Exploring the Semantic Gap in Compiling Embedded DSLs

Peter Zangerl, Herbert Jordan, Peter Thoman, Philipp Gschwandtner and Thomas Fahringer
University of Innsbruck, 6020 Innsbruck, Austria
Email: {peterz,herbert,petert,philipp,tf}@dps.uibk.ac.at

Abstract—**Optimizing compilers provide valuable contributions to the quality of processed code. The vast majority of application developers rely on those capabilities to obtain binary code efficiently utilizing processing resources. However, compiler proficiency is frequently misjudged by application developers. While for some constellations the effectiveness of those optimizations is grossly underestimated, for others, mostly involving higher-level semantic concepts of embedded DSLs, the compilers' influence on the code quality tends to disappoint.**

In this paper, we provide examples for the effectiveness and ineffectiveness of state-of-the-art optimizing compilers in improving application code. Based on those observations we characterize the differences between positive and negative examples and provide an in-depth explanation for the shortcomings of optimizing compilers. Furthermore, we present a semantic-aware compiler architecture rectifying those deficiencies and outline several example use cases demonstrating our architecture's ability to contribute to the efficiency and the capabilities of embedded DSLs.

1. Introduction

Domain-specific languages (DSLs) are well-established as an effective way of allowing the efficient and productive programming of algorithms in individual domains, without requiring low-level optimization expertise or hardware knowledge [1]. However, creating new DSLs requires a large initial implementation effort, as well as ongoing support and potentially extension to new target hardware architectures.

Embedded DSLs (EDSLs) circumvent this issue by leveraging the existing community, support network and target platforms of a so-called *host* language, which is usually chosen for being well-established and broadly supported, while offering sufficient flexibility to naturally express the concepts of the domain in question. EDSLs are then implemented as libraries in their host language, allowing them to fully utilize the existing tooling of this host language, including e.g. development environments, debuggers, and — most crucially for our exploration in this paper — highly sophisticated optimizing compilers.

On paper, therefore, EDSLs combine the best of both worlds: offering a highly productive interface to domain experts, without much initial implementation effort, while leveraging the full potential of modern mainstream language toolchains. And to a large extent this holds true in practice, with EDSLs spreading rapidly across many distinct fields of computer science research and applications, from high-performance computing [2] all the the way to embedded systems [3]. In fact, the increasing prevalence and success of EDSLs has in turn influenced programming language design, with mainstream languages adopting features which facilitate more powerful and natural EDSLs, such as meta-programming, higher-order functions and closures, or reflection [4], [5], [6].

However, embedded DSLs present unique challenges to modern optimizing compilers, which may lead practitioners to simultaneously over- and underestimate their optimization capabilities. This situation can result in sub-optimal code — either in terms of productivity, as low-level optimizations that are easily automated are hardcoded, or in terms of performance in case crucial optimizations for hot code regions are prevented even though their applicability seems obvious from a high-level programmer's view.

In this work, we explore this issue from the compiler as well as the EDSL/library author perspective, defining, exploring, and offering an attempt to rectify the underlying source of many of these optimization shortcomings: the *semantic gap* between the high-level knowledge encoded in an EDSL's specification and invariants, and the low-level analysis that is the primary enabler for today's highly effective compiler optimizations. Our concrete contributions are as follows:

- An overview of the strengths and weaknesses of state-of-the-art compiler infrastructures, in particular as they relate to the EDSL use case,
- an investigation of the reasons for these shortcomings in current compiler architectures, and
- the design of a high-level compiler architecture facilitating semantics-aware optimizations.

