Санкт-Петербургский Государственный Университет

Математическое обеспечение и администрирование информационных сетей
Кафедра системного программирования

Киргизов Григорий Валерьевич

Библиотека программирования гетерогенных встраиваемых архитектур

Бакалаврская работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Рецензент:
инженер-программист ООО "Интеллиджей Лабс"
Д. А. Мордвинов

Санкт-Петербург
2018

SAINT PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering Department

Grigorii Kirgizov

Programming Library for Heterogeneous Embedded Architectures

Graduation Thesis

Scientific supervisor:
senior lecturer Iakov Kirilenko

Reviewer:
software engineer OOO «IntelliJ Labs»
Dmitrii Mordvinov

Saint Petersburg
2018

Contents

Introduction **5**

1. Problem Statement **7**

2. Similar Work **8**

2.1. OpenCL & CUDA 8
2.2. Texas Instruments MultiCore SDK 8
2.3. Delite 9
2.4. LLVM ORC 9

3. Discussion of the Previous Work **11**

3.1. Inherited Decisions 11
3.2. Shortcomings 12

4. System Architecture **14**

5. DSL **16**

5.1. Design 16
5.2. Description and Examples 16
5.2.1. Types 17
5.2.2. Generative Programming 18
5.2.3. Comma Operator and Variables 19
5.2.4. Generic Functions 20
5.2.5. Error Messages 21
5.3. Implementation Details 21
5.3.1. Generic Functions 21
5.3.2. Static Dispatch in Translation 22
5.3.3. Dependent Behavior of DSL Constructs 23

6. Subsystems description **25**

6.1. MemoryManager 25
6.2. CodeLoader 25
6.3. Connection Module: Host 25

6.4. Connection Module: Target Runtime 26

7. Approbation 27

8. Demonstration 28

9. Discussion 30

9.1. Library Applicability 30

9.2. Limitations 30

Conclusion 31

Appendix 32

References 36

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 which 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 the primary 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).

Another problem is a dynamic configuration 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. But it may not be possible due to nature of the task or too often or rapid (for manual operating) changes of the environment. A variation of this scenario is a runtime configuration for specific peripheral devices (e.g. different models of sensors and actuators).

This work is based on preliminary results of [6].

1 Problem Statement

The goal of this work is to design and implement a system for programming heterogeneous embedded platforms that provides a possibility of dynamic configuration and simplifies software development by proposing a unified programming model.

This goal has been decomposed on the following tasks:

- Conduct a survey.
- Analyze existing system implementation.
- Redesign the system.
- Implement according to the new architecture:
  - independent DSL subsystem;
  - embedded architectures programming library.
- Conduct a system approbation.

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're several conceptual approaches to aforementioned difficulties.

2.1 OpenCL & CUDA

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) [10] and CUDA framework from Nvidia [7]. Both these frameworks propose a 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.

2.2 Texas Instruments MultiCore SDK

Multicore Software Development Kit [9]—is a platform that simplifies programming for heterogeneous systems of TI Keystone architecture (ARM+DSP kind of systems). It helps developers by providing common reusable components, support for SYS/BIOS and LHOS (Linux High Level

OS), API for programming DSP, integration with IDE Code Composer Studio and through other means.

Although it is a very helpful tool, that offers a convenient programming environment, it is aimed only at the specific family of heterogeneous systems and doesn't solve the problems of dynamic configuration.

2.3 Delite

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 [3].

Among their examples Delite—a heterogeneous parallel framework for domain-specific languages [2, 11]—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) [8] 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 Delite framework and the presented system start from the same idea of multi-stage programming, they significantly differ in the approaches and application domains. Most importantly, presented system uses 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.

2.4 LLVM ORC

LLVM ORC (On-Request Compiler) [4]—is a subsystem of the LLVM framework [5] for Just-In-Time (JIT) compilation. Thanks to separation between the notions of a JIT-server and a JIT-client it allows to organize remote JIT compilation on heterogeneous platforms.

In principle, it can help to solve the problems of dynamic configuration and optimizations. But it is not suitable for embedded systems because, in general, peripheral processors and microcontrollers are low-powerful devices and can't run a full-fledged JIT server with its dependencies. Moreover, they don't need most of its functionality. So, LLVM ORC subsystem is seen as too heavy-weight and its functionality is excessive for embedded systems.

3 Discussion of the Previous Work

Previous work [6] presented a prototype of the library that showed the viability of the idea of dynamic code generation for secondary processors of heterogeneous system.

