An LLVM Back-end for MLton

by

Brian Andrew Leibig

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Matthew Fluet

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

August 2013

The project "An LLVM Back-end for MLton" by Brian Andrew Leibig has been examined and approved by the following Examination Committee:

Dr. Matthew Fluet
Assistant Professor
Project Committee Chair

Dr. James Heliotis
Professor

Warren R. Carithers
Associate Professor

Abstract

An LLVM Back-end for MLton

Brian Andrew Leibig

Supervising Professor: Dr. Matthew Fluet

This report presents the design and implementation of a new LLVM back-end for the MLton Standard ML compiler. The motivation of this project is to utilize the features that an LLVM back-end can provide to a compiler, and compare its implementation to the existing back-ends that MLton has for C and native assembly (x86 and amd64). The LLVM back-end was found to offer a greatly simpler implementation compared to the existing back-ends, along with comparable compile times and performance of generated executables, with the LLVM-compiled version performing the best on many of the benchmarks.

Contents

Abstract iii

1 Introduction 1

2 Background 4
- 2.1 Back-end Targets 4
- 2.2 MLton Design and Architecture 6
  - 2.2.1 IL Pipeline 7
  - 2.2.2 Back-end Design 10
  - 2.2.3 Execution Model 10
  - 2.2.4 Trampolining 12
  - 2.2.5 Global Variables 13
  - 2.2.6 Machine IL 15
- 2.3 LLVM Design 19
  - 2.3.1 Representation 20
  - 2.3.2 Structure 20
  - 2.3.3 Type System 23
  - 2.3.4 Instruction Set 24
  - 2.3.5 Intrinsic Functions 28
  - 2.3.6 Optimizing LLVM IR 29
  - 2.3.7 Using LLVM for MLton 30
- 2.4 Related work 31
  - 2.4.1 Essential Haskell Compiler 31
  - 2.4.2 Glasgow Haskell Compiler 31
  - 2.4.3 Erlang 32
  - 2.4.4 Other Projects 33

3 The LLVM Back-end 34
- 3.1 Code Generation Strategies 34

3.2 Back-end Compilation Process 36
3.3 LLVM Code Generation 38
3.3.1 Signature and Structure 38
3.3.2 Code Generation Strategy 39
3.3.3 Translating Machine IL to LLVM 40
3.3.4 Main Module and Label Indices 45
3.4 Other Stages 46
3.4.1 LLVM Optimizer 46
3.4.2 LLVM Compiler 46

4 Evaluation 47
4.1 Implementation Complexity 47
4.2 Executable Quality 48
4.2.1 Code Size 48
4.2.2 Compilation Time 49
4.2.3 Execution Time 52
4.3 Summary 54

5 Conclusions 56
5.1 Future Work 57

Bibliography 58

List of Tables

2.1 Description of LLVM types 24
2.2 LLVM intrinsic functions used by the LLVM back-end for MLton. 28

4.1 SLOC comparisons of different back-ends 47
4.2 Specifications of the computer running the benchmarks 48
4.3 Code size results of benchmarks in bytes 50
4.4 Compilation time results of benchmarks in seconds 51
4.5 Execution time results of benchmarks in seconds 53

List of Figures

1.1 General compiler pipeline 2

2.1 MLton Compilation Process 8
2.2 Diagram of the memory layout of a running program compiled by MLton . 11
2.3 LLVM toolchain and compilation process 21

3.1 MLton Back-end Pipeline 37

Listings

2.1 Example of a trampoline 12
2.2 Example of a chunk 14
2.3 Example of non-SSA pseudocode 23
2.4 Code from listing 2.3 compiled to LLVM IR 23
3.1 Signature for MLton's LLVM code generator 38
4.1 Source of the *flat-array* benchmark 54

Chapter 1

Introduction

Compilers are some of the most important tools in the software ecosystem, being the tool that turns code written in high-level programming languages to executables that can run directly on hardware. They are also complex software systems, having to face the challenges of transforming high-level language constructs and abstractions to efficient lower-level representations, and handle the portability aspects of supporting compilation to multiple platforms.

Compilers are generally architected in a three-phase design: The front-end, optimizer and back-end, shown in figure 1.1. The front-end is responsible for the lexing, parsing, and type checking of the source code, transforming it into an abstract syntax tree (AST), which acts as an intermediate representation (IR) in the compiler. The optimizer improves the efficiency and performance of this intermediate representation by transforming the code to simpler yet semantically equivalent versions, possibly using different representations if it is useful to do so. In the back-end, the code is emitted to an executable form, usually as either machine code that can be directly run on hardware, or bytecode that can be run on a virtual machine.

