Heterogeneous Architectures Programming Library

G.V. Kirgizov <gkirgizov@gmail.com>
I.A. Kirilenko <y.kirilenko@spbu.ru>
Software Engineering Department,
Saint Petersburg State University,
University Embankment, 7, Saint Petersburg, 199034, Russia.

Abstract. Embedded platforms with heterogeneous architecture, considered in this paper, consist of one primary and one or more secondary processors. Development of software systems for these platforms poses substantial difficulties, requiring a distinct set of tools for each constituent of the heterogeneous system. It also makes achieving high efficiency the more difficult task. Moreover, many use cases of embedded systems require runtime configuration, that cannot be easily achieved with usual approaches. This work presents a C-like metaprogramming DSL and a library that provides a unified interface for programming secondary processors of heterogeneous systems with this DSL. Together they help to resolve aforementioned problems. The DSL is embedded in C++ and allows to freely manipulate its expressions and thus embodies the idea of generative programming, when the expressive power of high-level C++ language is used to compose pieces of low-level DSL code. Together with other features, such as generic DSL functions, it makes the DSL a flexible and powerful tool for dynamic code generation. The approach behind the library is dynamic compilation: the DSL is translated to LLVM IR and then compiled to native executable code at runtime. It opens a possibility of dynamic code optimizations, e.g. runtime function specialization for specific parameters known only at runtime. Flexible library architecture allows simple extensibility to any target platform supported by LLVM. At the end of the paper a system approbation on different platforms and a demonstration of dynamic optimizations capability are presented.

Keywords: metaprogramming; code generation; embedded DSL; heterogeneous systems; embedded systems.

DOI: 10.15514/ISPRAS-2018-30(4)-3

For citation: Kirgizov G.V., Kirilenko I.A. Heterogeneous Architectures Programming Library. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018, pp. 45-62. DOI: 10.15514/ISPRAS-2018-30(3)-3

1. Introduction

Embedded systems have been in a widespread use a long time, and today they become even more relevant because of the rapid development and adoption of new application fields, for example, Internet-of-Things, ”smart houses” and robotics.

Many of the embedded systems used in these areas have heterogeneous architectures due to nature of their tasks. Typically, they consist of one primary, more powerful processor that executes the main program and performs common control, and one or several secondary microcontrollers or processors that provide read/write access to sensors and peripheral devices or may perform some other special functions. Examples of such systems may be: Raspberry Pi (main) + Arduino with Atmel AVR (peripheral) and Odroid XU4 (main) + stm32f4 microcontroller (peripheral).

Heterogeneity of these systems causes noticeable overhead. Traditional development workflow requires use of IDEs and toolchains that are specific for each part of the system. This need to develop each part of the system in a separate project using a different set of platform-specific tools makes system development processes more complex and expensive. The amounts of resources required for support and changes also grow.

The efficiency of the system suffers too. Due to specificities of each microcontroller and their limited hardware capabilities they often have only basic firmware, which only capabilities are reading sensors, communicating results back to main processor, receiving data and control commands from it and writing the received data to special registers of peripheral devices. All core program logic is contained on the primary processor, and, as secondary processors/microcontrollers do not contain even a part of this logic, constant communication between them is unavoidable (because of the nature of control cycle: request sensor data, wait for it to arrive, compute control output, send it back to the secondary processors, repeat).

This work is based on preliminary results of [1] that showed the viability of the idea of dynamic code generation. We revise previous architectural choices, fully reimplement the library because of shortcomings of existing implementation and substantially extend it in terms of functionality and possible applications/uses.

In particular, the new DSL is completely abstracted from other parts of the library and can be used independently in other projects based on the idea of metaprogramming. Moreover, the new DSL implementation allows employing various dynamic optimizations, which are not possible in heterogeneous systems using traditional programming techniques. The contribution of this work is twofold. We present:

- C++ embedded DSL for dynamic metaprogramming;
- a library that simplifies development of programs for heterogeneous systems providing unified programming interface; it also allows to achieve higher efficiency of the system and implement better organizations of work between its parts.

The library is based on the idea of a dynamic compilation of programs for peripheral processors.

We also demonstrate system's capabilities on a number of examples that show important features of the new DSL and some applications in embedded systems domain. Source code with build instructions can be found in the project repository[1].

Several possible use cases of this library can be imagined. First use case is avoiding the overhead of constant communication between processors. Of course, it's possible to accomplish it without this library: move part of the program logic to peripheral processors on top of their basic firmware. However, with usual tools, it incurs additional costs for development and support because with this approach there is no more single point of change in core logic of the system. There is unavoidable need to support several projects and ensure proper integration. Whereas presented library allows avoiding both communication overhead and unnecessary complexity of the development process.