3.1 Inherited Decisions

The following decisions have shown themselves as reasonable and grounded and thus are inherited from the previous work. 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's more demanding to maintain the compiler extension to keep it up-to-date with the needed compiler versions. And it's 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 [5] 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.

3.2 Shortcomings

Current implementation has several shortcomings that complicate further development of its ideas; the most notable problem is DSL design.

- No distinction is made between the stages of work with a DSL code:
  - DSL code is tied up to the specific platform bit width already on the stage of definition, which is clearly not necessary;
  - the same module (Function) is used to represent both loaded and unloaded code—a user must remember what functions are already loaded or check it every time to avoid mistakes;
  - DSL Function addresses (i.e. location in the target memory) must be fixed prior to the compilation, although it can be handled further at the code loading stage—it prevents a reuse of the already compiled functions and thus brings extra compilation overhead.
- Ineffective and inconvenient translation to LLVM IR—instruments provided by the LLVM framework (e.g. IRBuilder class) are ignored and manual string concatenation is used instead. It is more error-prone and difficult to extend. Moreover, generation of a correct IR code is essential for the work of LLVM optimization passes.
- Code compilation for secondary processors currently doesn't involve any optimizations.
- Connection module has no separation between the handling of the library command protocol (by which host and targets communicate) and a specific communication protocol (e.g. TCP or I2C); this also complicates system extensibility (i.e. to new communication protocols).

This work revises previous architectural choices, redesigns and reimplements the library and substantially extends it in terms of functionality and possible applications. In particular, the new DSL is completely abstracted from other parts of the library and can be reused independently in other projects based on the idea of metaprogramming. Most importantly, the new DSL implementation opens a possibility of dynamic optimizations.

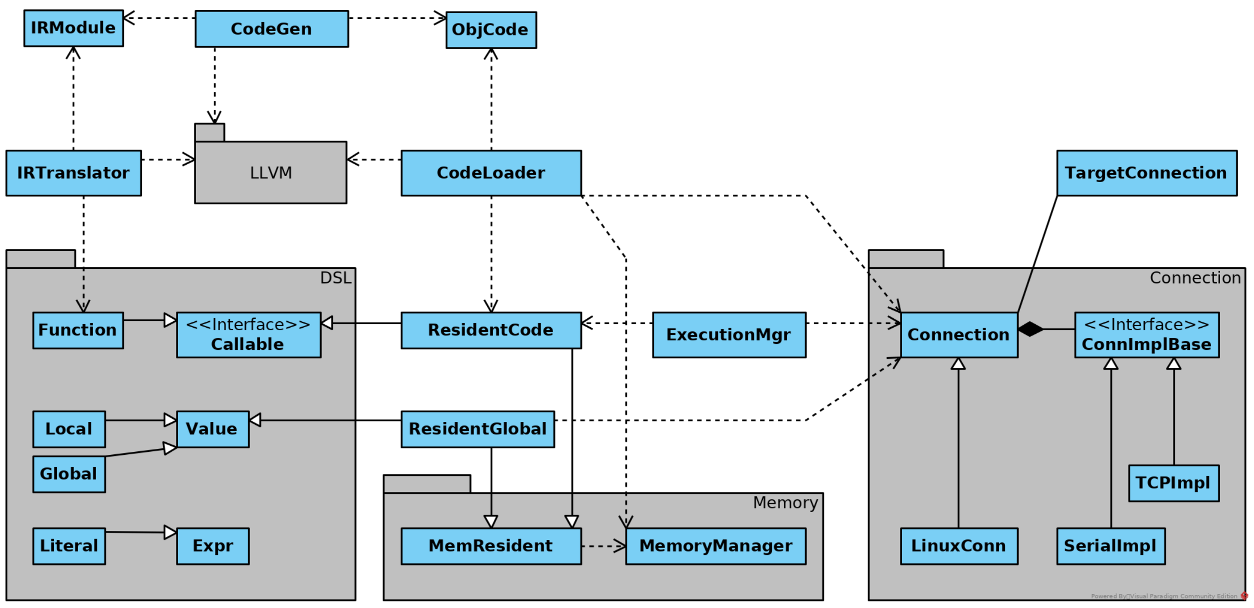

Figure 1: UML class diagram of the system. DSL class hierarchy is shown only approximately because of its breadth and templated, dynamic nature. IRTranslator together with non-resident DSL constructs constitute independent and reusable DSL subsystem.

4 System Architecture

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

Fig. 1 shows the new architecture of the system.

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

This architecture has a benefit of simple extensibility. Each of the following parts of the library can be extended independently from the 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, a reader can proceed to the following sections.