Because of the split between the different components of a compiler, it is possible for multiple compilers for different languages to share the same back-end system. One of the long-standing issues in compiler design is how to best solve the challenges in utilizing a common back-end technology. Compiler developers want to ensure the compiled programs perform the best that they can, but they also want to leverage language-agnostic tools and libraries that free them from being concerned with the low level details required to make the

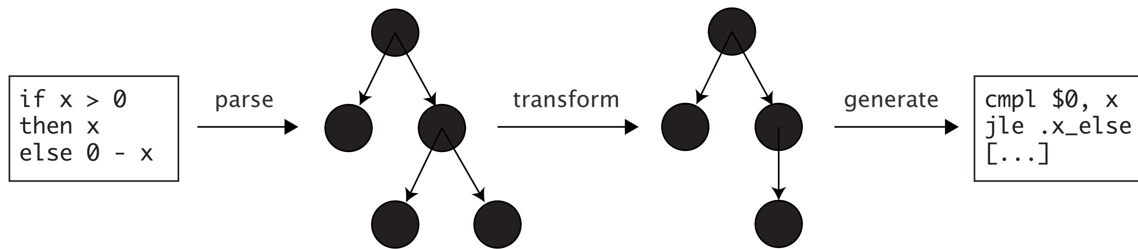

Figure 1.1: General compiler pipeline

compiler generate high-performance executables. One such project that solves this issue effectively is LLVM [25], which defines a high-level, target independent assembly language that can be aggressively optimized and compiled to several different architectures.

This report examines an LLVM back-end for MLton [19], an open source, whole-program optimizing compiler for the Standard ML (SML) programming language. MLton is written primarily in SML with a runtime system written in C, and is able to self-host. MLton's features include support for a large variety of platforms and architectures, the ability to handle large, resource intensive programs, and aggressive compiler optimizations that lead to efficient programs with fast running times. The MLton compiler currently has three back-ends: C, x86, and amd64. The C back-end emits the compiled program as C code, and uses an external C compiler to compile to native code. The x86 and amd64 back-ends, known together as the native back-ends, emit assembly language directly which is then assembled by the system assembler into native code. The native back-ends offer better performance and compile times, but have a limited set of supported platforms.

In this report, we will be looking at the design and implementation of the new back-end for the MLton Standard ML compiler, using the LLVM IR as a target and its associated toolchain to handle the compilation process. The hypothesis is that LLVM will be a feasible back-end choice for MLton based on evaluation of the following criteria:

- How complex is the implementation of the LLVM back-end compared to the existing back-ends, with respect to code-size and simplicity of design?

- How much does the new back-end affect MLton's overall compilation times?
- How good is the quality of executables generated the different back-ends, based on file size and running time of various benchmark programs?

The rest of the report is organized as follows. Chapter 2 goes over the design of MLton and the techniques it uses to compile Standard ML programs. It also goes over the design of LLVM, and how it can be used as a compiler back-end. Chapter 3 describes the design and implementation of the LLVM back-end, going over the design choices made and the strategy used for translating to LLVM. Chapter 4 goes over the evaluation of the LLVM back-end according to the hypothesis, including results from running the MLton benchmark suite. Chapter 5 gives concluding remarks for this project and ideas for further work.

Chapter 2

Background

2.1 Back-end Targets

A big challenge in the design of compilers for high-level languages is finding the best way to implement the compiler's back-end. Compiler developers want to use a technique that allows the generated executables to run as efficiently as possible, as that is an important factor in judging the quality of the compiler. However, they also want to minimize the effort in implementing the back-end, ideally by sharing infrastructure with other compilers. Due to many compiler writers prioritizing the former, this challenge has led to a situation where the popular implementations of languages like C, Java, Python, and Haskell share little or nothing in common.

A big factor in the design of back-end is the kind of language the back-end will target. Possible targets can be grouped into four categories [27]:

1. Native Assembly: This is the most straightforward choice, as it offers the compiler writer the most control over how the compiled executable is written, and it minimizes the reliance on external tools as all you need is the system assembler. However, this approach takes the most amount of effort by the compiler writers to implement and maintain. Assembly languages are complex and expose many low level details, so it takes a considerable amount of effort to write an effective implementation. Also, an assembly back-end is target specific, so adding support for a new architecture in a compiler requires a lot of effort.

