Technical Disclosure Commons

Defensive Publications Series

March 08, 2018

An Approach Towards Supporting Types with Unknown Compile-Time Size in a Typical C Compiler

Kiran Kumar T P
Hewlett Packard Enterprise

Soumitra Chatterjee
Hewlett Packard Enterprise

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
T P, Kiran Kumar and Chatterjee, Soumitra, "An Approach Towards Supporting Types with Unknown Compile-Time Size in a Typical C Compiler", Technical Disclosure Commons, (March 08, 2018)
https://www.tdcommons.org/dpubs_series/1084

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

An Approach Towards Supporting Types with Unknown Compile-Time Size in a Typical C Compiler

Abstract and Introduction

One of the basic premises in the implementation of a ISO C compiler is that the size and alignment of all types specified by the language is known at compile time. The ISO C standard unambiguously states the size and alignment requirements of all types specified by the language. It follows that the size and alignment of all composite types is hence also known at compile time. The compiler uses the knowledge of the size and alignment of the types to be able to correctly allocate objects, generate object layouts, compute member offsets into structures and array indices, load/store alignments, allocate function call stack frame, etc. Even for platforms that support both 32-bit and 64-bit data models, with the latter providing wider sizes for some of the basic types such as pointers and long, the compiler needs the data model to be specified at compile-time in view of the above considerations.

There are certain domain-specific extensions to the C language, most notably the source language for EFI Byte Code (EBC), that do not specify the size and alignment of certain types, leaving them to be determined by the underlying architecture at runtime. Such specifications allow for the generated code (typically a high-level byte code) to be platform agnostic, allowing execution without change on multiple platforms. One of the primary motivations for such non-standard lenient language specification is the significant cost savings realized by the consolidated development and testing of the platform agnostic code deployed on multiple platforms.

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 domain-specific extensions to the C language specification for integral and pointer types of unknown compile-type size none of the existing compiler implementations can be used to handle this. Also, a well-defined technique to modify any existing compiler to support such requirements are missing as of today. This disclosure addresses this gap and provides a generic approach to handle pointer types and types with runtime target dependent size to be able to implement a C compiler with the ability to compile to (target) platform and processor independent application.

One such requirement is for generating EFI Byte Code (EBC). There are no well-defined methodologies for a static language compiler to cater to compiling languages with runtime platform dependent type system.

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)

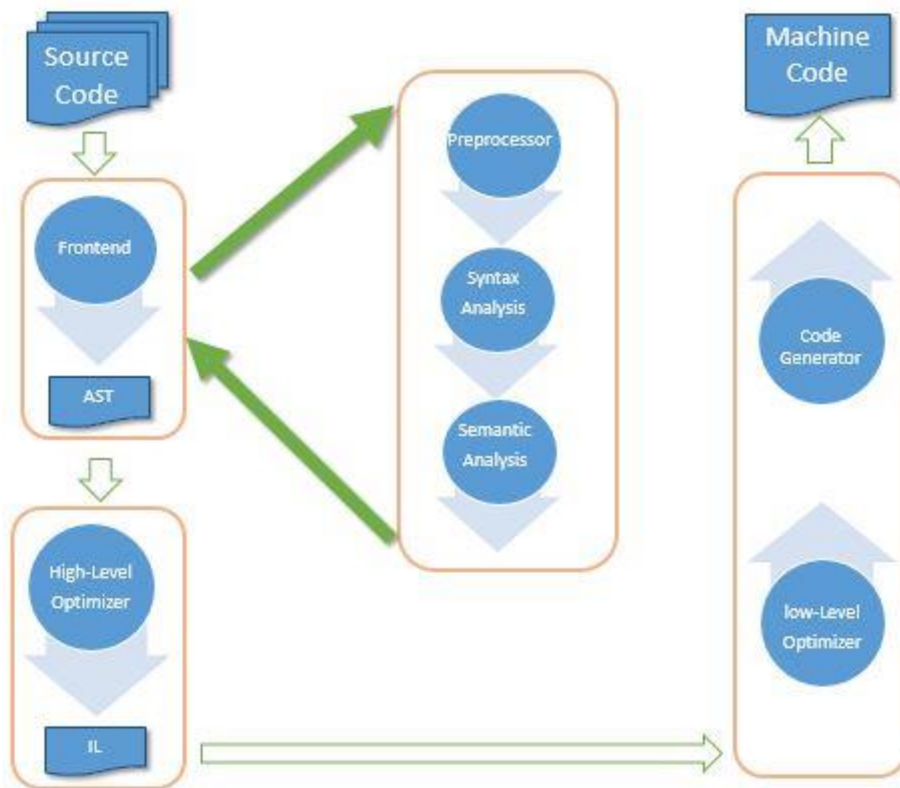
~ (High-level optimization)

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

https://www.tdcommons.org/dpubs_series/1084

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 is the component specifically concerned with the target architecture (e.g. EBC).

1. **Handling existing types (long) or introducing new integral types with unknown compile-type size**

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, where the target architecture is not known at compile-time, types with unknown compile-type size 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, these new types (which is in turn type "long") 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 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.

2. **Implementation of the sizeof operator for pointer and integral types with unknown compile-type size**

The size computations should be deferred until runtime for types with unknown compile-time size such as pointers, new integral types and aggregates composed of such types. This can be achieved by designing the compiler to leverage the existing implementation towards support for Variable Length Array (VLA).

Variable Length Array (VLA) is described by the C99 ISO C standard as an array, the size of which is not known at compile time. Typical compiler implementations supporting VLA defer the size computations of the VLA to runtime as described below.

The size of an array is the result of the product of the size of the base type with the number of elements in the array.

In case of VLA, while the size of the base type of the array is known, the number of elements is unknown at compile time. The size computation for a VLA is hence generated as a runtime expression, with the number of elements being supplied as a runtime value.

In considering the need for runtime platform dependent size, this disclosure proposes a similar approach, using a fixed number of elements known at compile time, and using a runtime

variable for the size of the base type. The runtime variable depicting the size of the base type can then be computed at runtime using the target specific initialization in prologue section.

The advantage of this approach is that the existing implementation of VLA support is leveraged to compute the size of types with unknown compile-time size at runtime. This approach requires minimal changes into the compiler frontend and backend, with no changes required in the IR.

Disclosed by:

Kiran Kumar T P

Hewlett Packard Enterprise

Soumitra Chatterjee

Hewlett Packard Enterprise