5 DSL

5.1 Design

The main part 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 [3]. It makes runtime code generation and domain-specific optimization a fundamental part of the program logic.

As authors of [3] 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 direct interest for this work, 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. [1].

5.2 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 didn't run it returns default-initialized value (generally, zero-initialized).

5.2.1 Types

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

5.2.2 Generative Programming

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 2.

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

Listing 2: Basic definition of a DSL generator and a DSL function

It allows simple and effective reuse of needed DSL constructs, as in the next example on listing 3.

```
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);
```

Listing 3: Basic reuse of the previously defined DSL generator

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 stage of DSL code generation user can utilize C++ constructs to build DSL code (for examples see listings 4 and 5). So, although DSL provides only basic programming constructs, a user of DSL is not restricted only to them and can use expressive high-level C++ constructs to build a low-level DSL code. It is essentially a realization of the idea of generative programming.

```
// 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 4: 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 5: Generator of DSL reduce function over arbitrary DSL expressions.

5.2.3 Comma Operator and Variables

Listing 6 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 6: Use of comma operator and local variables.

5.2.4 Generic Functions

Generic DSL functions is another very useful feature. As can be seen from the 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.

The following listing 7 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 7: Generic DSL function definition and use

5.2.5 Error Messages

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.

5.3 Implementation Details

5.3.1 Generic Functions

Generic DSL function is implemented as a C++14 generic lambda with a variable number of arguments. The function `make_generic_dsl_fun` is a factory of these lambdas; it accepts DSL generator as its argument. The lambda captures DSL generator by copy to avoid subtle problems with its lifetime management. Otherwise user would be required to watch that DSL generators are accessible during the calls to the corresponding generic DSL functions.

On the call such lambda accesses the template function repository to get the reference to the instantiation of the generic function for the specific types of input arguments that were provided. This instantiation represents a usual `dsl::Function`. On this access either new instantiation is added to the repository or the existing one is returned. If the generic DSL function cannot be called with the provided arguments then the compilation will fail at this moment. Then, when the instantiation of the generic function is obtained, the expression representing a call to it (`ECall`) is returned.

Overall, from the user point, it is similar to the call to a C++ template function with an automatic template argument type deduction from the function input arguments.

5.3.2 Static Dispatch in Translation

Every DSL construct is translated differently. At the same time they must provide uniform interface for translation (generally, a single translation method `toIR()`). It can be achieved by one of the two ways:

- Dynamic dispatch through virtual translation method overloaded in each DSL construct.
- Static dispatch through templated translation method.

Static dispatch has several advantages because it allows to avoid the overhead of virtual function calls[1]:

- Extra indirection (pointer dereference) for each call to a virtual method.
- Virtual methods usually can't be inlined, which may be a significant cost hit for some small methods.
- Additional pointer per object. On 64-bit systems, this is 8 bytes per object. For small objects that carry little data this may be a serious overhead.

[1]`https://waybackmachine.org/web/20171001124440/http://eli.thegreenplace.net:80/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c` (Accessed: May 29, 2018)

The problem of translation may benefit from this even more, because it involves a big number of small objects (DSL constructs). These efficiency considerations can be especially important for embedded platforms.

Translation happens entirely in IRTranslator class, that maintains the state of translation (e.g. stack frames and id-to-value mappings). All that method `toIR()` in any DSL construct does is accepting `IRTranslator` by reference and passing that DSL construct to method `IRTranslator::accept` for translation. Correct `accept` overload is chosen based on the type of argument (i.e. type of the DSL construct). As this function is the same for any DSL construct, it is implemented as macro `IR_TRANSLATABLE` (see listing 8).