The remainder of this paper is structured as follows. We begin by motivating and illustrating both the impressive capabilities of modern optimizing compilers and their weaknesses in several common EDSL scenarios in Section 2. Subsequently, in Section 3, we investigate the architecture of current state-of-the-art compilers, and how it fails in providing an effective basis for EDSL optimization. Semantics-aware compiler architectures, as presented in Section 4, attempt to rectify this shortcoming. Finally, in Section 5 we summarize our findings and provide an outlook on the future of semantics-aware compiler optimization

2. Motivation

In this section, we will consider several real code examples, as well as the results of compiling using a well-established optimizing compiler. We chose C++ for this motivation due to its suitability for implementing EDSLs for high-performance and embedded systems, but — as demonstrated in Section 3 — our conclusions apply equally to many other mainstream languages and infrastructures. All examples were compiled for the x86-64 architecture using GCC 8.1 with the `-std=c++17 -O3 -mavx2` options.

```
// <EDSL>
#include <cmath>

template<typename T>
struct Vec3 {
  T x, y, z;
};

template<typename T>
Vec3<T> operator-(const Vec3<T>& a, const Vec3<T>& b) {
  return {a.x-b.x, a.y-b.y, a.z-b.z};
}

template<typename T>
Vec3<T> operator*(const Vec3<T>& a, float f) {
  return {a.x*f, a.y*f, a.z*f};
}

template<typename T>
T lengthSquare(const Vec3<T>& a) {
  return a.x*a.x + a.y*a.y + a.z*a.z;
}

template<typename T>
T length(const Vec3<T>& a) {
  return sqrt(lengthSquare(a));
}

template<typename T>
Vec3<T> norm(const Vec3<T>& a) {
  return a * (1/length(a));
}

using Point = Vec3<float>;
using Force = Vec3<float>;

struct Object {
  Point pos;  // < position
  float m;    // < mass
};

Force gravity(const Object& a, const Object& b) {
  auto diff = a.pos - b.pos;
  float r2 = lengthSquare(diff);
  float l = (a.m * b.m) / r2;
  return norm(diff) * l;
}
// </EDSL>

float f() {
  Object s { {0, 0, 0}, 1 };
  Object e { {0, 0, 1}, .01 };
  Force f = gravity(s, e);
  return f.z;
}
```

Listing 1: Simple Newtonian Physics EDSL (C++)

Before we consider individual examples, we would like to note that for the purpose of evaluating the semantic gap and its impact on program optimization, any high-level library can be considered a small EDSL. Crucially, this applies also to standard libraries, which are of course not part of a language, even if perceived as such by many of its users. These observations allow us to formulate suitably compact yet still representative examples for this motivation section.

Listing 1 shows a small EDSL for defining Newtonian objects and determining their interactions. The resulting assembly in Listing 2 illustrates that, despite the level of abstraction employed, the compiler is capable of fully computing the result of function `f`, turning it into a simple data lookup. This is the type of sophisticated optimization capability which EDSL authors seek to leverage.

```
f():
  vmovss xmm0, DWORD PTR .LC1
  ret
.LC1:
  .long 3156465418
```

Listing 2: x86-64 Output for Listing 1

However, this highly successful optimization result is surprisingly brittle. In Listing 3 the only change is moving object allocation from the stack to the heap. While this code is semantically equivalent on a high level to Listing 1 — obviously so for a human reader versed in the language — the compiler's capabilities are severely neutered.

```
float f() {
  Object *s = new Object { {0, 0, 0}, 1 };
  Object *e = new Object { {0, 0, 1}, .01 };
  Force f = gravity(*s,*e);
  delete s;
  delete e;
  return f.z;
}
```

Listing 3: Physics EDSL Usage with Heap Allocation

In this case, the assembly result as depicted in Listing 4 performs the full computation. As highlighted, this includes heap allocations for both objects, as well as calls to the `sqrt` library function which were statically evaluated at compile time in Listing 2.