2. High level languages: Compilers can output to a different high-level language, using compilers for that language as external tools to complete the compilation process. Most often the language is C [22], because it is low level enough to not interfere too much with the semantics of the source language or final IR of the compiler, and because the language has excellent performance and little runtime overhead. Also, this strategy grants portability for free due to the ubiquity of C compilers across most computing platforms. This approach still has its flaws, due to lack of fine control of code generation details such as tail calls, and longer compilation times due to having to parse and compile source again as part of the back-end stage.

3. Managed virtual environments: A popular choice for programming languages in the past couple of decades is to compile down to a relatively high-level and portable bytecode format, and at run-time have it execute on a virtual machine. This has been the choice of execution model for modern compiled languages like Java and C#, and for scripting languages like Python and Lua. To help overcome the performance penalty of executing on an interpreter, the virtual machines often use just-in-time (JIT) compilation which compiles the executing code to native machine code as it gets executed. The benefits of this technique include portability on all platforms the virtual machine runs on, and allowing code to take advantage of existing libraries and rich runtime features provided by the platform such as garbage collection and exception handling. However, this approach generally suffers from worse performance compared to compiling to native assembly, and can raise issues when the runtime features of the platform do not match up nicely to the runtime features needed by the language. For example, the garbage collection techniques may not work well based on the style and frequency in which the language allocates objects, or the exception handling system on the virtual machine may not match the semantics of exceptions in the language.

4. High-level assembly: The final alternative strikes a middle ground between native

assembly and high-level languages or bytecode. High level assembly languages are low-level enough to not interfere with the abstractions in high level programming languages, but also high-level enough to abstract away the very low level details of assembly language such as register allocation and instruction scheduling. They also have the ability to be optimized in both target-independent and target dependent ways, producing high-performance executables. One such language that implements all of these features is the LLVM IR.

The MLton compiler originally went by the second method, emitting C code and using the GNU C Compiler (GCC) to handle compilation and linking from that point on. Because of the multitude of platforms GCC runs on, MLton itself is able to run on and compile code for a large number of platforms. Also, because MLton's runtime system is written in C, integrating the runtime with the emitted code is simple as well. However, in an effort to increase the performance of compiled programs, the first choice was also pursued and native back-ends were added for two of the most commonly used architectures (x86 and amd64). As expected, the native back-ends produce better code, but they have a substantially more complex implementation. This project adds a new back-end by going with the fourth option and using the high-level assembly language provided by LLVM to achieve excellent performance of compiled programs with minimal effort.

2.2 MLton Design and Architecture

To compile a functional language as high-level as Standard ML to fast native code requires facing the challenge of mapping the high-level constructs in SML to the low level methods of computation used in an assembly language, all while optimizing along the way to minimize any inefficiencies that are found. MLton's strategy is to orient the flow of the program being compiled through several intermediate languages (ILs) that progressively get lower level and closer to the final target language.

2.2.1 IL Pipeline

MLton's method of efficiently compiling SML code involves translating the program through a series of intermediate languages where abstraction level is successively lowered and the representation is simplified through many optimizations that are performed along the way. Since MLton is a whole-program compiler, it has the source code of the entire program at hand when optimizing, and having the maximum amount of information possible makes these optimizations substantially more effective. Figure 2.1 shows the overall structure of MLton's compilation process and each IL involved. An overview of each IL is listed below.

AST This is the first IL, generated by the compiler front-end which lexes and parses SML source code. MLton applies a pass called *Elaborate* which type-checks the program according to the definition of Standard ML [18]. If successful, it performs defunctorization which removes all module constructs, giving each identifier a globally unique name to avoid name conflicts, and translates to CoreML.

CoreML This is a polymorphic, higher-order representation with nested patterns, based on predicative System-F. This IL is optimized through an aggressive dead code elimination pass, removing all unused declarations, and then translated to XML by performing linearization and removing nested patterns. (In this paper, XML is a MLton-specific IL and is not to be confused with the Extensible Markup Language).

XML This is a higher-order intermediate language in A-normal form, which has no nested patterns and has every intermediate computation bound to a variable. This IL is optimized through passes that eliminate all unused type arguments, and simplify the code by eliminating unnecessary declarations. After optimizations are run, this IL is translated to SXML.