```
#define IR_TRANSLATABLE \
inline void toIR(IRTranslator &irt) const { \
    irt.accept(*this); \
}
```

Listing 8: IR_TRANSLATABLE macro

5.3.3 Dependent Behavior of DSL Constructs

Each DSL expression represents description of a computation rather than computation itself. For example, dereferencing a `Ptr<T>` produces not a object of type T, but rather *dereference expression* `EDeref<Ptr<T> >` representing dereference operation. At the same time it must behave in the DSL computations exactly as an object of type T.

The problem is that T is an arbitrary type—it can be a simple arithmetic value, as well as a complex expression involving function calls and pointer arithmetic. This behavior is determined by a set of non-member (e.g. binary operators) and member functions (e.g. dereferencing, address-of and assignment operators).

For non-member functions the problem takes a slightly different form: although they are template functions, the set of correct template parameters is constrained. It can be achieved with the SFINAE C++ idiom (*Substitution Failure Is Not An Error*) through the use of `std::enable_if`.

For member functions the problem is more interesting. Usual inheritance (i.e. dynamic polymorphism) can't be used here, because for the correct

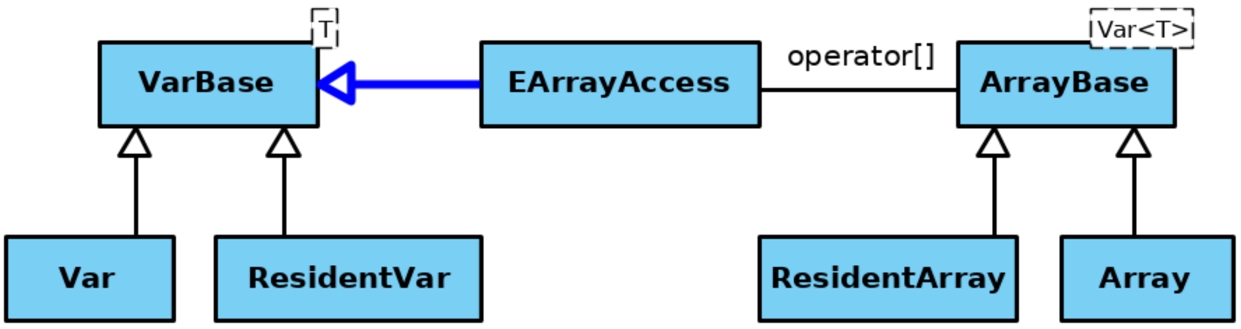

Figure 2: UML class diagram showing special mechanism of inheritance that some DSL constructs utilize.
Note template parameter `Var<T>` for `ArrayBase`—the array element type. `Var<T>` itself depends on template parameter `T`—one of the basic arithmetic types. It determines what behavior acquires `EArrayAccess`; that is, what it inherits from (as shown by the blue arrow).
It's also important to note that each of the classes in this hierarchy inherits from its own base class (due to CRTP), and they're shown in the same hierarchy only for presentation purposes.

translation all types must be known at the compile time (as explained in the section 5.3.2). The solution is, essentially, an elaborated form of the so-called *Curiously Recurring Template Patter (CRTP)* C++ idiom. It involves inheritance of expressions such as `EDeref<Ptr<T> >` from the type, that specifies behavior for the type T and that is parameterized by the type of this expression, thus preserving it. «The type, that specifies behavior for the type T» can be T itself or one of its CRTP ancestors. It can also be seen as an inheritance determined by a template type parameter. Exactly this allows to achieve correct and effective handling of expressions of any complexity (for example, as on the listing 9).

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

Listing 9: DSL allows to construct complex expressions.

The reader can refer to the fig. 2 depicting as an example one simple variant of inheritance of this kind for DSL constructs `Array` (with element type `Var<T>`) and `EArrayAccess`. Listing 10 in the Appendix shows how this kind of inheritance can be implemented in the code.

6 Subsystems description

6.1 MemoryManager

Conceptually MemoryManager is a 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. The 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.

6.2 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.

6.3 Connection Module: Host

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:

- read specified number of bytes at a specified address;
- write data to a specified address;

- call function at the specified address;
- set function at the specified address on execution by the timer;

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).

6.4 Connection Module: Target Runtime

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 accommodate 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.

7 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 6.3). They can be reproduced with a suitable peripheral processor and a suitable runtime running on it. The runtime for targets with a UNIX-like operating system can be found in the project repository[2] (`local_loader` target in CMake build system). The runtime for stm32f429i-discovery microcontroller can be found in a separate repository[3]. Tests are implemented with Google Test framework[4].

[2] `https://github.com/gkirgizov/hetarch`

[3] `https://github.com/gkirgizov/hetarch_stm32f4`

[4] `https://github.com/google/googletest`

8 Demonstration

For a demonstration of dynamic optimization possibilities, which this library opens, a reader can refer to the listings of PID control (list. 11) and its tuning for specific conditions of the deployment environment (list. 12) in the Appendix.

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 data produced by it 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.

To emphasize possible dynamic optimizations, fig. 3 presents a comparison between listings of the PID controller code for two cases:

- C code from listing 13 compiled with clang without this library;
- DSL code from listing 11 dynamically optimized with this library.

There're several things on fig. 3 to note:

