LANCET: A Nifty Code Editing Tool

Ludo Van Put Bjorn De Sutter Matias Madou Bruno De Bus
Dominique Chanet Kristof Smits Koen De Bosschere

Ghent University, Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{lvanput,brdsutte,mmadou,bdebus,dchanet,ksmits,kdb}@elis.ugent.be

ABSTRACT

This paper presents LANCET, a multi-platform software visualization tool that enables the inspection of programs at the binary code level. Implemented on top of the link-time rewriting framework DIABLO, LANCET provides several views on the interprocedural control flow graph of a program. These views can be used to navigate through the program, to edit the program in a efficient manner, and to interact with the existing whole-program analyses and optimizations that are implemented in DIABLO or existing applications of DIABLO. As such, LANCET is an ideal tool to examine compiler-generated code, to assist the development of new compiler optimizations, or to optimize assembly code manually.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*graphical environments*; D.3.4 [**Programming Languages**]: Processors—*optimization*

General Terms

Design, Experimentation

Keywords

visualization, optimization, instrumentation, assembler, binary code

1. INTRODUCTION

During the last decades, an extensive range of (graphical) software engineering tools has been developed. These tools most often operate on high level, abstract program representations, such as graphical program blocks (e.g., Labview), screen widgets (Glade), or object-oriented languages (Eclipse, Visual Studio .NET, ...). But despite the ability to work with abstract program representations, there still exist numerous scenarios in which both researchers and developers need to examine assembler code.

A first scenario that comes to mind is the development of compiler optimizations. First, existing code has to be scanned for inefficiencies in order to discover interesting opportunities for new compiler optimizations. Once such an opportunity is found, a compiler researcher might try to apply an the appropriate optimization manually to evaluate its influence on execution speed or power consumption. Obviously this requires the ability to examine and edit the assembler code. If successful, the researcher may start implementing the necessary analyses and the automated transformation in his compiler. At this stage, he needs to verify that the resulting changes to the compiled programs are as intended. Again, assembler code needs to be examined.

Another important scenario is that of an embedded programmer. These days, embedded programs are most often written in higher-level programming languages, of which executable code is generated automatically by compilers. It is not unusual however, that performance critical kernels of an application, such as loops that need to handle streaming media in real time, are optimized manually because the compiler cannot exploit architectural peculiarities. Or that certain (partially) automated optimizations are triggered manually, for example if a developer decides that one hot loop needs to be unrolled while another hot loop, for whatever reason, need not be unrolled.

Finally, sometimes a program does not perform as expected. In that case, a developer needs to be able to study the generated code to get insights in the bottlenecks. Maybe an optimization was unexpectedly not applied by the compiler, or maybe some combination of statements resulted in a particularly bad instruction schedule. In this scenario, it is important that the developer can navigate through the program easily, that he can find the hot spots efficiently, and that he can extract information about those hot spots, such as the results of data flow analysis. The latter can learn the developer why some optimization was not applied.

To the best of our knowledge, there exist no tools today to support the tasks described above on a complete program. To fill that gap, this paper presents LANCET (Lancet is A Nifty Code Editing Tool), a GUI on top of the link-time program editing framework DIABLO. In short, LANCET provides an interface to navigate through a graph representation of a binary program, edit the program, and interact with a wide range of analyses and transformations.

The remainder of this paper is structured as follows. Section 2 provides an overview of LANCET's functionality and its implementation. The use of LANCET in a plethora of user scenarios is discussed in Section 3. Section 4 discusses related work, and conclusions are drawn in Section 5.

2. OVERVIEW

LANCET basically is a graphical user interface to DIABLO [3] (http://www.elis.ugent.be/diablo), a link-time program rewriter that has been used to implement, amongst others, several link-time optimizers [2, 5] and the program instrumentation toolkit FIT [4].

This section presents an overview of LANCET's base functionality and implementation. Some specific user scenarios for which we have implemented specific features in LANCET are discussed in Section 3.

