LLDSAL: A Low-Level Domain-Specific Aspect Language for Dynamic Code-Generation and Program Modification

Mathias Payer
Department of Computer Science
ETH Zurich, Switzerland

Boris Bluntschli
Department of Computer Science
ETH Zurich, Switzerland

Thomas Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Dynamic binary translation frameworks enable late modifications of binary programs. The binary translator needs to generate dynamic code at runtime for trampolines, translated control flow instructions, additional runtime checks, and lookups. The code must be efficient, low-level, and cannot rely on any calling conventions.

A Low-Level Domain Specific Aspect Language (LLDSAL) is a natural fit to specify dynamically generated code snippets at compile time. The code is then generated by the translator on demand at runtime and integrated into the translated application code. The LLDSAL is tightly coupled to a host language and provides full access to data structures of the host language. The syntax of the LLDSAL is comparable to inline assembler but the code is generated at runtime. The advantage of an LLDSAL that generates dynamic code is that references to runtime data structures are encoded directly in the machine code without indirections. Most parameters in the generated low-level aspects can be hard coded to reduce the number of passed parameters.

This paper presents the design and implementation of such an LLDSAL. The LLDSAL is integrated into a binary translation framework that enforces application security.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Code generation

General Terms Performance

Keywords Assembly Language, Dynamic Low-Level Domain Specific Language, Aspect Language, Program Modification

1. Introduction

Dynamic Binary Translation (BT) is a well known tool for late and on-the-fly program modifications. The application is dynamically translated at runtime. Additional features and aspects are added during the compilation/translation. A BT framework must dynamically generate (low-level) code sequences to implement the aspects for the individual translation functions, to alter the control flow, to switch between translated code and code of the BT framework, and to apply optimizations that improve the translated code. The required code sequences are not known at compile-time, they must be constructed dynamically at runtime.

The dynamically generated code must be fast and cannot rely on a regular runtime environment or some specific calling conventions (i.e., callee/caller save registers) due to low-level BT details. The BT framework runs alongside the application. Low-level code that transfers control to the BT internal functions must store all registers and code that switches back to the translated application must resume to the same state from before the BT function. Aspects that are woven directly into the translated code must adhere to the same restrictions and may not leave any artifacts after they complete. In addition, the generated code needs access to thread local data structures (e.g., the thread local code cache or the thread local mapping table between translated and untranslated code). The location of these data structures is known only at runtime. Due to features like Address Space Layout Randomization and dynamic runtime data it is not possible to statically foresee the number of threads or the dynamic memory addresses.

BT therefore needs a flexible, fast way to generate low-level machine code at runtime. A Low-Level Domain-Specific Aspect Language (LLDSAL) solves the problem of dynamic on-the-fly code generation for program modifications in a BT. The aspects generated by the LLDSAL are woven into the translated application code at runtime. Aspects are used to change the application behavior, e.g., to change control flow transfer targets, to collect profiling information, or to generate dynamic policies.

The syntax of the DSL is similar to inline assembler in system languages. The code can be specified using an extended assembler syntax with access to high-level data structures. The extensions allow access to C `structs` and fields, access to high-level functions that are part of the BT framework, and rich runtime extensions for control flow transfers.

Listing 1 shows how the DSL is used to implement on-demand low-overhead profiling for unmodified applications in a binary translator. The DSL-code is located in the BT-internal function that translates indirect jumps. LLDSAL generates dynamic code that is added to the translated indirect jump. The DSL resolves the indirect reference to the thread local `nr_ind_calls` variable and uses the final address in the generated machine code. If the instrumented application executes the translated indirect jump it executes the DSL-added code as well.

```
BEGIN_ASM(code)
  // 'incl' does not change flags
  incl {&tld->stat->nr_ind_calls}
END_ASM
```

Listing 1. Dynamic profiling of generated code.

The LLDSAL is compiled into machine code snippets in the final file. These snippets are used to construct machine code sequences at runtime with pointers to dynamic data structures directly encoded in the machine code. The advantage is that no parameters are passed to these low-level aspects. The machine code accesses the hardcoded parameters through direct references and does not use indirect pointers and parameters. This trade-off reduces the number of passed parameters by generating specialized code.

The implementation of the LLDSAL and some design details focus on the IA32 Instruction Set Architecture (ISA) and the C language. The general concepts can be applied to other ISAs and languages as well.

2. Design

A dynamic binary translator needs a way to generate dynamic machine code at runtime to implement aspects that modify the input program, for translated control flow transfers, and for system access (e.g., system calls, and signals). The machine code is specified using an extended assembler syntax. Dynamic code generation is an essential component of the BT framework and should be as simple as possible. A domain specific language for dynamic low-level code generation must fulfill the following requirements:

Usability: the language must be easy to read and easy to maintain. The language follows a trade-off between raw assembly code and access to high-level language constructs of the host language. Compared to related approaches the language should simplify programming and facilitate debugging.

