July 30, 2018

A least-disruptive mechanism to compile integral and pointer types of unknown compile-time size to EFI Byte Code

Kiran Kumar T P
Hewlett Packard Enterprise

Soumitra Chatterjee
Hewlett Packard Enterprise

A least-disruptive mechanism to compile integral and pointer types of unknown compile-time size to EFI Byte Code

Abstract and Introduction

Unified Extensible Firmware Interface (UEFI) is a specification which describes an interface between the operating system (OS) and the platform firmware. UEFI is a replacement for BIOS. The UEFI Specification describes how to write drivers for a device. One of the design goals in the UEFI Specification is keeping the driver images as small as possible. But the primary disadvantage of this specification is, these device drivers will be platform- and processor-dependent.

In addition to the standard architecture-specific (native) device drivers, the EFI specification also provides for a processor-independent device driver environment, known as the EFI Byte Code (EBC) Virtual Machine, which enables execution of device drivers written in EFI Byte Code, independent of the platform and architecture.

It follows that the target machine architecture is unknown during the compilation of an EBC image, since the compiled EBC image can potentially be run on any architecture and platform that supports the EBC Virtual Machine. In order to support writing such (EBC) code that can be executed unchanged on both 32- and 64-bit architectures, the UEFI specification describes a natural indexing mechanism, along with the introduction of a new integral type INTN, the size of which is dependent on the runtime target architecture. This implies that the size of the pointer (void*) and the newly introduced natural types (INTN/UINTN) is unknown at compile time, being dependent on the runtime architecture of the platform on which the EBC driver is executed.

The above specification creates several challenges in implementation of the EBC compiler. One of the basic assumptions in any compiler implementation is that the size and alignment of all types are known at compile time, enabling the compiler to be able to generate correct object layouts, structure field offsets, array indexes, load/store alignments, and to be able to correctly compute the function call stack frame allocations, etc. Accordingly, all static language compilers such as C/C++ compilers expect the type system of the language to be well-known at compile-time, with no concept of types dependent on the runtime platform. Unfortunately, the UEFI specification defines the source language for EBC device drivers to be the C language with certain restrictions, implying that the compiler implementation for the static C language be able to handle such runtime platform dependent types.

It would follow that a well-defined technique to handle pointer (void*) and natural (INTN/UINTN) types with runtime target dependent size is critical in being able to implement a C compiler with the ability to compile to (target) platform and processor independent EBC.

Unfortunately, there are no documented and well-defined techniques to implement a compiler with the ability to support such types with unknown compile-time size and alignment.

Problem statement

All static language compilers expect size of a pointer and size of every type to be known at compile-time. This reduces the runtime overhead. This makes the compiler design very straight forward for generating correct object layouts, structure field offsets, array indexes, load/store alignments, and to be able to correctly compute the function call stack frame allocations.

With the specification (Unified Extensible Firmware Interface Specification, V2.6) for variable pointers

and Natural types (INTN/UINTN) none of the existing compiler implementations can be used to generate EFI Byte Code (EBC). Also a well-defined technique to modify any existing compiler to support variable pointers and Natural types (INTN/UINTN) is missing. Out Invention disclosure fills this gap and provides a least-disruptive mechanism to compile integral and pointer types of unknown compile-type size to EFI Byte Code.

However a proprietary compiler implementation from Intel® targeting EBC does exists as of the time of this disclosure, there are no well-defined methodologies for a static language compiler to cater to compiling languages with runtime platform dependent type system. A well-defined technique to handle pointer (void*) and natural (INTN/UINTN) types with runtime target dependent size is critical in being able to implement a C compiler with the ability to compile to (target) platform and processor independent EBC.

Problem Solved

A typical compilation consists of the source language being transformed into an intermediate representation, which is then converted to the low-level machine code or assembly, roughly depicted as below:

Source code

~ (Preprocessor + Syntax analysis + Semantic analysis)

Abstract Syntax Tree (AST)