SXML This is a simply-typed version of XML. It comes from the *Monomorphise* pass, which eliminates parametric polymorphism by duplicating the definition of data types and functions at each use with the types they are instantiated with. This IL then goes

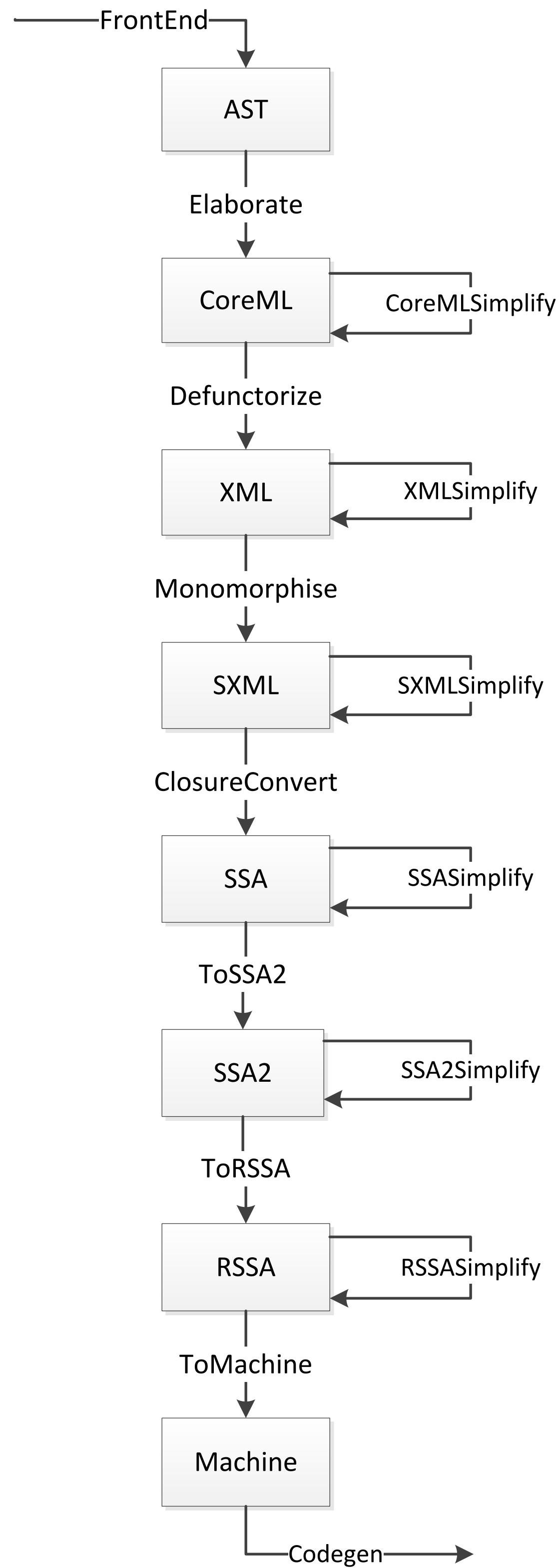

Figure 2.1: MLton Compilation Process

through its own simplification pass, and is then translated to SSA.

SSA SSA comes from SXML after it goes through the *ClosureConvert* pass. This is a relatively major pass as the IL is lowered to a first-order form from a higher-order form, and becomes imperative rather than functional. The translation uses defunctionalization [1] to convert closures into a more explicit form. SSA (static single assignment) is a first-order, simply typed IL where the program is represented by a control-flow graph (CFG) of basic blocks, which each block containing a list of instructions and ending with a transfer instruction to one or more other blocks. This is the main IL used for optimizations, such as constant propagation, function inlining, and loop invariant code motion. SSA is translated to SSA2, a slightly more efficient and lower-level version of SSA.

SSA2 SSA2 is a variant of SSA, which has more efficient representations of arrays and objects. In SSA2, `ref` types are transformed to an object with a single mutable field, which avoids heap allocation and the extra layer of indirection. SSA2 is optimized through "flattening" passes, which for example, transforms a `(int * int) array` from an array of pointers to int pairs into an array of int pairs. SSA2 is then translated to RSSA, another SSA variant that is lower level.