Integration into host language: the DSL integrates naturally into the host language. The DSL supports expressions of a host language inside the DSL code (e.g., access to individual fields of data types of the host language). These expressions are resolved during the DSL integration step in the compiler toolchain.

No runtime dependencies: the DSL must compile to static code that fits into the BT framework. This reduces both runtime overhead and dependencies to external libraries.

The LLDSAL defines a dynamic assembly language to express and generate dynamic low-level code at runtime. The LLDSAL is used in a BT framework to implement the dynamic code generator, the translator, and to weave aspects into the translated application code.

The assembly language is extended with additional semantics to access high-level data structures and variables. Other differences to assembly language are that the LLDSAL includes functions and extensions to redirect control flow to dynamic locations (e.g., to encode targets that are only available at runtime).

2.1 Dynamic assembly language

The BT framework needs a way to specify low-level code that is independent from any calling conventions or runtime environments. Assembly language is a natural fit for these requirements.

Dynamic assembly language in LLDSAL has the same expressiveness as inline assembler in system languages. Any instruction of the target ISA can be expressed using the regular assembler mnemonic. It makes sense to use the general IA32 assembler syntax for the LLDSAL. As our implementation of the LLDSAL is integrated into libdetox [8] and builds on the GNU C compiler toolchain, we choose the AT&T assembler syntax.

```
void eip_gadget() {
  char *eip;
  char *(*code)(void);
  // code must be mapped executable
  code = (char *(*)())mmap(0, 4096, \
    PROT_READ|PROT_WRITE|PROT_EXEC, \
    MAP_PRIVATE|MAP_ANON,-1,0);

  // code generation modifies 'ptr'
  char *ptr = (char *)code;

  BEGIN_ASM(ptr)
    movl (%esp), %eax
    ret
  END_ASM

  // execute generated code
  eip = code();
}
```

Listing 2. DSL example: modification of a local variable.

The LLDSAL is used in a regular program when dynamic code needs to be generated with runtime parameters. The code is generated at a specific address and can include dynamic parameters. `BEGIN_ASM(ptr)` starts a dynamic LLDSAL block and `END_ASM` ends it, generated code is stored at `ptr`. Listing 2 shows a simple example of the LLDSAL that generates a dynamic function that determines the current instruction pointer.

2.2 Data (variable) access

Interaction between LLDSAL and the host language is an important topic. The LLDSAL needs access to variables and state of the host language as well as functions of the host program. LLDSAL uses the `{expr}` expression to evaluate `expr` at runtime in the context of the host language.

```
char *code = ...
BEGIN_ASM(code)
  // store %esp value in some_var
  movl %esp, {&tld->some_var}
END_ASM
```

Listing 3. DSL example: access to structured data.

Listing 3 shows a variable access to `some_var`. The LLDSAL embeds the reference in the generated code at runtime. The generated code uses a direct reference when executed and no longer needs the indirection through `tld->some_var`. Table 1 shows an overview of all possible reference patterns.

Pattern	Description
`${&foo}`	use address of foo
`${foo}`	use static value of foo at translation
`{&foo}`	use dynamic value of foo (indirect reference)
`{foo}`	use dynamic value of the address of foo (double indirect reference)

Table 1. List of supported variable access patterns.

2.3 LLDSAL Macros

Frequently used code fragments are encapsulated in macros. These macros are used like regular statements in the code. LLDSAL extends these statements and executes additional code (e.g., additional checks or logic). The BT framework uses macros to generate direct jumps and direct calls at runtime. The macros take absolute target addresses and calculate relative offsets that are encoded in the IA32 machine code. Additional checks ensure the targets are reachable with the given addressing scheme (e.g., the target of an 8 bit relative conditional jump must be within -128 to 127 bytes from the jump instruction).

Listing 4 shows a DSL snippet that generates a dynamic relative call at the given location. The `call_abs` extension takes a given absolute address, checks the location of the target, and generates a relative call according to the current `code` pointer at the location of the `code` pointer.

```
void function() { ... }

/* later */
char *code = ...
BEGIN_ASM(code)
  call_abs {&function}
END_ASM
```

Listing 4. DSL example that calls a high-level function.

3. Implementation

LLDSAL is implemented as a minimally-invasive extension of the GNU C compiler toolchain. The DSL is resolved in an additional compiler pass between the C preprocessor and the actual compiler. The LLDSAL is translated into C during the compiler pass. The DSL parts of the input source file are replaced with the generated C code in the output source file.