This example illustrates the first common reason for low-level state-of-the-art optimization failing to reach its full potential for high-level EDSLs: opaque memory management and **heap allocations**.

```
f():
  push rbp
  mov edi, 16
  push rbx
  sub rsp, 24
  call operator new(unsigned long)
; [... 4 instructions ...]
  call operator new(unsigned long)
  vmovss xmm1, DWORD PTR [rbx+4]
; [... 24 instructions ...]
  vmovss DWORD PTR [rsp+4], xmm0
  call operator delete(void*, unsigned long)
  mov rdi, rbp
  mov esi, 16
  call operator delete(void*, unsigned long)
  vmovss xmm0, DWORD PTR [rsp+4]
; [... 6 instructions ...]
  vmovss DWORD PTR [rsp+8], xmm3
  vmovss DWORD PTR [rsp+4], xmm4
  call sqrtf
  vmovss xmm2, DWORD PTR .LC2
  vmovss xmm5, DWORD PTR [rsp+12]
; [... 3 instructions + 10 constants ...]
```

Listing 4: x86-64 Output for Listing 3 (excerpt)

A second common source for optimization failure are **external library calls**, which the analysis cannot investigate and has no semantic understanding of. Listing 5 contains a very basic usage of the C++ standard library `std::map` template type. As outlined previously, any library interface can be considered a minimal example of an EDSL for optimization purposes.

```
#include <map>

int main() {
  std::map<int, int> dic;
  dic[5] = 10;
  return dic[5];
}
```

Listing 5: Basic STL map Optimization Failure

In this particular case, it is immediately obvious to programmers familiar with the semantics of the map data type that the given code is functionally equivalent to `return 10;`. Nonetheless, as the compiler lacks this semantic, high-level understanding of the data type (that is, the EDSL it represents) it generates around 300 lines of assembly code, including calls to the external functions for inserting new nodes into the tree representation for a map, and potentially replacing it. In the realistic large-scale usage of EDSLs, this type of issue is a common occurrence.

So far, we have considered the compiler perspective in this section, and examined its successes as well as limitations. Naturally, EDSL authors are also aware of these constraints, and have developed techniques for optimizing at the EDSL implementation — and thus library — level. One increasingly common technique of performing some level of compile-time optimization within a library in C++ are *expression templates* [7]. While these allow for good results to be achieved in some domains, they cannot close the semantic gap in EDSL compilation, as they lack crucial analysis-derived information.

```
// <EDSL>
struct Data {

  template<typename Op>
  Data map(Op op) const { /* ... */ };

  template<typename Op>
  Data filter(Op op) const { /* ... */ };

  bool empty() const { /* ... */ };
};
// </EDSL>

template<typename Pred>
bool f(Data data, Pred predicate) {
  return !data.map([](auto x) {
    return x.eval();
  }).filter(predicate).empty();
}
```

Listing 6: Simple Data Processing EDSL and its Usage

TABLE 1: Optimization Capability Comparison

	Low-level Analysis	Runtime Library	Semantics-aware Compiler
Stack Allocation	+	+	+
Heap Allocation	-	+	+
External library calls	-	+	+
Internal higher-order	+	-	+
External higher-order	-	-	+
Runtime data dependencies	-	-	-

Listing 6 demonstrates this issue. The `Data` type represents a simple EDSL for data processing. Data can be `filter`ed, `map`ped and checked for `empty`-ness. Note that the former two operations are naturally implemented by *higher-order functions*.

As the author of this EDSL, we would want to optimize successive calls of map/filter/empty based on the semantics of those operators. In particular, for optimal chained execution, we only need to iterate over the data once, and if the surrounding calling context is `!empty` we can implement an early exit as soon as a single element remains after the filtering operation. We could accomplish this goal by returning a *generator expression* tracking which operations to perform and only evaluating those at a later point, when the full information is available. However, at the library level, there is no way for us to ensure that `Op op` has no side-effects preventing deferred execution.

A compiler analysis would have no problem determining this fact in many cases, but could not benefit from it without the additional knowledge of library semantics which resides solely with the EDSL author. All types of EDSL operations which are customizable with user-provided code suffer from this issue at some level, which leads us to identify **higher-order functions** as the third and final major reason for EDSL optimization failure using traditional techniques.

As these examples illustrate, neither current compiler-side nor library-side optimization techniques alone can fully leverage the optimization potential of even rather simple EDSLs. Therefore, a new approach, *semantics-aware high-level compilation*, is required to combine the advantages of sophisticated compile-time analysis and optimization with DSL-specific semantic knowledge. Table 1 summarizes our findings in this section, and provides an outlook on the capabilities of semantics-aware compiler infrastructures as presented in Section 4.