The second use case is to allow dynamic specialization of heterogeneous systems for their operating environment. Some types of embedded heterogeneous systems can be deployed in a wide range of environments with various conditions. When their operation depends on these conditions, developers of programs for such systems must anticipate in the code all possible conditions. It may be implemented through constant monitoring of the environment. Another alternative is on-place configuration or tuning of each particular system. However, it may not be possible due to nature of the task or too often or rapid (for manual operating) changes of the environment. Another variation of dynamic specialization scenario is a runtime configuration for specific peripheral devices (e.g. different models of sensors and actuators).

Our library can help there in the case of sufficiently slowly changing environment (relative to a number of control cycles, when the time required for dynamic recompilation will pay off). It can be better shown on the specific example of PID controller tuning. Firstly, PID controller with tuning subpro gram is loaded on the peripheral microcontroller. Then, when optimal parameters are found, microcontroller program can be recompiled with these particular coefficients, thus yielding system that is maximally suited for its operating conditions. For the specific case of not changing environment this tuning and dynamic recompilation can be executed only once on deployment. This example is elaborated on in greater detail in the section Demonstration.

The paper is organized as follows. The next section discusses similar works that are based on the similar ideas. The third section describes main architectural decisions and presents the architecture of the system. The fourth section is devoted to the DSL and provides a reader with a number of examples. The following section describes other parts of the system and their functionality in greater detail. The Approbation section describes test setups and the Demonstration section shows benefits of

[1] https://github.com/gkirgizov/hetarch

dynamic recompilation on a specific example and discusses scope and applicability of the library. The paper is closed with conclusion and discussion of possible directions of further work.

2. Similar Work

The difficulties, which heterogeneous systems cause, are not unique for the embedded software engineering. Programming of heterogeneous systems is an old problem, and there are several conceptual approaches to aforementioned difficulties.

The most known area that faces it is programming with graphical processors. In this case, heterogeneous system consists of CPU and one or more GPUs. (The case of graphics programming, i.e. using shaders and graphics pipelines, is further from heterogeneous programming and is not considered here.) It is an old problem in this field: how to effectively and, not less importantly, conveniently use GPU in usual, CPU-centric programs? There are two main examples of systems that answer this question: Open Computing Language (OpenCL) [2] and CUDA framework from Nvidia [3]. Both these frameworks propose the use of C and C++ languages extended with special functions and attributes for writing device code (code to be executed on secondary processors). It can be written, depending on user’s aims and requirements, either in separate files or in the main program files together with usual C/C++ host code that is intended to be executed on CPU. OpenCL uses dynamic compilation (at runtime) of device code; some device vendors provide offline compilers for their devices (for example, Intel Code Builder for OpenCL API). CUDA similarly provides both possibilities: Nvidia has an offline compiler called NVCC and a runtime compilation library NVRTC.

The motivation behind these examples and presented in this paper library is essentially the same: use of the same programming interface for all constituents of a heterogeneous system.

Another area that this work touches is the ideas of generative, multi-stage programming and runtime code generation. A good discussion of general motivations and trade-offs behind these ideas, as well as examples of some actual realizations and a number of references provides [4].

Among their examples Delite—a heterogeneous parallel framework for domain-specific languages [5], [6]—is of particular interest. Delite’s focus is on the performance of parallel heterogeneous systems, e.g. mixed CPU/GPU architectures and clusters. It is built on top of Lightweight Modular Staging (LMS) [7] system, that makes use of a form of metaprogramming to construct a symbolic representation of a DSL program. LMS provides a basis for DSLs embedded in Scala. On top of this layer, Delite is structured into a compiler framework and a runtime component. The framework provides primitives for parallel operations and generates Scala, CUDA or C++ code from DSLs.

Although both we and the authors of Delite start from the same idea of multi-stage programming, our systems significantly differ in the approaches and application domains. Most importantly, we use dynamic code generation and thus employ the generative programming at runtime to achieve dynamic optimizations. The authors of Delite, on the other hand, require static compilation of DSLs—they promote the use of additional compilation stage to perform domain-specific optimizations.

3. High Level Description

Further in the text by the word host is meant primary processor, by target—one of the peripheral processors or microcontrollers, by the user—developer who uses this library.

3.1 Main Architectural Decisions

The following decisions have shown themselves as reasonable and grounded and thus are inherited from the previous work [1]. They are discussed here to provide better context.

Runtime changes in executable code on targets can be achieved by two approaches: dynamic compilation, which happens on the host, and code interpretation which happens on targets. Because modern interpreted languages generally have higher requirements and cause more overhead, the first decision is to use dynamic compilation on the more powerful host.