2.1 Functionality

To carry out the scenarios described in the introduction, LANCET supports a number of interactions with the internal program representation that the DIABLO framework builds of a statically linked program.[1] This core representation mainly consists of an interprocedural control flow graph of a whole program. Furthermore, LANCET offers an interface to trigger many of the lower-level and higher-level program analyses and transformations that DIABLO implements. Together, these two interfaces offer the following functionality:

Program Navigation and Examination. LANCET can visualize the call graph of a program and the control flow graphs (CFGs) of the procedures. Obviously, zooming and panning these graphs is possible. Moreover, by hovering over nodes or edges, additional information is presented. For example, when hovering over an edge, this edge is highlighted and the blocks at its head and tail are shown in a pop-up window. This is useful in complex graphs where it is difficult to follow individual edges. Also, when hovering over a basic block, data flow information, such as liveness on register contents, can be shown. Through the color of the nodes in these graphs, users can quickly locate the frequently executed code. The profile information required hereto can be gathered with FIT [4], the instrumentation tool developed on top of DIABLO. This profile information may include execution counts of conditional instructions, and if so, the CFGs indicate by means of colors which instructions are frequently executed. A screenshot showing some graphs is depicted in Figure 1.

Furthermore, lists of basic blocks and procedures can be searched by sorting them on different properties (procedure names, execution counts, addresses, etc.). This also allows for easy targeting of important code.

Finally, it is important to note that all this navigation can happen in multiple windows, on different snap-shots of a program. As such, the user can compare the CFG of a procedure before and after optimizations have been applied.

Program Editing. Besides navigating the graphs of a program, LANCET also provides means to edit the graphs. For example, additional basic blocks can be added to CFGs with a mouse click. Heads and tails of edges in the graph can be dragged and dropped to different basic blocks. Pop-up menus provide more high-level functionality such as splitting basic blocks, forwarding edges, etc.

Furthermore, the instructions in basic blocks can be edited in an instruction editing window. Currently, only the ARM and x86 assemblers are supported, but adding support for additional architectures is a rather simple task. To reschedule code, instructions can also be dragged and dropped. Figure 1 also includes the instruction editing window.

Optimization Triggering. As a large number of interprocedural analyses and optimizations have been implemented in DIABLO while developing link-time optimizers, it was obvious that a user of LANCET should be able to trigger these analyses and optimizations, after which the resulting code can be examined. To that extent, LANCET offers an easy interface to enable/disable all optimizations that a user of DIABLO can otherwise control through command-line options.

2.2 Implementation

To implement the provided functionality, LANCET basically consists of a number of communication channels between three libraries.