3. Classic Compiler Architectures

The observed abilities and inabilities of optimizing compilers are rooted in their design. All widely-used general purpose compilers converting high level languages into executable code follow a common architecture. This architecture favors certain types of optimizations, while the

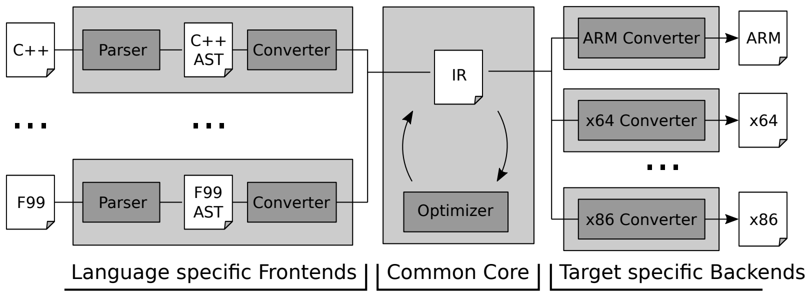

Figure 1: Classic Compiler Architecture

integration of others is severely constrained. This section provides a generic overview of a state-of-the-art compiler architecture, followed by some real-world examples and a discussion of the resulting restrictions on compilers' optimization capabilities.

3.1. Reference Architecture

Fig. 1 outlines the coarse-grained architecture of most contemporary state-of-the-art compilers. Derived from their original role of providing inter-platform portability for high-level programming languages, compiler architectures are designed to facilitate targeting a variety of instruction sets. To that end, the core of a compiler architecture is a common, platform independent intermediate representation (IR) which can be converted through various backends to target code.

While within a compiler's backend certain performance relevant optimization steps including instruction selection, instruction scheduling, and register allocation take place, the bulk of optimizations are performed on the IR level. A wide range of (mostly) platform independent optimizations, including constant propagation and folding, strength reduction, loop unrolling, inlining, vectorization, and dead code elimination are implemented on this level to facilitate re-use among all supported target platforms. By utilizing a common IR as the substrate for optimization passes and as the input of backend conversion steps, IRs are attractive targets for developers of high-level languages.

On the other end of the compiler's architecture, components translating different high-level languages into IR code are referred to as compiler frontends. Those typically comprise a parser for the input language, creating an implicit or explicit abstract syntax tree representation of the processed application code, which is converted into IR code in a second phase. In between, the abstract syntax tree (AST) structure is usually utilized for language specific operations like symbol name resolution, type deduction, template instantiation, and diagnosis steps reporting syntactic or semantic issues back to the end user.

Due to its central role, the design of the IR is the key element of every compiler architecture. Choosing its level of abstraction determines the complexity of incoming and outgoing conversion steps, program analysis, and transformations. It furthermore determines the scope of supported optimizations due to the available information and utilized primitives. For instance, supporting loops and arrays as primitives in an IR facilitates transformations targeting those common language structures, while increasing the complexity of analysis and transformation tools. To increase the variety of supported features, some compiler designs chose to implement multiple levels of IRs of gradually decreasing levels of abstraction.

Another common characteristic of widely-used real-world compiler architectures is its *translation-unit* based processing of application code. For modularity, high-level languages support the separation of program code into individual files. Each file contains the definition of a number of elements, typically data structures and functions, as well as declarations of elements defined in other files. During compilation, each file is processed independently by the compiler, forming a single *translation unit*. This breaks up large applications into smaller entities to be translated, thereby reducing resource requirements. It also facilitates the concurrent translation of individual parts of an application. However, it also limits the obtainable knowledge of the optimizer on the overall application. Elements only declared in one translation unit, defined in other units, can not be investigated by analyses and transformation steps within the compiler core. Only while linking the resulting binary object files, all the relevant information comes together. To that end, some architectures provide the option of link-time optimizations or just-in-time compilation support, facilitating the integration of optimization passes based on the additionally available information. In the case of just-in-time compilation, information obtained during runtime can also be considered by the optimization processes. However, all of these techniques are necessarily required to operate on or below the associated compiler IR's level of abstraction.

