architecture is not so popular as the other ones (RISC and CISC). Secondly, the inner design of VLIW processors significantly differs and it is hard to adapt its constructions and constraints in a decompiler.

The main contribution of this paper is a study of VLIW-specific features and presentation how to handle them within decompilation process. The implementation of these approaches is not ready yet. However, it is planned to adapt them within the Lissom project retargetable decompiler. The preliminary steps (e.g. support of VLIW in the ISAC ADL) were already done. In future, we would like to adapt the remaining approaches presented in this paper. Finally, it will be necessary to analyze VLIW-specific optimizations (software pipelining, hyperblock scheduling, etc.) and reconstruct such code during decompilation.

Acknowledgments

This work was supported by the BUT grant FIT-S-14-2299 Research and application of advanced methods in ICT.

1. *Ďurfina L., Křoustek J., Zemek P., and Kábele B.* Detection and recovery of functions and their arguments in a retargetable decompiler // In 19-th Working Conference on Reverse Engineering (WCRE'12), (Kingston, ON, CA). IEEE Computer Society, 2012. – P. 51–60.
2. *Eilam E.* Reversing: Secrets of Reverse Engineering. Wiley, 2005.
3. *Ďurfina L., Křoustek J., and Zemek P.* Generic source code migration using decompilation // In 10-th Annual Industrial Simulation Conference (ISC'2012). EUROSIS, 2012. – P. 38–42.
4. *Fisher J.A., Faraboschi P., and Young C.* Embedded Computing a VLIW Approach to Architecture, Compilers and Tools. – San Francisco, US-CA: Morgan Kaufmann Publishers, 2005.
5. *Křoustek J., Židek S., Kolář D., and Meduna A.* Exploitation of Scattered Context Grammars to Model VLIW Instruction Constraints // In 12-th Biennial Baltic Electronics Conference (BEC'10). IEEE Computer Society, 2010. – P. 165–168.
6. *Faraboschi P., Brown G., Fisher J.A., Desoll G. and Homewood F.* Lx: A Technology Platform for Customizable VLIW Embedded Processing // In 27-th International Symposium on Computer Architecture (ISCA'00), (New York, US-NY). IEEE Computer Society, 2000. – P. 203–213.
7. *The* LLVM Compiler Infrastructure. http://llvm.org/, 2013.
8. *Křoustek J. and Kolář D.* Preprocessing of binary executables towards retargetable decompilation // In 8-th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13), (Nice, FR). International Academy, Research, and Industry Association (IARIA), 2013. – P. 259–264.
9. *Křoustek J., Matula P., and Ďurfina L.* Generic plugin-based convertor of executable formats and its usage in retargetable decompilation // In 6-th International Scientific and Technical Conference (CSIT'11). Ministry of Education, Science, Youth and Sports of Ukraine, Lviv Polytechnic National University, Institute of Computer Science and Information Technologies, 2011. – P. 127–130.
10. *Ďurfina L., Křoustek J., and Zemek P.* Psyb0t malware: A step-by-step decompilation case study // In 20-th Working Conference on Reverse Engineering (WCRE'13), (Koblenz, DE). IEEE Computer Society, 2013. – P. 449–456.
11. *http*://decompiler.fit.vutbr.cz/decompilation/
12. *Fisher J.A.* Very long instruction word architectures and the ELI-512 // In 10-th Annual International Symposium on Computer Architecture (ISCA '83), (New York, US-NY). ACM, 1983. – P. 140–150.
13. *http*://gmplib.org/
14. *http*://www.mpir.org/
15. *Přikryl Z., Křoustek J., Hruška T., Kolář D., Masařík K., and Husár A.* Design and debugging of parallel architectures using the ISAC language // In Annual International Conference on Advanced Distributed and Parallel Computing and Real-Time and

Embedded Systems (RTES'10). Global Science and Technology Forum (GTSF), 2010. – P. 213–221.

16. *Emmerik M. van and Waddington T.* Using a decompiler for real-world source recovery // In Proceedings of the 11-th Working Conference on Reverse Engineering (WCRE'04), (Washington, DC, USA). IEEE Computer Society, 2004. – P. 27–36.

Data received 18.09.2014

Information about author:

Jakub Křoustek
Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc degree from the same university in 2009. He is currently working on the Lissom research project as the leader of the retargetable decompiler. His current research interests include reverse engineering, malware detection, and compiler design, with special focus on code analysis and reverse translation.

Affiliation:

Faculty of Information Technology,
Brno University of Technology,
Božetěchova 1/2, 612 66 Brno,
Czech Republic.
E-mail: ikroustek@fit.vutbr.cz