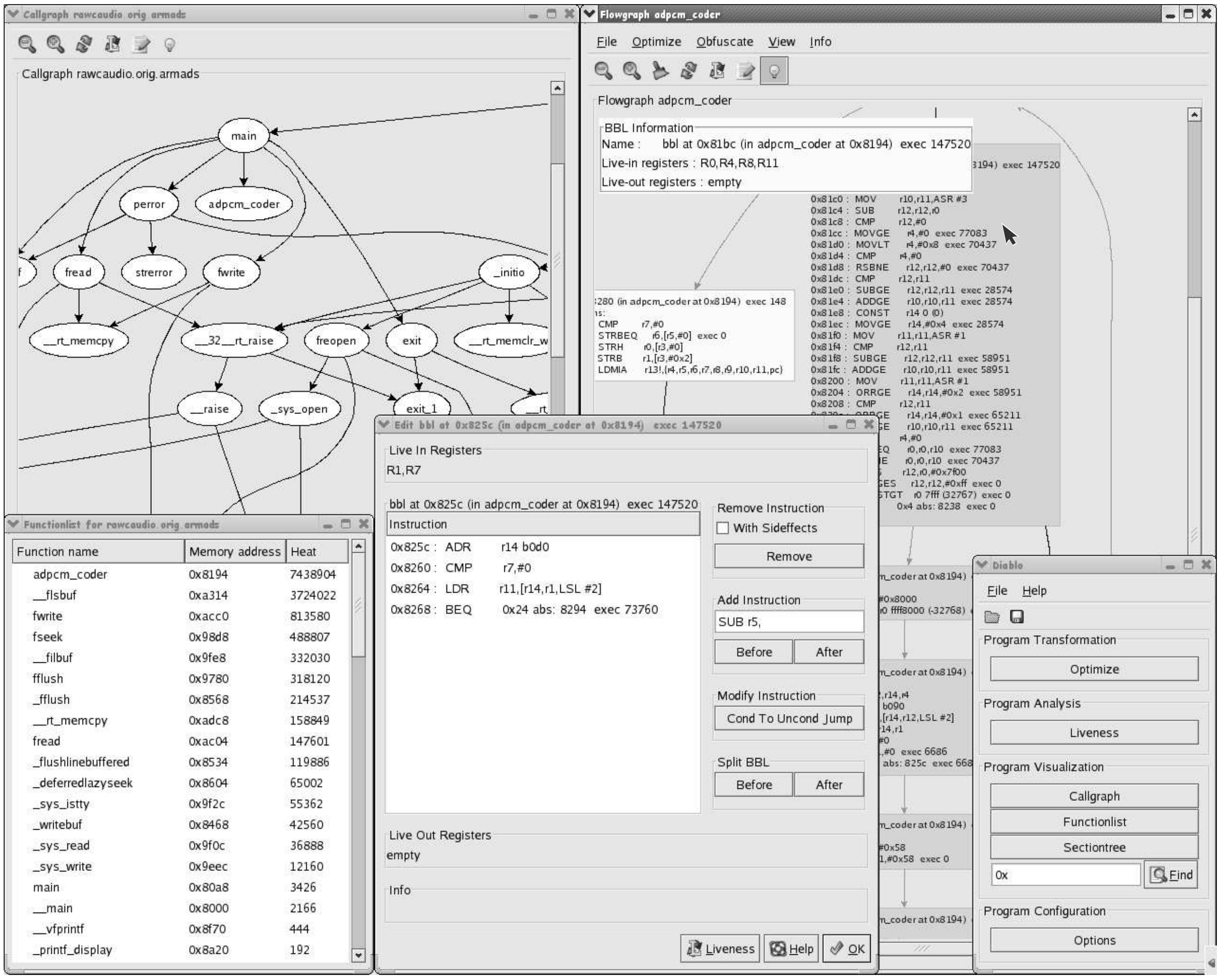
The most important underlying library is DIABLO, our link-time code editing framework. Because DIABLO was previously used in numerous applications, a large number of analyses and code transformations have already been implemented. So on top of the low-level operations that DIABLO offers on its internal program representation, a large number of mid-level to high-level analyses and transformations are available. These range from, e.g, computing interprocedural dominators or removing unreachable code to simply adding instructions to basic blocks, or adding edges to a CFG.

DIABLO these days includes all the necessary backends to rewrite statically linked programs of different architectures (ARM, x86, IA64, Alpha, MIPS), and different object file formats (ECOFF, ELF), and to construct their interprocedural control flow graphs, which contain the disassembled instructions. It are these graphs that the LANCET user views. Currently, LANCET only supports the ARM and x86 targets however.

After a user has edited a program, DIABLO provides all the necessary functionality to write the transformed program to disk again. In this process, all necessary relocation is automatically performed by DIABLO. Unlike simple program editors that only allow local changes in order not to change code size, the user of LANCET can edit the assembler code of a whole program without having to worry about changing addresses calculations. This allows a user to insert instructions, or to replace specific instructions with instructions that occupy more bytes.

Besides DIABLO, we have also used the libraries GTK2 - GNOMECANVAS (http://www.gnome.org) and GRAPHVIZ (http://www.graphviz.org). GRAPHVIZ can layout and visualize graphs that are described in the .dot format, a textual graph description that enables the embedding of additional information in a graph's nodes and edges. We use this capability to include information on nodes and edges that is to be shown in pop-up windows when hovering over the nodes or edges.

[1] DIABLO currently does not work on dynamically linked code or self-modifying code. While the latter poses problems for which we have no obvious solutions yet, the lacking support for dynamically linked code is purely an implementation issue. So far, we simply have not had the research incentive to add support for dynamically linked code.



