Diabatik:
Diablo's instrumentation toolkit

Bruno De Bus
and
Koen De Bosschere
Ghent University, ELIS
bdebus@elis.ugent.be

In this paper, we present Diabatik, a new, retargetable, open-source instrumentation tool that can be used to build program analysis tools for ARM, Alpha, i386, IA64 and MIPS binaries. Diabatik is built on top of Diablo, a retargetable link-time rewriting framework and offers the same user interface as ATOM – another widely used link-time instrumentation tool – to specify analysis tools. Moreover, it offers additional functionality to overcome ATOM's limitations.

1. INTRODUCTION

Collecting run-time information about a program is an important technique used in the design cycle of new computer systems and in computer architecture research. It is also used to analyze the performance of programs and to assist in the development of new development tools. That is why over the past decades many systems for collecting run-time information have been developed. [Pierce, Smith, and Mudge 1995] and [Uhlig 1995] present a nice overview of all the different systems.
Although many systems for collecting runtime information exist, we feel they fail in at least one of the following areas:

—extensibility: most systems can only collect one type of information, and it is hard to collect other information than what they were designed for. Users should be able to specify what information they are interested in;

—retargetability: most systems are designed for one architecture, and can only handle code for this architecture. To verify experimental research result and to compare result on different architectures, it would be useful if more architectures were supported;

—trade-off between speed and accuracy: most systems are either accurate and slow or fast and not very accurate. Users should be able to specify what they prefer.

The above problems have hampered the adoption of many systems for run-time information collection. Diabatik tries to overcome these limitations for one class of runtime information collecting systems: instrumentation tools.

2. INSTRUMENTATION

The most commonly used methods to obtain run-time information are instruction-set emulation and instrumentation. Instruction-set emulators interpret the instruction set in software. To avoid the slowdown of interpretation (1000 times slower is no exception) advanced emulation techniques like dynamic compilation and threaded code execution are used, but even then a slowdown by a factor of 10 is the best result one can expect.
Because this slowdown can be quite cumbersome, instrumentation tools were created. Instrumentation tools work by rewriting the program for which run-time information is required: they insert extra data-collection routines at certain points in the program. To collect the data, the instrumented binaries must then be executed on the architecture they where created for. This makes instrumentation useful for collecting run-time data on existing systems only, but, as all code in the instrumented binaries can execute without interpretation or recompilation, they can be faster then emulators.
Instrumentation can be done at virtually every system level: systems exist that instrument code in the compiler [Larus 1990], the linker [Wall 1992; Srivastava and Eustace 1994], the executable level [Larus and Ball 1992] or even at runtime [Maebe, Ronsse, and De Bosschere 2002]. The latter three systems are generally considered more useful than the first, because they do not need all source code (including libraries!) to be recompiled when instrumenting a program. Instrumenting at the executable level[1] is the hardest to implement, because the targets of indirect jumps are hard to identify at this level. Methods to overcome this problem exist,

[1]We say a tool instruments at the executable level, if that tool can instrument executable programs, by using only the information that is required for executing the program and thus without relying on extra information provided by either the linker or the

but they involve reexecuting and reinstrumenting the executables and thus increase the total time needed to gather information.
Link-time instrumentation fails in the presence of dynamically created code. If dynamically created code must be instrumented, the only viable instrumentation technique is runtime instrumentation. However, since runtime instrumentation cannot analyze the entire program (analyzing the program would impose the same problems as instrumenting the binary at the executable level), a link-time instrumentation tool can do a better job at optimizing the program after instrumentation. Furthermore, a link-time optimizer creates a flowgraph of the program. A runtime instrumentation tool can not do this which makes it harder to feed the obtained information back to other development tools, like compilers and link-time optimizers.
When we started developing Diabatik we favored optimization above the ability to instrument dynamically created code. We therefore opted for link-time instrumentation. If instrumentation of dynamically created code is necessary, have a look at [Maebe, Ronsse, and De Bosschere 2002].

3. EXTENSIBILITY

Most instrumentation tools (and instruction-set emulators for that matter) are tailored toward one specific type of information gathering. They are hard to extend and it is hard to turn off unnecessary functionality. A tool that generates insufficient information is of course of no use to the user. A tool that generates excessive information makes post-processing of the information obligatory to sift out the useful information and introduces more overhead then necessary. [Srivastava and Eustace 1994] designed ATOM to counter this problem for the Alpha architecture. They separated the tool-specific parts of an analysis tool from the low-level instrumentation functionality.
Diabatik uses the same approach as ATOM to overcome the extensibility problem, but it extends ATOM's user interface to allow querying architectural properties for other architectures then the Alpha architecture. Like ATOM it uses two input files to control the instrumentation: the instrumentation code and the analysis code. The analysis code specifies the data-collection routines that will be executed during the execution of the instrumented binary. The instrumentation code specifies where in the execution the analysis routines need to be called and what parameters must be passed to them. The instrumentation file is compiled and linked with the Diabatik library to create a new, analysis specific instrumentation tool. When this tool is run on an executable, it links the analysis code into the executable and adds calls to the analysis code according to the rules specified in the instrumentation code. Because parameters are passed from instrumentation to analysis code the data of the instrumentation code is also linked into the instrumented executable.
In ATOM, analysis code can call any function from the standard C library. Because many of these C library functions maintain an internal state, and because this state is shared between analysis and normal code, calling these functions has the undesirable side-effect that the execution of the instrumented program will differ from the normal execution of the program. In Diabatik using the C library is optional. Diabatik provides its own library that offers basic IO and memory operations, but that has no influence on the internal state of the c library.