The second decision is to use embedded domain specific language (DSL) as a basis for dynamic code generation. An alternative of using code attributes with compiler extension (e.g. as used by OpenCL) is less viable due to several reasons. First, code defined in a such way can be manipulated at the runtime only as a string of characters. It complicates analysis and dynamic code specialization, requiring additional step of semantic analysis before that, whereas DSL approach gives semantic information ’for free’. Second, it is more demanding to maintain the compiler extension to keep it up-to-date with the needed compiler versions. In addition, it is still necessary to use dynamic compilation tools. It seems excessive to support both the compiler extension and the dynamic compilation tools. Moreover, it would restrict library users to only one compiler, which can be especially inconvenient in the world of embedded systems.

LLVM [8] is used as a compilation backend. There is no real alternative, and its excellent design and convenience of use made this work possible.

C++ is chosen as a language of implementation by several reasons: firstly, it is a natural choice for embedded systems domain; secondly, it allows to avoid overhead of interfacing with LLVM; and, most importantly, with template metaprogramming it provides the necessary expressive power for implementation of the DSL, which itself must be very expressive and general to be applicable in a wide range of use cases. Specifically, the latest C++17 standard is used.

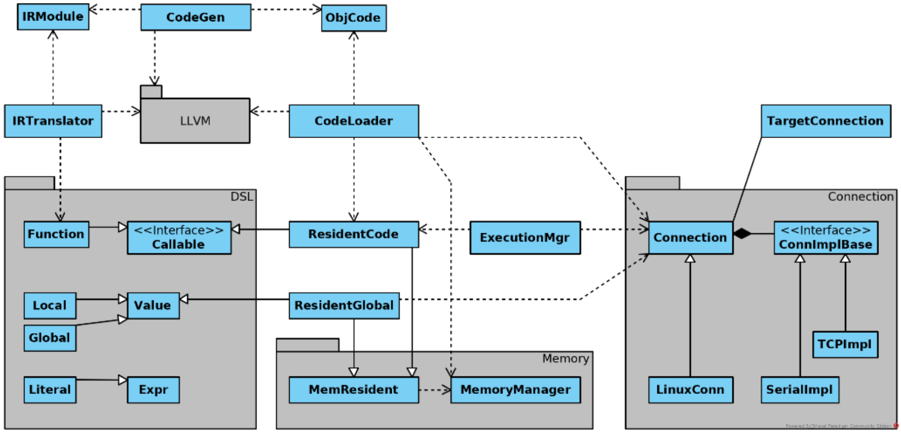

Fig. 1. UML class diagram of the system. DSL class hierarchy is shown only approximately because of its breadth and dynamic nature. IRTranslator together with non-resident DSL constructs constitute independent and reusable DSL subsystem.

3.2 Architecture Overview

DSL allows the user to describe the code, which will be executed on targets. CodeGen module provides a simplified interface to LLVM compilation and optimization facilities. CodeLoader, Execution and Connection modules let user load code on targets, communicate with them (for example, using global variables) and control the code execution. Management of the target's memory is provided by the host through MemoryManager module.

Fig. 1 shows the structure of the system.

This architecture has a benefit of simple extensibility. Each of the following parts of the library can be extended independently from others:

- DSL constructs and operations (for example, support array slicing or exponentiation at the language level);
- communication protocols;
- target runtime functionality;
- most importantly, target platforms.

For details on these points, the reader can proceed to the following sections.

4. DSL

4.1 Design

The core of this library is a powerful embedded C-like DSL. It is translated to LLVM Intermediate Representation (IR) to allow code compilation for a wide range of targets supported by LLVM. This design of the DSL as translated and compiled at runtime is directly motivated by the concept of generative (or multi-stage) programming when the abstraction power of high-level languages is used to compose pieces of low-level code [4]. It makes runtime code generation and domain-specific optimization a fundamental part of the program logic.

As authors of [4] note, the usual appeal of DSLs is in increasing productivity by providing a higher level, more intuitive programming model for domain experts, who are not necessarily expert programmers (”user-facing” DSLs). The other direction, which is of interest for us in this paper, is in using DSL as a means for exposing knowledge about high level program structures to a compiler.

This DSL implementation makes heavy use of powerful template metaprogramming capabilities of C++, up to C++17 standard. The idea to leverage C++ templates to cope with challenges that poses development of DSLs aimed at generative programming goes back at least to the work of Czarnecki et al. [9].