BBL Information
Name : bbl at 0x81bc (in adpcm_coder at 0x8194) exec 147520
Live-in registers : R0,R4,R8,R11
Live-out registers : empty

```
0x81c0 :  MOV     r10,r11,ASR #3
0x81c4 :  SUB     r12,r12,r0
0x81c8 :  CMP     r12,#0
0x81cc :  MOVGE   r4,#0  exec 77083
0x81d0 :  MOVLT   r4,#0x8  exec 70437
0x81d4 :  CMP     r4,#0
0x81d8 :  RSBNE   r12,r12,#0  exec 70437
0x81dc :  CMP     r12,r11
0x81e0 :  SUBGE   r12,r12,r11  exec 28574
0x81e4 :  ADDGE   r10,r10,r11  exec 28574
0x81e8 :  CONST   r14 0 (0)
0x81ec :  MOVGE   r14,#0x4  exec 28574
0x81f0 :  MOV     r11,r11,ASR #1
0x81f4 :  CMP     r12,r11
0x81f8 :  SUBGE   r12,r12,r11  exec 58951
0x81fc :  ADDGE   r10,r10,r11  exec 58951
0x8200 :  MOV     r11,r11,ASR #1
0x8204 :  ORRGE   r14,r14,#0x2  exec 58951
0x8208 :  CMP     r12,r11
```

```
CMP      r7,#0
STRBEQ   r6,[r5,#0]  exec 0
STRH     r0,[r3,#0]
STRB     r1,[r3,#0x2]
LDMIA    r13!,{r4,r5,r6,r7,r8,r9,r10,r11,pc}
```

Function name	Memory address	Heat
adpcm_coder	0x8194	7438904
__flsbuf	0xa314	3724022
fwrite	0xacc0	813580
fseek	0x98d8	488807
__filbuf	0x9fe8	332030
fflush	0x9780	318120
_fflush	0x8568	214537
__rt_memcpy	0xadc8	158849
fread	0xac04	147601
_flushlinebuffered	0x8534	119886
_deferredlazyseek	0x8604	65002
_sys_istty	0x9f2c	55362
_writebuf	0x8468	42560
_sys_read	0x9f0c	36888
_sys_write	0x9eec	12160
main	0x80a8	3426
__main	0x8000	2166
__vfprintf	0x8f70	444
_printf_display	0x8a20	192

Live In Registers
R1,R7

bbl at 0x825c (in adpcm_coder at 0x8194) exec 147520

```
Instruction
0x825c :  ADR    r14 b0d0
0x8260 :  CMP    r7,#0
0x8264 :  LDR    r11,[r14,r1,LSL #2]
0x8268 :  BEQ    0x24 abs: 8294  exec 73760
```

Add Instruction: SUB r5,

Live Out Registers
empty

Info

Figure 1: A screenshot of Lancet, showing a number of views on the program rawcaudio, compiled for the ARM architecture. In the upper right window, the block entitled “BBL Information” shows the liveness information that appears after hovering over the basic block shown in gray. This gray marks that the block is frequently executed, as can be seen from its execution count of 147520. On the right of each conditional instruction, the number of times its condition evaluated to true is also presented. The upper left window shows part of the program call graph, while the lower left window shows the list of functions in the program, sorted on “heat”, i.e., on number of instructions times their execution count. The middle window shows the instruction editing window. Existing instructions can be unconditionalized, and dragged and dropped. New instructions can be typed (as is partially done on the right side of the window, and prepended or appended to the basic block. Finally, the lower right window is the main window of Lancet, from which different views can be opened, and optimizations or analyses can be triggered.

The actual displaying of all graphs and menus is implemented with GnomeCanvas. GnomeCanvas directly offers clickable objects on a canvas, which facilitated the implementation of interactive graph editing tremendously.

3. APPLICATIONS

This section describes how Lancet supports a number of applications such as the user scenarios described in the introduction. For most of these applications a great deal of functionality has already been implemented, for others the implementation is ongoing or future work. We will mention for each scenario what functionality is not implemented yet at the time of writing.