4. RETARGETABILITY

Many instrumentation tools can only instrument applications for one architecture. Others (e.g., ATOM) claim to be retargetable, but the backends for other architectures are not available. For research purposes it would be very useful if results from different architectures could be compared. Moreover, a system that is retargetable, and for which many backends exist will have a bigger change of being adopted by people who need an instrumentation tool.
Diabatik uses Diablo [De Bus, Kästner, Chanet, Van Put, and De Sutter 2003] to rewrite binaries. Diablo is designed from the ground up to be retargetable and has been used to rewrite ARM, i386, IA64, MIPS, PowerPC and SH3 Linux binaries. It also works on Alpha/Tru64Unix systems and on ARM ROM images.
Retargeting Diabatik to a system supported by Diablo is easy: all that needs to be done is to specify how function calls are made on that architecture and how registers can be spilled to the stack. Apart from that the library that provides basic input-output and memory operations must be ported to the new architecture. For most architectures it suffices to add the appropriate system calls.
Adding a new architecture to Diablo is more difficult, but can be done in a reasonable time by an experienced programmer. To give an idea, it took one developer two weeks to implement an Alpha backend in Diablo. For the i368 it took a developer a couple of months.

---

compiler. Most so called executable rewriters are actually link-time systems for which the link-time information is passed as extra information in the executable.

Last but not least, both Diablo and Diabatik are open source, so anyone can add new architectures. The code for Diablo can be downloaded from `http://www.elis.ugent.be/diablo`. The Diabatik code is not yet considered to be mature enough for public release, but when it is finished it will be available from the same location.

5. TRADE-OFF BETWEEN SPEED AND ACCURACY

Because instrumentation tools modify the program, and because data-collection routines run in the same address space as the observed code, they have to take great care that the observed behavior of the instrumented program is the same as the behaviour of the original program. In practice, this can be done in two ways. The first is to encapsulate the data-collection routines with code that maps the values of program characteristics that where modified by the instrumentation process to the values before instrumentation. The second is to replace all instructions that produce faulty values in the instrumented binary by code sequences that produce the correct values. An example of an instruction that can produce faulty values after instrumentation is a call instruction. Because instrumenting involves changing the code of the program, it might be the return address changes. To compensate for this, we can replace the call by an instruction that pushes the old return value (the address of the return site before instrumentation) on the stack and a normal jump instruction. Mind that when we change calls in this way, we also need to change all return instructions in the program by code sequences that translates the old return address to the new one, before actually returning.
Because of this, instrumentation tools face a difficult problem:

—if too much compensating code is added to the instrumented binary, the instrumented binary becomes too slow (even slower than with emulation, 100 times slower)

—if, on the other hand, too little compensating code is added, the observed behaviour might differ too much for the results to be usable

ATOM favors execution speed and adds almost no compensating code. The only added compensating code makes sure that the original address of an instruction is returned to the analysis routines, when it is requested. Diabatik favors correctness. Based on the instrumentation routines and the type of parameters passed to these routines it infers which compensating code should be added. However, as this approach might be to conservative, the user can change this behaviour as he sees fit, by specifying which information must be kept correct and what information may change.

6. CONCLUSIONS

We gave a coarse overview of the Diabatik link-time instrumentation tool. The objective of this project is to create an instrumentation system that (i) can easily be extended, (ii) can instrument binaries for different architectures and can easily be retargeted to new architectures and (iii) allows the user to change the trade-off between speed and accuracy.

ACKNOWLEDGMENTS

The work of Bruno De Bus is sponsored by a grant from the Fund for Scientific Research Flanders (IWT).

References

De Bus, B., Kästner, D., Chanet, D., Van Put, L., and De Sutter, B. 2003. Post-pass compaction techniques. *Communications of the ACM 46*, 8, 41–46.

Larus, J. R. 1990. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practices & Experience 20*, 12 (Dec.), 1241–1258.

Larus, J. R. and Ball, T. 1992. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083 (25 Mar.), Computer Sciences Department, University of Winconsin-Madison, Madison, WI, USA.

Maebe, J., Ronsse, M., and De Bosschere, K. 2002. Diota: Dynamic instrumentation, optimization and transformation of applications. In *nternational Conference on Parallel Architectures and Compilation Techniques*, M. Charney and D. Kaeli, Eds. (Charlottesville, Va., Sept. 2002).

Pierce, J., Smith, M. D., and Mudge, T. 1995. Instrumentation tools. In *Fast Simulation of Computer Architectures*, T. M. Conte and C. E. Gimarc, Eds. Boston, MA: Kluwer Academic Publishers.

Srivastava, A. and Eustace, A. 1994. Atom: A System for Building Customized Program Analysis Tools. In *Proceedings of the Conference on Programming Languages Design and Implementation* (June 1994). 196.

Uhlig, U. 1995. *Trap-Driven Memory Simulation*. Ph. D. thesis, EECS Department, University of Michigan, Ann Arbor, MI.

Wall, D. W. 1992. Systems for Late-Code Modification. In *Code Generation - Concepts, Tools, Techniques*, R. Giegerich and S. L. Graham, Eds., 275–293. Springer-Verlag.