3.2. Real World Architectures

To provide evidence on the validity of our reference model, the following sub-sections will discuss real world architectures and their relation to our reference architecture.

3.2.1. GCC. The main distribution of the GNU Compiler Collection (GCC) supports 6 different frontend languages (C, C++, Objective C, Fortran, Ada, and Go) and 48 different target architectures, extended by numerous optional third-party modules. Internally it uses three different IR formats of decreasing level of abstraction: GENERIC, GIMPLE, and RTL. Additionally, different flavours of those are utilized (e.g. high GIMPLE and low GIMPLE), varying in the set of permitted constructs. The highest level IR, GENERIC, constitutes an AST like representation, reduced to a basic set of common imperative constructs. GENERIC is further on reduced to GIMPLE, a subset of GENERIC restricted to three-address code operations [8]. GIMPLE is the main IR for optimization passes. Finally, the lowest-level IR, the *register transfer language* (RTL), provides the data format for backends to perform register allocation and code generation operations.

3.2.2. LLVM. Like GCC, LLVM (formally low level virtual machine) supports numerous frontends and backends for a large variety of input languages and target platforms [9]. However, unlike GCC, LLVM places a strong focus on a single IR, the *LLVM intermediate representation*. It is a strongly typed, platform independent, assembly-like low-level language utilizing a reduced instruction set computing (RISC) based design. Its type system supports higher level data types like arrays, structs, and function types. All optimization passes are applied on this level of abstraction. Furthermore, the architecture facilitates link-time optimization as well as just-in-time compilation.

Due to their modular design, LLVM frontends can be independently utilized to build tools. In particular its C/C++ frontend *clang* provides valuable C/C++ parsing capabilities. Thus, the integrated clang-AST representation serves as the foundation for many source-level inspection, linter-, and refactoring tools.

3.2.3. JVM. Like LLVM, the *Java Virtual Machine* (JVM) architecture focuses on a single IR, the *Java bytecode* [10]. The Java bytecode has been specifically designed to be a platform independent IR interpreted or just-in-time compiled by JVM implementations. Thus, the JVM's IR is utilized as the binary format for shipping applications and the backend support is integrated into the runtime environment along with the optimization components. Similar to LLVM, Java bytecode constitutes an assembly like, platform independent IR. Furthermore, just as for LLVM, due to its widespread utilization and platform support, Java bytecode has become the target code for many additional third-party language frontends.

3.3. Limitations

As shown by our reference model, and underlined by the real world architectures, contemporary compiler platforms focus their optimization efforts on low-level intermediate representations. While suitable for constant folding, loop unrolling, and vectorization, the utilized IRs provide insufficient information for integrating desired higher-order optimizations based on semantic insights on utilized EDSLs.

For instance, to optimize the utilization of filters in our example EDSL, data sources, filters and predicates need to be recognized reliably and related with each other among translation unit boundaries. However, already in a frontend's AST, those elements are represented through generic definitions and declarations, reflecting their implementation details. This mirrors the nature of EDSLs being emergent structures of programming languages instead of actual language features. After lowering the AST into a common compiler IR, even more of this structural information is lost.

Thus, ASTs represent the most promising compiler stage for performing EDSL aware optimizations. However, typical ASTs — while accurately reflecting the semantics of the input code, including numerous syntactic elements aiding the development of application code — complicate the analyzability and transformability. In particular the requirement of conducting data flow and control flow analysis on ASTs is, while promising [11], known to be notoriously difficult due the complexity of the involved primitives [12], [13], [14]. Furthermore, the focus on a single translation unit limits the scope, and thus effectiveness of intended EDSL aware optimizations.

Consequently, due to the lack of IRs offering an adequate level of abstraction to realize semantic aware compilation, conventional compilers are required to be extended by additional IRs and processing steps on the high-level end of the conversion pipeline.