3.1 Program Visualization

Compiler writers, instruction set architects, embedded systems programmers, and advanced compiler course teachers all need to study machine code, most of which will be generated by compilers. These developers and instructors can all benefit from a graphical presentation of the machine code using control flow graphs that highlight interesting parts.

Having a view on the internals of a program leads to improved understanding of the software. When programming a certain part of an application, it is not always clear how the code fits in the total picture or from where it will be called. A graphical interface that lets the user explore and navigate his program including all library code, from the function level down to the machine code level, can give unusual but useful information. With no source code available, a view on the control flow graph of the program is a necessary requirement to analyze or reverse engineer a piece of software.

With LANCET, a user can easily navigate through the final code of a compiled program. By presenting the user a control flow graph of individual functions, the presented information is detailed and clear, but not overwhelming. At the function level, a user can get an overview of the program, aided by the possibility of viewing a call graph of the complete program. When more detail is necessary, the user can zoom in on a CFG and follow function calls by opening the CFG of a called function with a single click.

3.2 Program Surgery

With the term *program surgery*, we mean the act of *dissecting* a compiled program, *locating* a problem and *interactively editing* the program in order to remove the problem.

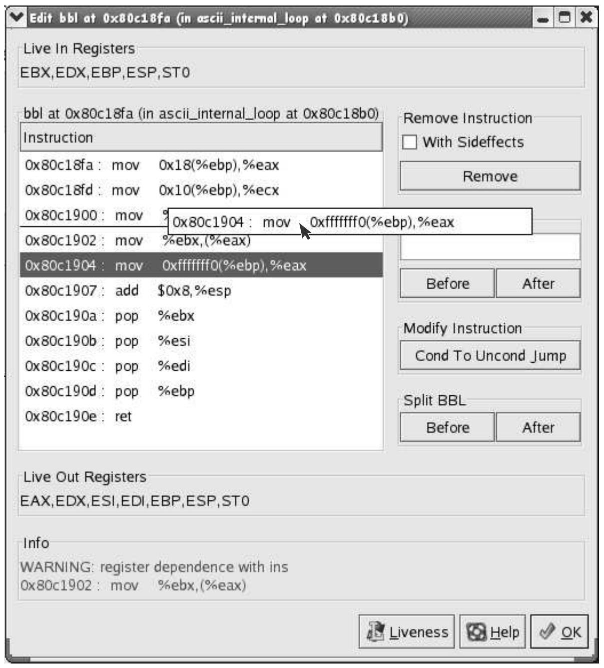
User scenarios that need program surgery include the manual application of known optimizations to kernel loops of embedded applications, manual experimentation with new optimizations, and program cracking.

Obviously LANCET offers the possibility of dissecting a compiled program, including all the linked in library code. In fact, this dissection is all done by the DIABLO framework. Being able to look at library code in the same manner as one looks at his own compiled code, offers an important advantage. More precisely, it allows one to treat all precompiled system libraries, that are most often not available as source code in proprietary development environments, as any other code.

If a program's execution is experienced as problematic (crashing, too slow, etc.), collecting profile information can be done with LANCET's cousin FIT [4]. Using the collected profiles, LANCET's navigation allows easy targeting of the hot or problematic code.

With interactive editing, we mean more than just editing the program as any hacker can do by means of a hexadecimal editor. First, the editing in LANCET is carried out at a much more abstract level, being the level of the program's control flow graph. Not only can one perform simple atomic operations on the control flow graph, such as moving edges, one can also perform more complex, aggregate operations such as adding a preheader to a loop, or duplicating a whole procedure.