4.2 Description and Examples

DSL provides all necessary language constructs with a familiar syntax:

- basic types (possibly cv-qualified):
  - arithmetic types;
  - pointers;
  - arrays of fixed length (possibly nested);
  - structs (possibly nested);
- operations:
  - arithmetic operators (with the support of pointer arithmetic);
  - logical operators;
  - bitwise operators;
  - C-like cast;
- control flow expressions:
  - sequential (comma operator expression);
  - conditional (if-else expression);
  - while loop;
- functions (with a fixed number of arguments; no recursion);
- literal values.

It is also easily extensible with other higher-level constructs (for example, Python-like array slicing) which will be translated directly to LLVM IR (i.e. will be efficient).

To allow simpler organization of the language, every DSL construct models either value or expression; there are no statements. For example, to return void from a function user needs to use special DSL construct ’`Unit`’. Loops naturally return value from their last cycle. If loop did not run it returns default-initialized value (generally, zero-initialized).

Any DSL construct has a corresponding underlying C++ type, which determines allowed operations on it and conversions to other types. Underlying C++ type can be accessed through member type alias `::type` which is present in every DSL type. And the DSL value type can be obtained (if there is one) from C++ type using `to_dsl<T>` type trait. In other words, there is a direct mapping between DSL types and C++ types. Type trait `to_dsl<T>` can be used as a convenient type factory.

Type of the DSL constructs (real C++ type, not the underlying C++ type) encodes how it was constructed and what child DSL constructs constitute it (for example see listing 1).

```
Var<int> x, y, z;
auto expr = (x + y) * z;
using expr_type =
  EBinOp< Instruction::FMul,
          EBinOp< Instruction::Add,
                  Var<int>,
                  Var<int>
          >,
          Var<int>
  >;
```

Listing 1. Type of some DSL expression

One of the most interesting features of the DSL is a separation of DSL abstract syntax tree (AST) construction from DSL function instantiation. It is achieved through the use of C++14 generic lambdas which play a role of DSL code generators (AST builders). Example can be seen on the next listing.

```
auto max_gen = [](auto x, auto y) {
    return If(x > y, x, y);
};
auto dsl_max = make_dsl_fun<int, int>(max_gen);
```

It allows simple and effective reuse of needed DSL constructs, as in the next example.

```
auto max3_gen = [&](auto x1, auto x2, auto x3) {
    return max_gen(x1, max_gen(x2, x3));
};
auto dsl_max3 = make_dsl_fun<int, int, int>(max3_gen);
```

This conceptually differs from simple function call as a means of code reuse and is closer to function inlining. In this way the new DSL generator is constructed which, in its turn, can be later reused. Moreover, on the point of DSL code generation user can utilize C++ constructs to build more complex DSL expressions (Listing 2).

```
// note: accepts arbitrary DSL expressions
auto reduce_sum_gen = [](auto ...xs) {
    // Using C++17 fold expression
    return (... + xs);
};
auto sum3 = make_dsl_fun<float, double, int>(reduce_sum_gen);
```

Listing 2. Use C++ code to build complex DSL expressions.

```
// note: accepts arbitrary DSL expressions
//  (e.g. other generators)
auto get_reducer = [](const auto& binary_op) {
    return [&](auto x1, auto... xs) {
        // Using C++17 fold expression
        return ( (x1 = binary_op(x1, xs)), ... );
        // Redundant assignments
        //  will be optimized out by LLVM
    };
};
auto max_vararg_gen = get_reducer(max_gen);
auto max3 = make_dsl_fun<int, int, int>(max_vararg_gen);
```

Listing 3. Generator of DSL reduce function over arbitrary DSL expressions.

Listing 4 shows two noticeable syntactic features of the DSL: the sequential operator that plays a role of C/C++ semicolon and DSL local variables. Generally, any DSL variable which is not an argument of DSL generator (enclosing lambda) will be considered a local one. For the more consistent syntax user can define local variables inside the generator lambdas. Also, note that they can't be defined inside the DSL expressions because they follow the rules of C++ expressions. To use global variables a user is required to first load them on the target because they are translated to LLVM IR as actual memory addresses.

```
Var<int> local1;
 // note lambda capture (can also be [&])
auto max_gen = [=](auto arg) {
    Var<int> local2;
    return (
        // variables can't be defined here!
        local1 += arg,
        local1 += local2,
        arg // last expression is returned
    );
};
```

Listing 4. Use of comma operator and local variables.