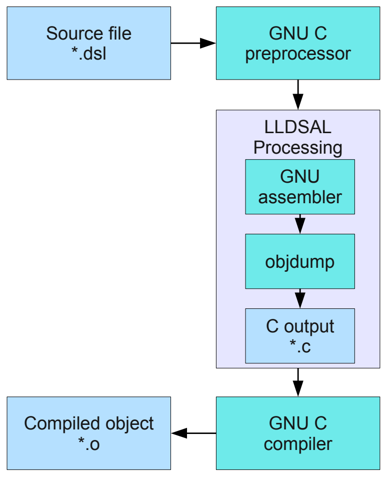

Figure 1. LLDSAL Compilation overview.

Figure 1 shows the implementation of the LLDSAL and the integration into the GCC toolchain. First the GNU C preprocessor is executed. This step resolves all macros in the source file (including macros in the DSL parts). LLDSAL is then implemented as a Python program that strips the assembler code from the input file, executes the GNU assembler to generate the matching machine code sequences, and uses `objdump` to transfer the raw machine code back into the source file. The GNU C preprocessor adds explicit line number information from the original input source file. The LLDSAL keeps this information intact and weaves the generated machine code into a temporary C file. The source file is then compiled using the GNU C compiler.

The LLDSAL separates the low-level assembler code and the C code. If the assembler code contains syntax errors then the assembler in the LLDSAL step emits a warning or an error that is mapped back to the original location using the available line number information. On the other hand if the C code contains any syntax or semantic errors then the compiler can emit warnings or errors using accurate line number information.

Listing 5 shows a simple translation from the C code enriched with LLDSAL code to regular C code. The assembly code that is part of the LLDSAL block is extracted in the additional compiler pass, translated using the GNU assembler, disassembled by objdump, and the dumped machine code is integrated back into the temporary C file.

```
  char *code = ...
  BEGIN_ASM(code)
    nop
    cmp $0, {tld->maptbl}(, %ebx, 8)
  END_ASM

/* is translated to */

  // BEGIN_ASM
  // nop
  *(code++) = 0x90;
  // cmpl
  *(code++) = 0x83;
  *(code++) = 0x3c;
  *(code++) = 0xdd;
  *((int*)(code)) = \
    (int)(tld->maptbl);
  code += 4;
  // END_ASM
```

Listing 5. Simple DSL translation.

4. Implementation alternatives

LLDSAL uses an additional compiler pass to translate the DSL into regular code of the host language. Other approaches are offline code generation (relying on inline assembly directives), macro-based approaches, or feature-rich JIT code generation toolkits and libraries.

4.1 Inline assembler

Inline assembler is a domain specific language that is available on top of many system languages like C or C++. The syntax of LLDSAL is similar to inline assembler. Both domain specific languages enable the combination of assembly code with access to high-level language constructs.

The difference between LLDSAL and inline assembler is that inline assembler produces static code at compile time. Source code written in inline assembler is directly integrated into the compiled program as part of individual compiled functions. LLDSAL, on the other hand, generates dynamic code at runtime and not at compile time. LLDSAL can be used for just-in-time code generation and self-modifying code.

Inline assembler cannot encode direct pointers to dynamic or thread local data. The LLDSAL encodes direct references to data structures in the machine code. Inline assembler needs parameters to access dynamic data structures, resulting in additional (unavoidable) overhead. Dynamically generated code optimizes the accesses to these data structures, removes parameters, and reduces indirect memory lookups.

4.2 Macro-based approach

C macros can be used to specify low-level instructions in a semi-comfortable way. The machine code representation is encoded in a macro that uses dynamic parameters as well. The macros are then used to generate machine code at dynamic locations at runtime.

```
#define PUSHL_IMM32(dst, imm) \\
  *dst++=0x68; \\
  *((int32_t*)dst) = imm; dst+=4

  char *code = ...
  PUSHL_IMM32(code, 0xdeadbeef);
```

Listing 6. Macro based translation.

Listing 6 shows a macro-based implementation. The state of the source file after the GNU C preprocessor step is comparable to the state after the LLDSAL is integrated into the final C file. Drawbacks of the macro-based approach are that every used machine code instruction needs to be hand-coded as a macro, changes to the program layout are cumbersome, and (manually) hand-coding machine code instructions is error prone.

Another drawback of this macro-based approach is that the compiler cannot statically check the dynamic assembly code. LLDSAL uses the compiler to check type sizes, assembly mnemonics, and the correct encoding of instructions at compile time.

4.3 JIT code generation

Another alternative is using just-in-time code generation toolkits like GNU Lightning [1] or asmjit [6] that are more complex than simple compilers with inline assembler directives. These projects provide an interface to generate dynamic low-level code at runtime based on a high-level interface.

The high-level languages are different from the assembler syntax. The APIs are geared towards complex just-in-time code generation, e.g., for virtual machines. Both libraries add additional dependencies to the runtime image of the application and support features like CPU/ISA detection and code optimization. These features are not needed in a table-based BT framework and only add unneeded additional complexity and overhead.