4. Semantic Aware Compiler Architectures

To integrate EDSL aware compiler support, an additional processing stage preceding the compiler frontend is required. Fig. 2 outlines our proposed overall architecture of such an optimization tool.

4.1. Overview

Like in a classic compiler, the frontend is tasked with parsing input files on a per-translation unit basis. Each input file is processed to obtain the resulting AST. In a second step, the AST is converted into a high-level intermediate representation (HL-IR), providing the necessary substrate for performing EDSL optimizations. The objective of the design of the HL-IR is to represent the processed program code, while facilitating the identification of EDSL objects and operators, supporting data and control flow analysis, offering transformability for re-writing operations, and the ability to be back-converted into the original source language — remaining faithful to imposed external interface requirements. The resulting HL-IR representation of all translation units of an application is then merged into a single HL-IR representation, providing a whole-program perspective to the subsequent optimization steps. Finally, the transformed IR is converted back into source-language code that can be utilized as a substitute of the original input code in the subsequent classic compilation process.

The EDSL optimizing compiler is thus performing a source-to-source conversion preceding the actual compilation process. Consequently, language portability and low-level optimization capabilities of classical compilers are preserved.

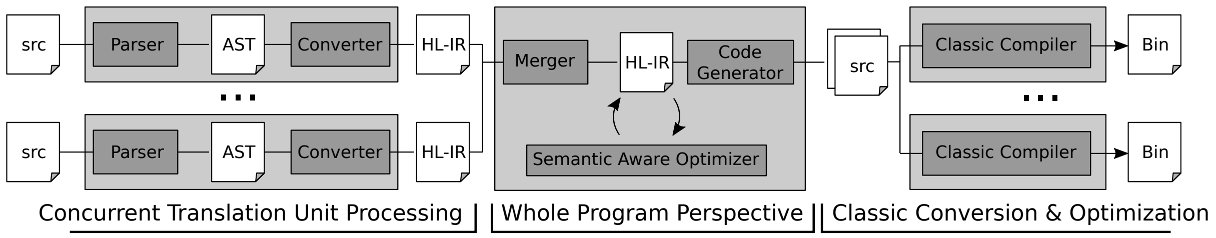

Figure 2: Semantic Aware Source-to-Source Compiler Architecture

4.2. Implementation

Our Insieme source-to-source compiler infrastructure [15] provides the necessary tooling for the realization of this approach for the C/C++ language family. The frontend parser is based on LLVM's C/C++ frontend, and the resulting clang AST is converted into Inspire [13], Insieme's high-level intermediate representation. Inspire provides the necessary high-level representation to facilitate the faithful modelling of C++ codes, while reducing complexity sufficiently to enable data and control flow analysis [14], [16]. Furthermore, Inspire code representations provide a binary format that can be stored in files and multiple Inspire fragments may be merged, as required by our architecture.

Inspire's modular, and extensible design enables the modeling of abstract EDSL concepts. Thus, EDSL primitives present in the C/C++ input code can be intercepted in the frontend's conversion step and encoded within Inspire accordingly. This enables the direct identification of primitives within Inspire during optimization steps as well as the specialization of code generation steps in the final step of the source-to-source compilation phase. Furthermore, by modeling EDSL concepts using abstract elements in the IR, implementation details of the underlying definitions remain hidden in the IR. Code analyses do not have to attempt to derive an EDSL operator's effect from its implementation, but can rely on its known semantics.

The latter point becomes especially important due to C++ customizability. As an example, in C++ every class can define a copy constructor, yet it is not guaranteed that every class is implementing it as intended. Indeed — although likely to cause difficult to identify issues in applications — creating copies of objects not being equivalent to their original instances still is a valid implementation for such a constructor. Program analyses, however, have to be conservative in their assumptions. Thus, effects of copy constructor invocations have to be deduced from their implementation in all cases. Being able to integrate the knowledge of faithful implementations of copy constructors of EDSL objects can significantly reduce the complexity of such analyses and simultaneously increase accuracy. Many similar examples can be identified, increasing the efficiency of whole-program high-level analysis.

```
auto fib = prec(             // creation of fib function
  [](int x) { return x <= 1; }, // base case test
  [](int x) { return 1; },   // base case
  [](int x, auto f) {        // step case
    return f(x-1) + f(x-2); // parallel recursive calls
  }
);
auto future = fib(12);       // creation of async task
```