The next listing demonstrates that DSL allows to construct complex expressions in familiar, close to C, syntax.

```
auto complex_expr = [](Ptr<Var<uint32_t>> ptr) {
    Var<uint32_t> tmp;
    return tmp = *ptr &= ~(*++ptr ^ Lit(1 << 8));
};
```

Generic DSL functions is another very useful feature. As can be seen from previous examples, DSL generators are not bound to specific types of parameters. Instead of explicit manual instantiation of DSL function with required types of parameters library user can instantiate generic DSL function with a help of function factory. If generic function is used with arguments of inappropriate types, compiler will catch this and compilation will fail with comprehensible error message.

Instantiated generic functions are stored in a function repository by a key which represents their type. As a type of DSL constructs encodes their AST, type of DSL functions encodes their body. Thus, the structural equivalence between functions is achieved without any overhead. Thanks to this repeated instantiation of the (structurally) same DSL functions is avoided. DSL function is deleted from the repository at the end of translation to LLVM IR. Needless to say, all this happens behind the scenes and a user isn't required to know about these details.

Listing 5 shows an example of the use of a generic DSL function.

```
auto generic_max = make_generic_dsl_fun(max_gen);

auto max4_gen = [&](auto x1, auto x2, auto x3, auto x4) {
    return generic_max(
            generic_max(x1, x2),
            Cast<float>(generic_max(x3, x4))
    );
};
// This will cause instantiation of 2 max functions:
//  for ints and for floats
auto max4 = make_dsl_fun<float, float, int, int>(max4_gen);
```

Listing 5. Generic DSL function example

Last, but not the least, DSL is designed with usability in mind. C++ code with a heavy use of templates is known for its complex error message on compilation failure. In DSL all major type constraints are checked with static assert standard library function which produces comprehensible compile time error messages.

5. Subsystems Description

5.1 MemoryManager

This centralized memory management organization allows to free less powerful targets from extra tasks and avoid extra communication cycles which would be inevitable to ensure correct memory allocation if targets managed their memory themselves. Best-fit, worst-fit and first-fit memory management algorithms are implemented. Conceptually MemoryManager is part of a CodeLoader and used only for data and code loading. That is, it's important to note that target code can't dynamically allocate memory on targets.

5.2 CodeLoader

With the help of CodeLoader module user can load DSL global variables and compiled code on targets. CodeLoader also allows getting a handle to already loaded variables and functions. In this case, no checks or memory allocation is performed, because, in general, there is no possibility to ensure correctness of user's actions. For example, functions can be loaded on a target in a persistent memory in one program run, and on another program run any knowledge about it will be lost, whereas the user may want to access previously loaded data and functions. So, it is assumed that user knows what he is doing.

5.3 Connection Module: Host side

Connection module consists of two parts: command protocol for communication between host and targets and underlying connection implementation. The functionality of the former is fully built on the primitives of the latter, which must provide synchronous read and write operations.

The core command protocol includes the following commands:

- echo (for testing);
- read specified number of bytes at a specified address;
- write data to a specified address;
- call function at the specified address (without arguments and return value);
- set function at the specified address on execution by the timer;
- set function at the specified address on execution on the specific interrupt.

This abstraction from specific implementation allows easier extensibility on new connection protocols. This work implements connection through TCP and through USB (used as a virtual serial port).

5.4 Connection Module: Target Runtime API

Each specific target platform requires its own firmware to interface with the host. It must provide functionality for communicating with the host and answering to requests according to the command protocol.

At this point an important consideration arises: targets must provide API sufficient for a wide range of tasks. Generally, peripheral devices on microcontrollers are memory mapped, which means that runtime API consisting of memory read and write functions can be sufficient. For example, the family of STM32 microcontrollers has fixed memory map and each device has a specific predefined address in memory.

Some platforms may need an extended API. When the target has an operating system, in particular Linux, it can additionally provide an interface to some of the system calls: `open()` for using devices represented as input/output ports and `mmap()` for correct work with library runtime process address space. It is implemented in the LinuxConnection module. Although for this platform it is also possible to implement an interface to arbitrary system calls and libraries using `dlopen()` and `dlsym()` functionality, the library runtime API for Linux is intentionally left minimal but sufficient for tasks concerned with controlling peripheral devices.

Another important question is a debugging interface. Issuing diagnostic messages to some local to target buffer can accomodate most of the needs and at the same time is easily implementable. Target must provide interface to read the buffer and to get an address of the target local logging function. This address is used to construct the DSL wrapper for remote logging function. From this point it can be further used in the DSL code.

6. Approbation

The system was tested on several setups:

- Linux on x86 plays the role of both host and target machines, communication is through TCP connection (setup for tests during development);
- the host is Linux x86, the target is Odroid XU4 (armv7a) with Linux, TCP connection;
- the host is Linux x86, the target is bare-bones stm32f429i-discovery microcontroller (armv7em), USB Virtual COM Port connection;
- the host is Odroid XU4 (armv7a) with Linux, the target is bare-bones stm32f429i-discovery (armv7em), connection through USB Virtual COM Port.