RSSA This is another variant of SSA, but with explicit representations for objects. This IL integrates some of the MLton runtime through adding implementation of exception handlers and profiling symbols (if the program was compiled with profiling enabled). RSSA is then translated to the final IL, Machine.

Machine This IL represents an abstract machine, with operations similar to actual hardware. This is not in SSA form, as in this form SSA registers have been allocated to Machine registers, which are mutable variables for primitive types and pointers. Also when translating to Machine, functions are assigned to groupings called chunks. The splitting of functions into chunks (called chunkifying) helps reduce compile times for large programs, as each chunk can be compiled or assembled separately and then

linked together at the last step. Machine is then used by the back-end of MLton to complete the compilation process.

2.2.2 Back-end Design

The MLton back-end process is responsible for translating the Machine IL to its equivalent in a non-MLton-specific low level language, where external compilers, assemblers, and linkers can complete the compilation process. MLton currently has back-ends that translates Machine IL to C, x86 assembly, or amd64 assembly; and now has a back-end that compiles to LLVM IR. The code generator (or codegen) used when compiling with MLton can be selected with the `-codegen` flag. If not given, MLton defaults to the native code generator (or codegen) when running on a platform that supports it, otherwise it falls back to the C codegen.

The Machine IL contains the program being compiled as one or more chunks. Each chunk is compiled to machine code based on the given code generation method. However, all codegens require an additional module known as the *main module* to be compiled. The main module holds the `main` function of the program, which makes it the entry point of a MLton-compiled executable. Along with being the starting point, this module is responsible for doing all of the boilerplate and setup required by the MLton runtime. An important aspect of this is initializing a C struct called `gcState`, which contains all of the data necessary for the MLton runtime to manage the ML heap. Certain fields in the `gcState` structure are referenced in the Machine chunks as well. The main module also contains definitions of global variables and arrays which are used by the program for various purposes, which are described in section 2.2.5.

2.2.3 Execution Model

An important factor in MLton's design is that it defines a special kind of execution model for the programs it compiles to account for certain runtime features that the Standard ML language requires. Specifically, it has its own way representing the call stack for SML