JIT-based code generation can check the generated code at runtime and throws a runtime exception if there is an error. Due to the library based implementation no compile-time check is possible. On the other hand a full JIT compiler is more flexible. JIT compilers can compile dynamic source input, the LLDSAL must know at compile-time which instruction sequences are used at runtime.

5. Related work

A rich body of related work in the area of domain specific languages already exists. This paper focuses on two aspects of related work: (i) the implementation of the DS(A)L, i.e., focuses on implementation issues and other problems with the integration of the DSL into the existing host language, and (ii) DSL applications that look at specific application scenarios for DSLs (e.g., meta-circular DSL languages, and virtual machines).

5.1 DSL implementation

The specified syntax of the DSL must be integrated into the selected environment. Many implementations choose source-to-source translations to integrate the DSL into the host language. Hudak et al. [5] and Mernik et al. [7] generalized the approaches how to build domain specific languages. They also discussed the possible scenarios when DSLs are useful. Stichnoth and Gross [10] present code decomposition to implement high-level transformations in compilers using a DSL.

Porkolab et al. [9] developed a compile-time DSL parsing and generation technique. The DSL is integrated into the host language and causes minimal syntactical and semantic overhead. They propose a template-based implementation to integrate a DSL into C++. Our approach is similar in that we also use an additional compiler pass to translate the DSL into the host language, reducing overall complexity.

5.2 DSL applications

DSLs fill a special need in a specific environment. Guyer and Lin [4] implement a specific DSL to optimize libraries for different runtime environments. The DSL uses high-level information to select specific optimizations for the library that fit the target environment. This makes it easier to develop the best fitting library for each individual configuration.

Khepora [3] abstracts the idea of DSLs by implementing a system to build source-to-source translated DSLs. The Khepora system is implemented as a simple compiler that parses the source file, resolves the DSL in the syntax tree, and pretty prints the final file in the host language.

Coady et al. [2] present a DSAL for virtual machines. The VM is implemented and extended using modular aspects. Individual aspects and add-ons can be implemented easily using the given DSAL. D4OL [11] is a specific DSAL that is used to specify object layouts for virtual machine implementations. The object layout can be adapted according to specific needs and constraints.

The implementation of LLDSAL is simple and straightforward. We choose to implement the DSAL directly using an additional compiler pass. Other approaches like source-to-source translation would be possible but would add additional complexity due to, e.g., the direct access to parameters in the host language, and additional dependencies on external libraries.

6. Conclusion

This paper presents the design and implementation of a Low-Level Domain-Specific Aspect Language (LLDSAL). The LLDSAL enables dynamic code generation in system languages without additional library dependencies. LLDSAL is used to add features and aspects dynamically during the execution of application code in a binary translator.

The comparison with other approaches shows that a compiler-based implementation of the DSL is feasible and no additional runtime libraries are needed. Using a low-level DSAL raises the level of interaction between developer and BT framework and thereby significantly adds to readability and maintainability of system software.

References

[1] Using and porting gnu lightning. `http://www.gnu.org/s/lightning/`, Dec. 2011.

[2] Coady, Y., Gibbs, C., Haupt, M., Vitek, J., and Yamauchi, H. Towards a domain-specific aspect language for virtual machines. In *DSAL'06: Proc. AOSD workshop on Domain-Specific Aspect Languages* (2006).

[3] Faith, R. E., Nyland, L. S., and Prins, J. Khepera: A system for rapid implementation of domain specific languages. In *DSL* (1997), pp. 243–255.

[4] Guyer, S. Z., and Lin, C. An annotation language for optimizing software libraries. In *DSL* (1999), pp. 39–52.

[5] Hudak, P. Building domain-specific embedded languages. *ACM Comput. Surv. 28* (Dec. 1996).

[6] Kobalicek, P. asmjit - complete x86/x64 JIT assembler for c++ language. `http://code.google.com/p/asmjit/`, Dec. 2011.

[7] Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv. 37* (Dec. 2005), 316–344.

[8] Payer, M., and Gross, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l conf. Virtual Execution Environments* (2011), pp. 157–168.

[9] Porkolab, Z., and Sinkovics, A. Domain-specific language integration with compile-time parser generator library. In *GPCE '10: Proc. 9th Conf. on Generative Programming and Component Engineering* (2010), pp. 137–146.

[10] Stichnoth, J. M., and Gross, T. R. Code composition as an implementation language for compilers. In *DSL* (1997), pp. 119–132.

[11] Timbermont, S., Adams, B., and Haupt, M. Towards a dsal for object layout in virtual machines. In *DSAL'08: Proc. AOSD workshop on Domain-Specific Aspect Languages* (2008).

The source code of the LLDSAL framework and additional examples can be downloaded at `http://nebelwelt.net/projects/libdetox`.