Tests were performed for each command from the command protocol (see above in the section 5.3).

7. Demonstration

For a demonstration of dynamic optimization possibilities, which this library opens, the reader can refer to the following listings of PID control (listing 6) and its tuning (listing 7) for specific conditions of the deployment environment.

```
using namespace hetarch;
using namespace hetarch::dsl;

typedef int32_t ctrl_t; // for control variables
typedef float coef_t; // for coefficients

// Example of the target
typedef uint32_t addr_t; // size_t of the target
conn::SerialConnImpl<addr_t> conn{"/dev/ttyACM0"};
SimplePipeline<addr_t> pipeline{"armv7e_linux_eabihf", conn};

// Global var-s to store error data between control cycles
auto perr = pipeline.load(Global{ Var<ctrl_t>{0} });
auto ierr = pipeline.load(Global{ Var<ctrl_t>{0} });

// dt -- control cycle durations (in seconds)
// sp -- setpoint
auto pid_gen = [&](auto Kp,auto Ki,auto Kd,auto dt,auto sp) {
  auto pid_ctrl = [&]{
    // Local variables:
    // pv -- process variable
    // cv -- control variable
    Var<ctrl_t> pv, cv, prev_perr, derr;

    // read_pv and write_cv are some dsl generators
    //  that perform actual input/output
    return (
      pv = read_pv(),

      prev_perr = perr,
      perr = sp - pv,
      ierr += perr,
      derr = perr - prev_perr,
      cv = Kp*perr + Kd*derr/dt + Ki*ierr*dt;

      write_cv(cv)
    );
  };
  return pid_ctrl;
};
```

Listing 6. PID controller DSL code.

```
auto tuner = [&](auto dt, auto sp){
  // For tuning coefficients are usual mutable DSL variables
  Var<coef_t> Kp{0}, Ki{0}, Kd{0};
  auto pid_ctrl = pid_gen(Kp, Ki, Kd, dt, sp);

  // Specific tuning method:
  //  determines current operating conditions
  //  (e.g. by reading some sensors)
  //  and returns tuning data that allows to compute
  //  optimal PID controller coefficients.
  // E.g. for Ziegler-Nichols method it is
  //  Ku -- "ultimate gain" and Tu -- oscillation period
  return (/* actual tuning code goes here */);
};

// Example parameters
Lit sp{42}; // Setpoint
int ms_delay{100}; // Control cycle duration
Lit dt{ms_delay / 1000.0};

auto tuning_code = make_dsl_fun(tuner, dt, sp);
// Translate, compile and load tuning code
auto tuning_fun = pipeline.load(tuning_code);
// Run tuning code and get tuning data
auto tuning_data = exec.call(tuning_fun, dt, sp);
// Compute coefficients using optimal tuning data
auto [Kp, Ki, Kd] = compute_coefs(tuning_data);

// Generate optimal PID controller
auto opt_pid_gen = pid_gen(Kp, Ki, Kd, dt, sp);
auto opt_pid_code = make_dsl_fun(opt_pid_gen);
// Translate, compile and load optimal PID controller
auto opt_pid = pipeline.load(opt_pid_dsl);

// Finally, run PID controller on timer
pipeline.schedule(opt_pid.callAddr, ms_delay);
```

Listing 7. PID tuning DSL code.

The work is organized in the following way:

- in the first phase host loads general version of the PID controller with tuning code on the target;
- in the second phase tuning code is called and its result is read by host;
- in the third phase host computes coefficients based on tuning data and recompiles PID controller with them;
- finally, host loads PID controller optimized for specific coefficients.

This example shows two advantages of using the library. Firstly, tuning code is completely absent from the final program running on the target. Dynamic code generation allows compiling code for specific constant coefficients to achieve better execution times and smaller program size.

Secondly, the dynamically generated code can be more optimal due to optimizations performed by LLVM. When coefficients are integer values, or, even better, integer powers of two (or float values, that can be rounded without big errors), resulting code will be generated with fewer (or completely without) expensive floating operations.

```
typedef int ctrl_t;
typedef float coef_t;

extern coef_t Kp, Kd, Ki;
ctrl_t perr = 0, ierr = 0;

ctrl_t pid_ctrl(float dt, ctrl_t sp, ctrl_t pv) {
  ctrl_t prev_perr = perr;
  perr = sp - pv;
  ierr += perr;
  ctrl_t derr = perr – prev_perr;

  return Kp*perr + (Kd*derr/dt) + (Ki*ierr*dt);
}
```

Listing 8. PID controller C code used for LLVM IR comparison.