More importantly, the editing process can interact with the existing data flow analyses implemented in DIABLO. During the editing process a user can get feedback from LANCET. Figure 2 illustrates this with an example. In the figure, we see the instruction at address 0x80c1904 being moved down with a drag and drop operation. During this operation, the info box at the bottom of the window warns the user that this motion breaks a register dependency. Obviously, this example of supervised, interactive editing is rather trivial. It is not hard to think about more advanced types of


Edit bbl at 0x80c18fa (in ascii_internal_loop at 0x80c18b0)

Live In Registers
EBX,EDX,EBP,ESP,ST0

bbl at 0x80c18fa (in ascii_internal_loop at 0x80c18b0)

```
Instruction
0x80c18fa :  mov   0x18(%ebp),%eax
0x80c18fd :  mov   0x10(%ebp),%ecx
0x80c1900 :  mov   %     0x80c1904 :  mov   0xfffffff0(%ebp),%eax
0x80c1902 :  mov   %ebx,(%eax)
0x80c1904 :  mov   0xfffffff0(%ebp),%eax
0x80c1907 :  add   $0x8,%esp
0x80c190a :  pop   %ebx
0x80c190b :  pop   %esi
0x80c190c :  pop   %edi
0x80c190d :  pop   %ebp
0x80c190e :  ret
```

Live Out Registers
EAX,EDX,ESI,EDI,EBP,ESP,ST0

Info
WARNING: register dependence with ins
0x80c1902 : mov   %ebx,(%eax)

Figure 2: A screenshot showing basic block editing window while an instruction is moved

analyses-supported feedback that can guide the manipulation by a programmer. For example, when inserting new instructions, LANCET may warn the programmer that he is overwriting a live value, and ask whether it is OK to insert the necessary spill code. Also, when eliminating an instruction, LANCET could warn the user of possible side-effects such as a changed stack frame layout.

Finally, this type of interactive editing could be extended to allow the user to specify which optimization should be applied to which code fragment. For example, LANCET could provide a list of all hot loops in the program, together with different unrolling techniques, and ask which techniques should be applied to which loops[2]. In short, LANCET forms an ideal framework to apply assembler code refactoring.

One of the most important advantages of LANCET in these scenarios, is the ability to produce, at any time, a working executable from the edited graph. Because of this, the user can easily test whether the edited program still operates correctly.

3.3 A Priori Optimization Estimation and Rapid Prototyping

When developing a new code optimization, it can be hard to predict its effectiveness without first applying and testing it. In general, a developer can walk two paths. An obvious path consists of manually trying out the transformation and testing the manipulated code. The alternative path is to make a rapid, buggy prototype implementation, and to try to improve the prototype's robustness until it operates correctly on some set of test programs. On both paths, using

[2]This scenario is not implemented because DIABLO does not offer multiple loop unrolling techniques even though it can detect loop in the interprocedural CFG.

graphical code editing software can alleviate the task, as we will describe in this section.

When the first of the above paths is followed, the developer wants to try out a transformation by hand before starting the implementation of an optimization algorithm. By doing so, the developer can evaluate the potential of his transformation *a priori*, and save the implementation effort if the result is disappointing. The implementation effort for a complex transformation can turn out to be high and although applying a transformation by hand could also take some time, *a priori* estimation could save the programmer a significant amount of time.

Many advanced transformations require additional information about the program, such as liveness information. Since it is time-consuming to gather this information by hand, a lack of such information would often prohibit a developer to implement the transformation by hand. LANCET can present the user all information that can be extracted from the program using the analyses that have been implemented in the underlying framework, DIABLO. When the information is presented in a clear way, the programmer can concentrate on his transformation instead of having to spend time analyzing the program.

To demonstrate this, we have added the possibility to show liveness information to the user upon request. In Figure 2, liveness information before and after the basic block is shown at the top and bottom of the assembly code listing. Additional information, like constant propagation information or dominance information can be added easily.

If a developer is more confident about the effectiveness of his transformation, or when manual application has shown that his transformation is worthwhile, he will try to build a working prototype in order to run some tests on a limited set of benchmarks. When the new technique is promising, the prototype implementation has to be refined to account for corner cases and previously unseen code constructs. The refinement of a transformation is typically a lengthy process, as unanticipated corner cases and worst-case scenarios can arise with every new test program that is added to a developers regression test suite, or with every new transformation that triggers bugs in existing ones.

We believe that the use of graphical code editing software and the possibility to interact with the transformation can shorten the refinement process considerably.