Listing 7: Example Fibonacci code, in the AllScale EDSL

4.3. Example Use Cases

The high-level awareness of the semantics of the `new` and `delete` operators enables analyses based on the Insieme analysis framework [16] to treat the code fragments in Listing 1 and Listing 3 equally. Furthermore, by intercepting `maps` and map-based operators, the result computed by Listing 5 can be successfully deduced.

In another setup, the integration of awareness for C++11's standard library functions for task parallelism facilitated the static analysis of task creation and synchronization points. The resulting knowledge on the parallel structure of applications, and the capabilities of performing high-level code alterations, lead to significant reductions in parallel overheads [17].

Furthermore, our architecture enables the development of API designs relying on the integration of high-level compiler analysis steps. The AllScale environment, for instance, constitutes an example use case for our semantic aware compilation toolchain [18]. At its core, AllScale provides an EDSL for parallel algorithms comprising a small set of parallel operators [19]. Among those, the *prec* operator [20] constitutes the sole operator for spawning parallel tasks, similar to *async* calls offered by other parallel models. Listing 7 illustrates an example use case. The *prec* operator creates a parallel recursive function based on a base-case test, a base-case implementation and a step-case function. The implementation of *prec* can not make any assumptions of the passed functions. In particular, it can not assume those to be independent of some external state, as it is the case in this example. Consequently, no (template) library based implementation could determine whether or not parts of the computation could be e.g. off-loaded to a different compute unit like an accelerator or another node in a cluster without additional user-provided information. However, a compiler could make this decision based on static analysis.

The AllScale compiler performs this kind of analysis. It utilizes Insieme's infrastructure to realize semantic-aware compilation for the AllScale EDSL primitives. The implementation of the *prec* operator is intercepted in the frontend and replaced by an abstract symbol that can be located in the resulting Inspire representation. During an optimization step, the body of the passed arguments is located, potentially by following data and control flow dependencies, and analysed for data dependencies. Those dependencies are then integrated into the code base by replacing the original *prec* call by a runtime system call accepting corresponding meta information. The compiler therefore automatically extracts data dependency information from the specified user code that otherwise would have to be specified manually. Additionally, the AllScale compiler may generate different variants of the tasks to be processed, facilitating the utilization of heterogeneous environments.

The AllScale environment constitutes a proof-of-concept environment demonstrating the abilities of semantic aware compilation in combination with EDSLs based on the Insieme infrastructure.

4.4. Open Challenges

Besides already providing practical results, semantic aware compilation still has to face numerous challenges. Among those are:

- the design of the high-level intermediate representation: while Inspire provides a suitable IR for semantic aware compilation, it imposes several restrictions. In particular goto-expressions and exceptions are not supported. Furthermore its structural nature, opposed to C++'s nominal foundation, makes debugging and source-locality tracking difficult. Alternative designs could circumvent these problems.
- high-level program analysis scalability: while Inspire and its associated analysis frameworks considerably reduce the complexity of both the development and execution of high-level program analyses, the available techniques still fail to scale to large scale code bases. Future research in high-level program analysis could considerably improve the applicability of semantic aware compilation techniques.
- lack of means to specify high-level semantics of user defined APIs: in our current design the semantics of EDSL primitives are encoded in analysis and transformation passes. Facilitating user-defined specifications of semantic properties of EDSL primitives would greatly increase the usability, and thus the acceptance of this technology. It would enable EDSL designers to rely on compiler analysis and transformation steps while designing their interfaces and optimizations, without having to explicitly contribute compiler passes.