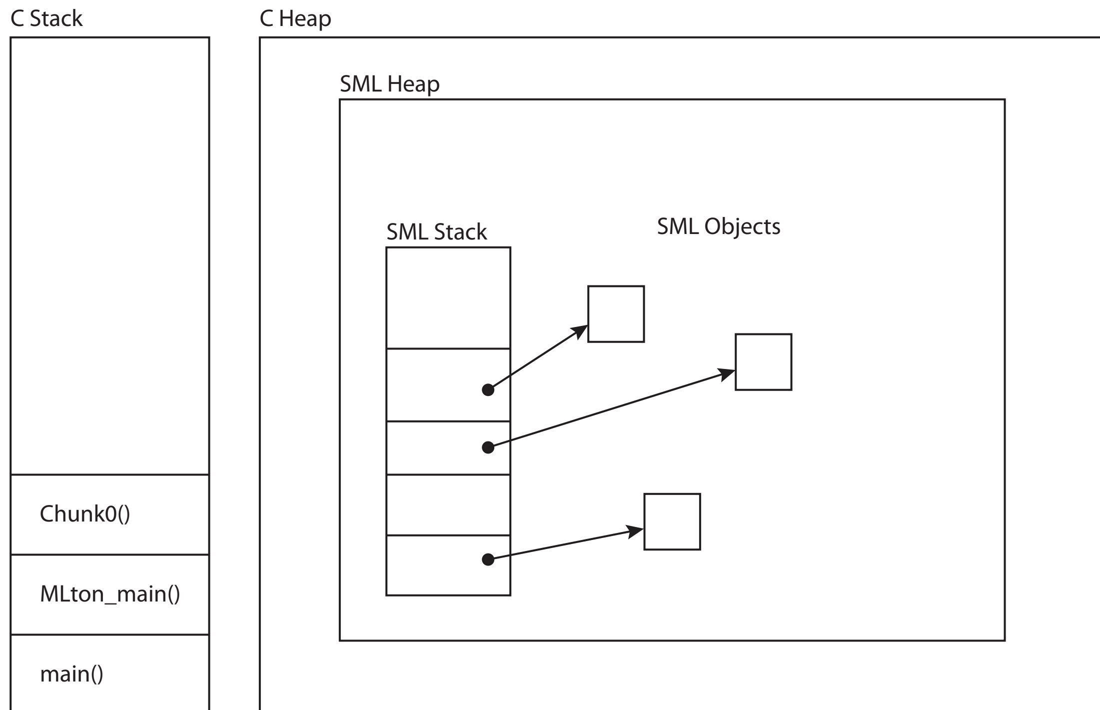

Figure 2.2: Diagram of the memory layout of a running program compiled by MLton

functions. Standard ML, being a functional language, relies heavily on recursion for implementing complex procedures, and recursive function calls can easily grow the call stack to an extreme size. Thus, if SML functions were mapped to C functions in the compilation process, programs could easily cause a stack overflow when executing if they rely on a large number of recursive function calls. MLton's solution to this problem is to place it in the C heap, where limitations on memory usage are less strict, and have it be managed at runtime as function calls happen. A diagram of the MLton memory model is given in figure 2.2.

The SML stack serves the same purpose as any other call stack. It holds a linear collection of stack frames (also known as activation records) which hold local variables used in the function corresponding to that frame, function arguments, and the return address of the caller to that function. The stack is created by the runtime system, but primarily managed by compiled code and inspected by the runtime for garbage collection.

```
int (*chunks[])();
int nextFun;
int main() {
    int nextChunk;
    nextFun = ...;
    while (1) {
        nextChunk = (*chunks[nextChunk])();
    }
}
```

Listing 2.1: Example of a trampoline

2.2.4 Trampolining

To implement the execution model described in section 2.2.3, MLton uses a technique called *trampolining* with the C and LLVM code generators to control the execution of SML functions without growing the C stack. When a MLton-compiled program starts running, the “main” function is executed, and after doing neccesary setup and boilerplate for the runtime, the trampoline is executed. With this method, calls to other SML functions do not grow the C stack as control flow either jumps elsewhere within the chunk or back to the trampoline where the chunk function returns and a new one is ran. The C stack only grows when calling other C functions, which are mostly functions provided by the runtime that do memory management. The trampoline conceptually looks like the code in listing 2.1. Here, `nextChunk` and `nextFun` are integers that uniquely identify a chunk and a basic block within that chunk. The `chunks` variable holds an array of function pointers to each chunk. The `nextFun` global variable is initialized with a value referring the first SML function to be executed. When the body of the while loop is executed, the `chunks` array is indexed to get a function pointer to the next chunk to enter, which is then dereferenced and called.

The implementation of a chunk function conceptually looks like the code in listing 2.2. The central idea is that each basic block inside the chunk has a unique integer called the *label index* which is held in the `nextFun` variable, and when a chunk is entered, the switch

statement transfers control to the entry label of that function. The chunk has local variables for two of the most important members of the `gcState` structure, `frontier` and `stackTop`, which act as cached versions of the values in `gcState`. There is also a local variable for each Machine IL register. When an SML function calls another SML function, the code pushes new stack frame on the stack by offsetting the `stackTop` variable, stores necessary data into the new stack frame, and transfers control flow to the entry label of that function via a `goto`. When a function returns, the stack frame is popped off by offsetting the `stackTop` by a negative amount, and a "return address" is loaded from the stack. This return address is not a real memory address but rather the label index of a block where control flow resumes after the called function returns. The program then does a `goto` to the `top` label right before the switch statement, and the switch statement transfers control flow to the block where the function resumes execution.

2.2.5 Global Variables

The main module defines a number of global variables which all chunks have access to. These variables serve two main purposes: to allow different chunks to communicate data between each other, and to assist the runtime by giving it more details on the code's runtime behavior. Some of the important variables are listed below:

- `gcState`: This is a C struct with over 50 fields that is used to hold all of the state needed by the MLton garbage collector. Some of its fields, like `frontier` and `stackTop` are directly manipulated by the SML code.

- `global`<type>: For each type used by Machine IL, there is global array for values of that type (*e.g.* `globalWord32`, `globalObjptr`). This is for data that is always in scope and live for the whole program execution, serving the same purpose as static variables.