```
 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
```

Figure 3: 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.
Left: compiled with clang from C code on list. 13. LLVM IR is showed for the last line.
Right: compiled with LLVM from DSL (see list. 11). 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`.

- 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[5], although it is possible to implement such optimizations on the DSL level.

[5]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.

9 Discussion

9.1 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 accommodated with existing tools (e.g. OpenCL or Delite) that are specifically aimed at scheduling and parallelizing the computations across bigger number of secondary processors. This library isn't 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.

9.2 Limitations

The library has some limitations.

It doesn't provide facilities for loading on the targets existing compiled code (e.g. other programming libraries). To be applicable to a wider range of use cases it requires support of this functionality.

DSL can also be extended with additional language constructs, for example, switch, goto or to support recursion. Support for a debugging in terms of the DSL (breakpoints, tracing) can also be a useful feature.

Conclusion

All tasks have been successfully accomplished:

- Survey with relevant works is presented.
- Previous system implementation is thoroughly analyzed and its problems are discussed.
- System is redesigned:
  - DSL subsystem is abstracted from other parts of the system;
  - additional abstractions are introduced to make the system more extensible and maintainable.
- Implemented according to the new architecture:
  - DSL subsystem;
  - embedded architectures programming library.
- System is tested on several platforms.

Source code with build instructions can be found in the project repository[6].

[6] `https://github.com/gkirgizov/hetarch`

Appendix

```cpp
template< typename T, typename TChild >
struct get_base {
    using type = typename T::template base_t<TChild>;
};
template< typename T, typename TChild >
using get_base_t = typename get_base<T, TChild>::type;

// EArrayAccess must behave exactly as an array element
template< typename TArray, typename TIndex >
class EArrayAccess :
    // get the type providing behavior for array element
    // and parametrize it by EArrayAccess (CRTP idiom)
    public get_base_t<
        typename TArray::dsl_element_t,
        EArrayAccess<TArray, TIndex>
    >
{ /* ... */ }

// ArrayBase type provides array-like behavior
//  e.g. operator[] (that returns EArrayAccess)
template< typename TArray, typename TElem, std::size_t N >
class ArrayBase {
public:
    using dsl_element_t = TElem;
    template<typename TChild>
    using base_t = ArrayBase<TChild, TElem, N>;
    // ...
}

// VarBase type provides behavior of basic arithmetic types
template< typename TVar, typename T >
class VarBase {
    template<typename TChild>
    using base_t = VarBase<TChild, T>
    // ...
}
```

Listing 10: Example of providing behavior of array element (e.g. VarBase<T>) for EArrayAccess.

```
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 11: 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 12: PID tuning DSL code.

```c
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 13: PID controller C code used for LLVM IR comparison.

References

[1] DSL Implementation in MetaOCaml, Template Haskell, and C++ / Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, Walid Taha // Domain-Specific Program Generation. — 2003.

[2] Delite: A compiler architecture for performance-oriented embedded domain-specific languages / Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee et al. // ACM Transactions on Embedded Computing Systems (TECS). — 2014. — Vol. 13, no. 4s. — P. 134.

[3] Go meta! A case for generative programming and DSLs in performance critical systems / Tiark Rompf, Kevin J Brown, HyoukJoong Lee et al. // LIPIcs-Leibniz International Proceedings in Informatics / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. — Vol. 32. — 2015.

[4] Hames Lang. ORC – LLVM's Next Generation of JIT API. — 2016. — Access mode: `http://llvm.org/devmtg/2016-11/#talk1` (online; accessed: 17-12-2017).

[5] Lattner Chris, Adve Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). — Palo Alto, California, 2004. — Mar.

[6] Melentev Kirill, Belkov Roman, Kirilenko Iakov. Sistema programmirovaniya kiberneticheskih geterogennyh arhitektur s ispolzovaniem LLVM // Second Conference on Software Engineering and Information Management (SEIM-2017)(short papers). — 2017. — P. 31.

[7] Parallel computing experiences with CUDA / Michael Garland, Scott Le Grand, John Nickolls et al. // IEEE micro. — 2008. — Vol. 28, no. 4.

[8] Rompf Tiark, Odersky Martin. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs // Communications of the ACM. — 2012. — Vol. 55, no. 6. — P. 121–130.

[9] White paper: Multicore software development kit : Rep. : SPRY168A / Texas Instruments Incorporated ; Executor: Raghu Nambiath Sanjay Bhal, Raj Sivarajan : 2011. — May. — Access mode: `http://www.ti.com/lit/wp/spry168a/spry168a.pdf`.

[10] Stone John E, Gohara David, Shi Guochun. OpenCL: A parallel programming standard for heterogeneous computing systems // Computing in science & engineering. — 2010. — Vol. 12, no. 3. — P. 66–73.

[11] A heterogeneous parallel framework for domain-specific languages / Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee et al. // Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on / IEEE. — 2011. — P. 89–100.