To emphasize possible dynamic optimizations, fig. 2 presents a comparison between listings of the PID controller code for two cases:

- C code from listing 8 compiled with clang without this library;
- DSL code from listing 6 dynamically optimized with this library.

```
1.  ; Kp * perr
2.  %9 = load float, float* @Kp
3.  %10 = sitofp i32 %perr to float
4.  %11 = fmul float %9, %10
5.
6.  ; Kd * derr / dt
7.  %12 = load float, float* @Kd
8.  %13 = sitofp i32 %derr to float
9.  %14 = fmul float %12, %13
10. %15 = fdiv float %14, %dt
11.
12. %16 = fadd float %11, %15
13.
14. ; Ki * ierr * dt
15. %17 = load float, float* @Ki
16. %18 = sitofp i32 %ierr to float
17. %19 = fmul float %17, %18
18. %20 = fmul float %19, %dt
19.
20. %21 = fadd float %16, %20
```

```
1.
2.
3.  ; Kp * perr
4.  %12 = shl i32 %perr, 2
5.  %13 = sitofp i32 %12 to float
6.
7.  ; Kd * derr / dt
8.  %14 = sitofp i32 %derr to float
9.  %15 = fmul float %14, 5.000000e-01
10. %16 = fdiv float %15, 1.000000e-01
11.
12. %17 = fadd float %16, %13
13.
14.
15. ; Ki * ierr * dt
16. %18 = mul i32 %ierr, 6
17. %19 = sitofp i32 %18 to float
18. %20 = fmul float %19, 1.000000e-01
19.
20. %21 = fadd float %20, %17
```

Fig. 2. Comparison of LLVM IR generated for expression "`Kp*perr + (Kd*derr/dt) + (Ki*ierr*dt)`*" (core part of the PID controller code; other lines are omitted here). Compiler options used:* `-O2 -target x86_64-pc-linux-gnu`*. LLVM IR is used instead of native assembler because it is more readable and optimizations are done on the IR. Left: compiled with clang from C code on list. 8. LLVM IR is presented only for the last line. Right: compiled with LLVM from DSL (see list. 6). For the sake of demonstration it is assumed that dynamically determined PID controller coefficients are* `Kp=4, Kd=6, Ki=0.5`*; and control cycle duration is* `dt=0.1`*.*

There are several things on the fig. 2 to note:

- dynamically generated code has fewer memory accesses because it is compiled for specific values (note lines 2, 7, 15 where usual code loads coefficients stored as global variables);
- instead of floating-point multiplications (lines 4 and 17 on the left) integer shift (line 4, right) and integer multiplication (line 16, right) are used;
- one apparent to a programmer optimization on line 9, right is missed: substitute multiplication by 0.5 with integer division by 2 or right shift by one; and it should be[2], although it is possible to implement such optimizations on the DSL level.

7.1 Library Applicability

The library is intended for use with embedded heterogeneous systems of a small scale with low-power secondary processors and microcontrollers that run heterogeneous tasks. The case of homogeneous tasks on the more powerful systems is better accomodated with existing tools (e.g. OpenCL or Delite) that are

[2] This compiler behavior is expected according to C11 standard (section F9.2.1), because representations of 0.5 and 2 maybe not be equivalent and the result can be different on some machines.

specifically aimed at scheduling and parallelizing the computations across bigger number of secondary processors. This library is not intended for such use cases and doesn’t provide any orchestration for parallel tasks. Each secondary processor should be managed manually and separately.

Generally, the benefits and applicability of the library should be considered in each particular case. As noted in the introduction, the library is well suited for the problems when the dynamic configuration of the system is required (either for particular environment conditions or for different peripheral devices and sensors). It’s also important to consider the price of dynamic recompilation: the benefits of the specialized and optimized code should amortize the compilation price.

8. Conclusion

This work presented a powerful DSL language aimed at metaprogramming and showed its application to the domain of heterogeneous embedded systems. Although the library misses some features (as noted in Further Work section), it constitutes a proof of concept that the idea of dynamic code generation is perspective and useful in the real-world scenarios

9. Further Work

The work can be continued in several directions.

The library does not provide facilities for loading on the targets existing compiled code, for example, libraries. To be applicable to a wider range of use cases it requires support of this functionality.

The development of the DSL is another direction. It can be extended with additional language constructs, for example, switch, goto or to support recursion. It can also be further developed to include more features of functional programming languages, e.g. functions as first-class citizens. Support for a debugging in terms of the DSL (breakpoints, tracing) can also be added.