- `CReturn`<type>: Also for each Machine IL type, there is a variable where results of calls to C runtime functions are placed.

```
extern int nextFun;
int Chunk0() {
    Pointer frontier;
    Pointer stackTop;
    int l_nextFun = nextFun;
    <declarations of register variables>
top:
    switch (l_nextFun) {
    case 0:
        ...
L_0:
        ...
L_1:
        ...
    case 1:
        ...
L_2:
        ...

    default:
        nextFun = l_nextFun;
        return nextChunk;
    }
}
```

Listing 2.2: Example of a chunk

- `frameLayouts`: This is an array of frame descriptions for stack frames of C and SML functions. Each entry is an array containing three values: an enum value to determine if it is for a C frame or ML frame, a reference to an array of offsets of live pointer values on the frame that locate where the arguments and local variables are, and the size of the stack frame in bytes. This is used by the MLton runtime, and specifically the garbage collector, when traversing stack frames to figure out which objects are still in use. The index of the frame layout for a specific function in this array is the same as that function's label index. This allows the runtime to find the frame layout of a frame by reading the value in the return address slot.
- `nextChunks`: An array of function pointers to chunk functions. This array is indexed using an SML function's label index, so the chunk pointed to at index i is the chunk where the function with label index i is located. This array is used in the trampoline to get the chunk function to start executing for a certain SML function.

2.2.6 Machine IL

Machine IL is the final intermediate language in the MLton compiler pipeline. All code generators work by mapping the Machine representation of the program to the equivalent constructs in the target language. This section describes the type structure that makes up the definition of Machine.

Program This type is the top-level structure for programs being compiled; there is only one of these. This is a record containing a list of Chunks which are translated by the codegen, along with global data used by the main module such as frame layouts.

Chunk This is a record that holds a list of Blocks, along with a ChunkLabel which identifies each chunk, and a closure that takes a CType and returns the number of Machine registers of that type the chunk needs.

Block This represents a basic block in a control flow graph, containing a sequence of statements and ending with a control flow transfer to one or more other basic blocks.

Kind Each basic block has a Kind which contains information as to its purpose. Kind is a variant type which can be one of the following values:

- Cont: A label jumped to after a Call or Return.
- CReturn: A label jumped to after a CCall (defined in Transfer) is executed.
- Func: A label that indicates the beginning of a SML function.
- Handler: A label jumped to after a Call or Raise.
- Jump: An ordinary inner block of a function.

Statement A block executes any number of statements, which may be one of the following:

- Move: Move a value to a register or location in memory.
- Noop: Do nothing.
- PrimApp: Perform a simple operation on one or more operands. The operation is a value of type Prim, which is defined outside of the Machine signature as it is used throughout the compiler in several other stages. Prim operations will be described later in this section.
- ProfileLabel: This statement is not generated for C or LLVM code generators. For native code generators, this is used to insert labels into assembly for efficient mapping from PC to profile info. For C and LLVM, the profile info is maintained dynamically (at a small runtime cost).

Transfer Every basic block is terminated by a Transfer instruction, which may be one of the following:

- Arith: Perform a basic arithmetic Prim operation (add, subtract, negate, or multiply) with overflow detection, and branch to different blocks depending on whether or not the operation overflowed.

- CCall: Call a C function, which is either MLton runtime function or a function imported through the FFI. After the call, this does an unconditional jump to label if the function returns (some runtime functions like `MLton_halt()` do not return as they terminate the process).
- Call: Call another SML function by jumping to a Func label, which may or may not be in the same chunk as the current one. If it's not in the same chunk, go back to the trampoline by exiting the current chunk.
- Goto: Unconditionally jump to a label, which is always in the same chunk.
- Raise: Raise an exception by jumping to a label indicated by a label index on the stack, and popping all frames off the ML stack up to the nearest enclosing exception handler.
- Return: Return from a SML function call by jumping to a label indicated by a label index on the stack, and popping off one frame from the ML stack.
- Switch: Conditionally branch to one of two or more labels depending on the value of an operand.

Operand The representation of a value, which is used as the arguments to statements or CCalls. An operand may be one of the following variants:

- ArrayOffset: An value contained in an array, given by an index and scale of a pointer.
- Cast: Another operand casted to a different type.
- Contents: The value pointed to by an operand that is a location in memory.
- Frontier: The value of the `frontier` local variable.
- GCState: The address of the `gcState` local variable.
- Global: The value of one of the global variables defined in the main module.
- Label: The label index of a basic block.

- Null: The null pointer.
- Offset: The value pointed to by a pointer offsetted by some number of bytes.
- Real: A floating-point literal.
- Register: The contents of a Machine IL register.
- StackOffset: The contents of an offset of the `stackTop` local variable.
- StackTop: The value of the `stackTop` local variable.
- Word: An integer literal.