3.4 Transformation Visualization, Feedback and Steering

A compiler or whole program rewriter can transform a program in numerous ways. For a programmer or researcher it is however not always clear how code is transformed throughout the optimization process. A graphical code editor can show the differences in the control flow graph before and after a transformation has been applied. With this possibility, the user can understand how his program is transformed, or why it is not transformed the way he intended it.

Transformation visualization opens up even more interesting possibilities. Why not make the user interact with the optimization process? Before a transformation is applied to a certain part of the program, a lot of pre-conditions are evaluated to locate valid optimization targets. This evaluation typically consists of numerous checks, and when one of the checks fails, no optimization is applied. Instead of ignoring the optimization possibilities and bailing out, the user can be prompted for interaction and assist in collecting additional information (*), thus steering the optimization process. For example, if the compiler analyses cannot find a free register needed for an optimization, LANCET can ask the user if he knows a free register. This is especially interesting to optimize manually written assembler code, of which DIABLO cannot make aggressive assumptions about code properties such as calling convention adherence. The assumptions that a compiler cannot make on such code can be complemented with additional information provided by the user.

Furthermore, once the necessary communication channels will be implemented in LANCET to support this scenario, it can also be used to obtain better insights in the application of existing algorithms. For example, it would then be trivial to provide the user with a list of code fragments on which a transformation has been applied, and another list of code fragments on which the transformation was not applied, and for what reason the transformation was not considered applicable (*). With this feedback, the user can easily pinpoint lacking pre-conditions or pre-conditions that can be relaxed because they are in fact too conservative.

We are currently finalizing an interface for DIABLO that allows the specification of pre-conditions to be used both in standard optimizers, and in scenario's where steering or feedback are wanted. To provide the functionality for the scenario's described in this section, LANCET will provide multiple implementations of this interface. At run-time, the implementation can then be chosen at any given moment, thus providing the highest flexibility for the user to switch between fully automated transformations, steering or feedback.

3.5 Point-Wise Instrumentation

The most common way to gather run-time information about a program, is to use an instrumentor or a simulator. Both methods most often come with a run-time overhead however. Instrumentation is generally faster than simulation, but it can still lead to unacceptable slow-downs.

To eliminate some of the overhead, instrumenting uninteresting code can be avoided. Instrumentation tools like FIT [4] or ATOM [10], let the user build custom instrumentors. In an instrumentation code file, the user specifies at which type of program points in the program the instrumentation code from a separate analysis code file has to be inserted. These program points can be basic block entries, single instructions, procedure entries, program exit points, etc. However, when only some specific instructions have to be instrumented, for example to collect the addresses used by a specific load instruction while debugging a program, the instrumentation process can be simplified with the use of a point-and-click instrumentor.

With a graphical interface, a user can explore the program to be instrumented and select the points at which instrumentation code should be inserted. LANCET builds on the same framework that FIT has been built on. Just like in FIT or ATOM, the user can provide the instrumentation code himself in a separate file, but we add the possibility to select it from a built-in collection of common instrumentation routines. As is the case in FIT, all optimizations available in the underlying link-time framework can be used to further minimize the instrumentation overhead.

Point-wise instrumentation lets a user gather detailed run-time information to assist the analysis of the input program. As discussed in Section 3.4, the information can be fed back to the data flow analysis. In future work instrumentation and feedback driven analysis could be integrated in a dedicated environment to combine static and dynamic analysis for program rewriting.

4. RELATED WORK

To the best of our knowledge, we do not know any software tool that offers all functionality that is combined in LANCET. Existing software tools mostly offer a subset of the possibilities available in our tool.

Almost all tool-chains can present the user a textual representation of a program's content. With the `objdump` utility from the GNU Binutils (http://www.gnu.org) for example, a program's sections and symbols can be shown, as well as a disassembled listing of the binary code. Datarescue Ida Pro (http://www.datarescue.com/idabase/) is a multiplatform disassembler that is able to show a control flow graph of disassembled code, but it lacks the capability to edit the program or to incorporate data flow analysis information.