References

[1]. K. Melentev, R. Belkov, and I. Kirilenko. The programming system for cybernetic heterogeneous architectures using LLVM. In Proc. of the Second Conference on Software Engineering and Information Management (SEIM-2017), 2017, pp. 31-35 (in Russian).

[2]. J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, vol. 12, no. 3, , 2010, pp. 66–73.

[3]. M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. IEEE micro, vol. 28, no. 4, 2008.

[4]. T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti et al. Go meta! A case for generative programming

and DSLs in performance critical systems. In LIPIcs-Leibniz International Proceedings in Informatics, vol. 32, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[5]. K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In Proc. of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2011, pp. 89–100.

[6]. A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Transactions on Embedded Computing Systems (TECS), vol. 13, no. 4s, 2014, article no. 134.

[7]. T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. Communications of the ACM, vol. 55, no. 6, 2012, pp. 121–130.

[8]. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04), Mar 2004, pp. 75-86.

[9]. K. Czarnecki, J. T. O’Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Domain-Specific Program Generation, LNCS, vol. 3016, 2003, pp. 51-72.

Библиотека программирования гетерогенных архитектур

Г.В.Киргизов <gkirgizov@gmail.com>
Я.А.Кириленко <y.kirilenko@spbu.ru>
Кафедра системного программирования,
Санкт-Петербургский государственный университет,
199034, Россия, г. Санкт-Петербург, Университетская набережная, 7

Аннотация. Встраиваемые системы с гетерогенной архитектурой, рассматриваемые в данной работе, состоят из одного управляющего и одного или нескольких периферийных процессоров. Разработка ПО для таких систем представляет заметные сложности, требуя различные наборы инструментов для каждой составляющей гетерогенной системы. Достижение высокой эффективности также становится более сложной задачей. Кроме того, во многих сценариях встраиваемые системы требуют настройки во время исполнения, что непросто обеспечить с использованием стандартных средств. Эта работа представляет C-подобный предметно-ориентированный язык (DSL) для метапрограммирования и библиотеку, предоставляющую единый интерфейс для программирования периферийных процессоров с использованием этого языка. Это позволяет разрешить упомянутые проблемы. DSL встроен в C++ и позволяет свободно манипулировать написанными на нем выражениями и, таким образом, представляет собой реализацию идеи генеративного программирования, когда выразительная мощь высокоуровневого языка используется для многоступенчатой генерации низкоуровневого DSL кода. Вместе с другими возможностями, например, обобщенными DSL функциями, это делает данный язык гибким инструментом для динамической кодогенерации. Подход, используемый в библиотеке, — это динамическая компиляция. Код, написанный на предметно-ориентированном языке, транслируется в LLVM IR и затем компилируется в машинный код во время исполнения. Это открывает возможность динамических оптимизаций кода, например, специализации функций для определенных значений, известных только во время исполнения. Гибкая архитектура библиотеки обеспечивает простую расширяемость на любые платформы, поддерживаемые LLVM. В конце работы также приводятся апробация библиотеки на нескольких системах и демонстрация возможности динамических оптимизаций.

Ключевые слова: метапрограммирование; кодогенерация; встроенный DSL; гетерогенные системы; встроенные системы.

DOI: 10.15514/ISPRAS-2018-30(4)-3

Для цитирования: Киргизов Г.В., Кириленко Я.А. Библиотека программирования гетерогенных архитектур. Труды ИСП РАН, том 30, вып. 4, 2018 г., стр. 45-62 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-3

Список литературы

[1]. К. Мелентьев, Р. Белков и Я. Кириленко. Система программирования кибернетических гетерогенных архитектур с использованием LLVM. Second Conference on Software Engineering and Information Management (SEIM-2017) (short papers), 2017, стр. 31-35.

[2]. J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, vol. 12, no. 3, , 2010, pp. 66–73.

[3]. M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. IEEE micro, vol. 28, no. 4, 2008.

[4]. T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti et al. Go meta! A case for generative programming and DSLs in performance critical systems. In LIPIcs-Leibniz International Proceedings in Informatics, vol. 32, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[5]. K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In Proc. of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2011, pp. 89–100.

[6]. A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Transactions on Embedded Computing Systems (TECS), vol. 13, no. 4s, 2014, article no. 134.

[7]. T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. Communications of the ACM, vol. 55, no. 6, 2012, pp. 121–130.

[8]. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04), Mar 2004, pp. 75-86.

[9]. K. Czarnecki, J. T. O’Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Domain-Specific Program Generation, LNCS, vol. 3016, 2003, pp. 51-72.