5. Conclusion

Our work has been motivated by an investigation of the strengths and weaknesses of optimizing compilers. While being capable of remarkable deductions under the right circumstances, heap-based operations and external library calls provide natural limitations due to the underlying compiler design. Unfortunately, many high-level interfaces and embedded DSLs utilized in applications are necessarily based on definitions relying on those optimization-prohibiting elements. Consequently, optimizing compilers fail to live up to their full potential — and to users' expectations.

To rectify, we have presented a semantic-aware compiler architecture implemented based on the Insieme compiler infrastructure. Its high-level intermediate representation and its associated type interception and static analysis features provide the foundation for semantic aware, high-level optimizations. We have detailed crucial design aspects and a number of example use cases. Furthermore, we have enumerated a range of open challenges constituting important areas of future research.

Acknowledgement

This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No 671603.

References

[1] A. Van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[2] S. Kozacik, "Unified, Cross-Platform, Open-Source Library Package for High-Performance Computing," EM Photonics, inc., Tech. Rep., 2017.

[3] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury, "Building Embedded Systems with Embedded DSLs," in *ACM SIGPLAN Notices*, vol. 49, no. 9. ACM, 2014, pp. 3–9.

[4] C++ Standard Committee, "Working Draft, Standard for Programming Language C++," ISO, Standard ISO/IEC 14882:2011, 2011.

[5] M. Torgersen, "Querying in C#: How Language Integrated Query (LINQ) Works," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 852–853. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297922

[6] Darcy, Joseph D., "JEP 126: Lambda Expressions & Virtual Extension Methods," Oracle, Tech. Rep., 2011. [Online]. Available: http://openjdk.java.net/jeps/126

[7] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, "High Performance Smart Expression Template Math Libraries," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012, pp. 367–373.

[8] J. Merrill, "Generic and Gimple: A new Tree Representation for Entire Functions," in *Proceedings of the 2003 GCC Developers' Summit*, 2003, pp. 171–179.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[10] B. Venners, *The Java Virtual Machine*. McGraw-Hill, New York, 1998.

[11] F. Logozzo and M. Fähndrich, "On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis," in *International Conference on Compiler Construction*. Springer, 2008, pp. 197–212.

[12] G. Necula, S. McPeak, S. Rahul, and W. Weimer, “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs,” in *Compiler Construction*. Springer, 2002, pp. 209–265.

[13] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, “INSPIRE: The Insieme Parallel Intermediate Representation,” in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 7–18.

[14] H. Jordan, “Insieme - A Compiler Infrastructure for Parallel Programs,” Ph.D. dissertation, University of Innsbruck, 2014.

[15] University of Innsbruck, “Insieme Compiler Project,” 2017. [Online]. Available: http://www.insieme-compiler.org

[16] A. Hirsch, “Insieme's Haskell-based Analysis Toolkit,” Master's thesis, University of Innsbruck, Innsbruck, Austria, 2017.

[17] P. Thoman, S. Moosbrugger, and T. Fahringer, “Optimizing Task Parallelism with Library-Semantics-Aware Compilation,” in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 237–249.

[18] H. Jordan, R. Iakymchuk, T. Fahringer, P. Thoman, T. Heller, A. Xavi, H. Khalid, K. Dichev, R. Emanuele, and L. Benoit, “D2.3 - Allscale System Architecture,” 1 2017, prerelease. [Online]. Available: http://www.allscale.eu/docs/D2.3%20-%20AllScale%20System%20Architecture.pdf

[19] H. Jordan, R. Iakymchuk, T. Fahringer, P. Thoman, T. Heller, X. Aguilar, K. Hasanov, K. Dichev, E. Ragnoli, and L. Benoit, “D2.6 - Allscale API Specification (b),” 1 2017, prerelease. [Online]. Available: http://www.allscale.eu/docs/D2.6%20-%20AllScale%20API%20Specification%20(b).pdf

[20] H. Jordan, P. Thoman, P. Zangerl, T. Heller, and T. Fahringer, “A Context-aware Primitive for Nested Recursive Parallelism,” in *Fifth International Workshop on Multicore Software Engineering (IWMSE16)*. Berlin, Heidelberg: Springer, 2016, pp. 1–12.