Using the Ida Pro disassembler and GrammaTech's CodeSurfer system (http://www.grammatech.com/) Balakrishnan et al. [1] have built CodeSurfer/x86, a platform for analyzing x86 executables. CodeSurfer/x86 works at the binary level and uses value-set analysis (VSA) to reconstruct an intermediate representation. This intermediate representation can then be 'browsed', similar to the navigation method described in Section 2.1. Using the results of the VSA, an analyst can partially recover the memory behaviour of the executable. CodeSurfer/x86 is designed to analyse the behaviour of malicious programs and offers no rewriting capabilities.

The aiPop optimizer suite (http://www.absint.com/aipop/) is a code compaction framework that works at the assembly code level [7]. A GUI lets the user select the transformations to be applied. aiPop can show CFGs of the input program, but manually editing the graphs or the code, even without supervision, is impossible.

A number of binary rewriters have been described in literature [11, 9, 6]. They are developed for several goals, but none of them offer a graphical interface to the internal program representation or allow fine-grained code editing.

Of all existing tools, the interactive compiler environment VISTA [12] bears the most resemblance to LANCET. VISTA offers most functionality that is offered by LANCET and additionally allows for specifying the optimization ordering, a feature that has been extensively used to study optimal application orders for compiler optimizations [8]. On the other hand, VISTA lacks some of the interesting features ofLANCET.

First of all, VISTA works at the compiler level and works on an RTL (Register Transfer Language) intermediate representation. The user can view the CFG of the compiled code and can edit individual intermediate instructions, but in VISTA the displayed CFG is nothing more than a list of instructions connected by edges. In VISTA, related, connected basic blocks are hence not displayed next to each other. LANCET, because of its use of GRAPHVIZ shows much more practical graphs from which the control flow can be derived easily. Furthermore, VISTA's operation on a compile-time intermediate code representation implies that no whole-program overview is available and thus no whole-program optimizations are possible. Moreover, the use of a generic RTL representation of a program hinders the exploit of architecture specific features. For example, in VISTA, a user cannot insert specific ARM instructions.

5. CONCLUSIONS

We have developed LANCET, a graphical, interactive, multi-platform, link-time binary program rewriter. LANCET provides several views on the internal representation of an input program and a user can navigate through or modify these representations. We described applications that arise from the availability of a graphical program rewriter. Some applications are novel and are solely enabled by the use of a graphical program rewriter. Other possible applications of LANCET simplify or ease existing tasks. We introduced the concept of program surgery and point-wise instrumentation, and we explored the possibility to let a programmer interact with program analysis and optimization. As such, LANCET opens new ways of exploring and modifying compiled programs.

6. ACKNOWLEDGMENTS

The authors would like to thank the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research Flanders (FWO) for their financial support. This research is also partially supported by Ghent University and by the HiPEAC network.

7. REFERENCES

[1] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In R. Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[2] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the linux kernel. In *Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005. To appear.

[3] B. De Bus. *Reliable, Retargetable and Extensible Link-Time Program Rewriting.* PhD thesis, Ghent University, 2005.

[4] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere. The design and implementation of FIT: a flexible instrumentation toolkit. In *PASTE '04: Proc. of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 29–34, New York, NY, USA, 2004. ACM Press.

[5] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004.

[6] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray. Combining global code and data compaction. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 29–38, 2001.

[7] D. Kästner. PROPAN: A retargetable system for postpass optimizations and analyses. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'00)*, 2000.

[8] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proc. of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, New York, NY, USA, 2003. ACM Press.

[9] R. Muth, S. K. Debray, S. A. Watterson, and K. De Bosschere. alto: a link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31(1):67–101, 2001.

[10] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proc. Conference on Programming Languages Design and Implementation (PLDI)*, pages 196–205, 1994.

[11] A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 49–60, 1994.

[12] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones. Vista: a system for interactive code improvement. In *LCTES/SCOPES '02: Proc. of the joint conference on Languages, compilers and tools for embedded systems*, pages 155–164, New York, NY, USA, 2002. ACM Press.