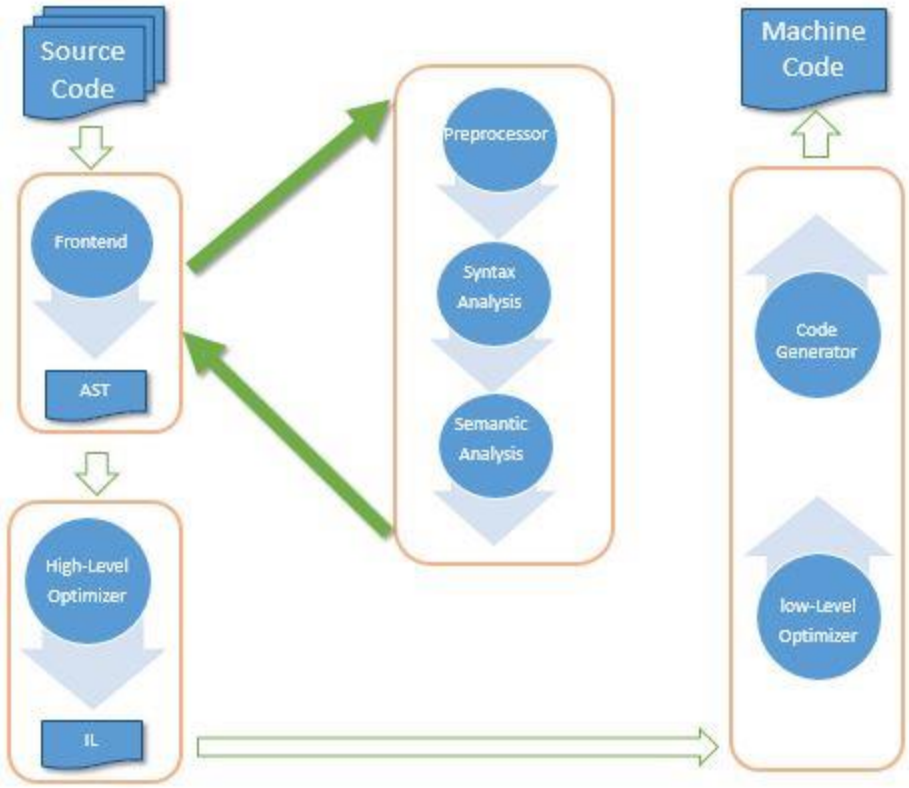
~ (High-level optimization)

Intermediate language representation (IR/IL)

~ (Low Level Optimization + Assembler / Code Generator)

Machine/Assembly code

This is depicted in below picture:



This invention disclosure proposes an approach to solve several of the above problems in the implementation of a compiler for domain-specific extensions to the C language specification, where the size of integral & pointer types are unknown at compile time, such as

1. Representing (new/existing) types with runtime platform dependent size in the Abstract Syntax Tree (AST)

2. Able to provide functionally correct implementations of the sizeof operator for pointer and types with unknown size, as well as aggregates composed of such types.

3. Extending the Intermediate Representation (IR) to be able to correctly and completely represent such types with unknown compile-time size.

Our solution

One of the primary aims of the techniques described in this disclosure is to lead the implementation complexity away from the compiler frontend, which is a language specific component, to the compiler backend, which the component concerned with the target architecture, which in this case is EBC.

1. Introducing Natural types (INTN/UINTN):

As described above, one of the primary objectives of this disclosure is to move the implementation complexity away from the compiler frontend to the compiler backend. To achieve this for EBC, where the target architecture is not known at compile-time, and natural types like INTN/UINTN should be treated as equivalent to the existing C99 "long" type, the size of which varies across 32- and 64-bit architectures. To ensure minimal changes into the compiler frontend processing the high-level source

language, the natural types can be assumed to be 64-bit wide, passing on the intermediate representation to the compiler backend, annotated accordingly, so that the respective target backend (generating EBC) can handle the complexity appropriately.

A similar approach is used in assuming pointers to be 64-bit wide and let the compiler backend handle it appropriately.

Note that the above approach relies on the intermediate representation being expressive enough to be able to retain the knowledge of the high-level types and the computations thereof should be retained so that the corresponding target backend is able to rewrite the computation as required by the index notation.

This approach, as stated in the goals, ensures that the frontend changes required are minimal towards supporting the new EBC target.

2. Implementation of the sizeof operator for pointer and natural types:

The UEFI specification dictates that size computations are deferred until runtime for types with unknown compile-time size such as pointers, natural types and aggregates composed using such types. UEFI specification describes an elaborate indexing mechanism to represent offsets and indexes involving such types with unknown compile-time size, which can then be used by the respective target (EBC) backend to provide runtime computation of the size.

We address the above requirement by making the compiler treat "sizeof" as an intrinsic call instead of an operator as described below.

The ISO C standard defines sizeof to be a compile-time operator that yields a compile-time constant, which works fairly well for static languages and the corresponding compiler implementations. Unfortunately, due to the runtime dependency of the size of pointer and natural types defined by the UEFI specification, the sizeof can no longer be treated as yielding a compile-time constant for such types and aggregates thereof.

In order to alleviate this restriction, this disclosure proposes that the sizeof operator be replaced by a compiler intrinsic call by the frontend, thus avoiding the evaluation of such expressions into a constant. The compiler frontend instead retains the sizeof computations in the form of an intrinsic expression in the IR, which is taken up by the respective target (EBC) backend to generate EBC instructions using the index notation as described by the UEFI specification.

The primary advantage of such an approach is that the changes required in the compiler frontend are minimal, which retaining most of the functional semantics of the sizeof operator. Once again, this approach does not require any changes to the existing IR.

One subtle side-effect that must be noted here is that the result of sizeof for such runtime platform dependent types will no longer be a constant, and hence cannot be used in programming constructs requiring a compile-time constant.

Disclosed by Kiran Kumar T P, Soumitra Chatterjee - Hewlett Packard Enterprise