Machine IL has a primitive type system. The only types are for integers (which are called Words are postfixed with their bit width, *e.g.* `Word8` or `Word32`), reals (which is the term SML uses for floating-point numbers), and pointers. Definitions for aggregate types such as records and arrays are not provided, as they are not dealt with directly. Instead, values of these types are handled by their pointers, and members are accessed by offsetting the pointer by a specific amount. This amount is computed by using target-specific data sizes, such as word size and pointer size, in an earlier compilation pass and represented in Machine as part of an Operand value, such as for the Offset or ArrayOffset variants.

Most statements in Machine do simple operations that are defined by the Prim structure, which is also used by the SSA intermediate languages. Prim operations typically correspond to CPU instructions that perform computation, and fall into one of the following categories:

- Arithmetic and logic operations for integer types, such as `Word_add` and `Word_lshift`. This includes arithmetic operations that can detect overflow. Overflow detection is necessary for most basic arithmetic operations as the semantics of the Standard ML language require the `Overflow` exception to be thrown if overflow occurs.
- Arithmetic operations for real types, such as `Real_sub`. There is `Real_muladd` which performs fused multiplication and addition for the SML `(*+)` operator.

- Math operations for reals values, including trigonometric functions like `Real_Math_sin` and other math library functions like `Real_Math_sqrt`.
- Comparison operations, such as `Word_equal` and `Real_lt`. These always compute a `Word32` value.
- Pointer arithmetic operations, such as `CPointer_Add` and `CPointer_diff`.
- Conversion operations for converting between words and reals, such as `Real_rndToWord` and `Word_castToReal`, and between words and pointers, such as `CPointer_fromWord`.

2.3 LLVM Design

LLVM (which originally stood for Low Level Virtual Machine) is a compiler infrastructure project that defines an intermediate representation (IR) that compilers can use to produce high-performance native code for a target platform. The IR is designed to be both language-agnostic and target-independent, which allows compilers for different languages to take advantage of sharing common functionality for back-end work such as optimizing and target-specific code generation. The LLVM project was started in 2000 by Chris Lattner as his master's thesis [13]. In 2003 it was made open source, and has been continually developed with support from companies like Apple, Google, Intel, Adobe, and Qualcomm. LLVM was the recipient of the 2012 ACM Software System Award because of its success and influence, and its high quality design and implementation. One of the first practical applications of the LLVM was the *llvm-gcc* project [14], which retrofitted a LLVM back-end on to GCC. (The `llvm-gcc` compiler is now deprecated, but the idea behind the project lives on in its successor project called *dragonegg* [5], which uses the plugin system found in the newer versions of GCC).

A major design decision for LLVM that contributes greatly to its flexibility and power is its library-oriented design. LLVM's design philosophy is to design a multitude of independent libraries that serve a specific purpose and are loosely coupled from each other,

allowing clients to easily choose just the parts they need for the features they want implemented. This also makes it easy to make a toolchain of command-line programs (shown in figure 2.3) that serve as driver programs for certain parts of the LLVM system. The LLVM back-end for MLton uses `opt`, the LLVM optimizer, and `llc`, the LLVM static compiler, to optimize and compile the LLVM IR code to either native assembly or an object file.

2.3.1 Representation

Another key feature of LLVM is that the IR has three first-class representations that can easily be converted to other forms without any loss of information. These forms are:

- A textual, human-readable assembly language. These are usually kept on disk in a file with suffix `.ll`.

- A binary format which is also usually kept on disk but is much more compact, in files with the `.bc` suffix. LLVM provides the command line tools `llvm-as` which assembles LLVM assembly to LLVM bitcode, and `llvm-dis` which disassembles LLVM bitcode into LLVM assembly.

- An in-memory symbolic representation of the IR that the LLVM libraries use for analysis, optimization, and native code generation. The libraries include functions for reading in from LLVM assembly or bitcode, and also include functionality for creating and manipulating LLVM IR directly, for use by back-ends that use the LLVM libraries directly to generate code.

2.3.2 Structure

LLVM can be described as a mid-level IR, as it offers a higher level of abstraction than typical assembly languages, but not enough to be as high-level as C. Unlike assembly languages, LLVM IR contains much more information in its code, such as a full static type system and attributes describing traits of defined functions and variables. The main purpose of this extra information is to enable more aggressive optimizations that are only possible

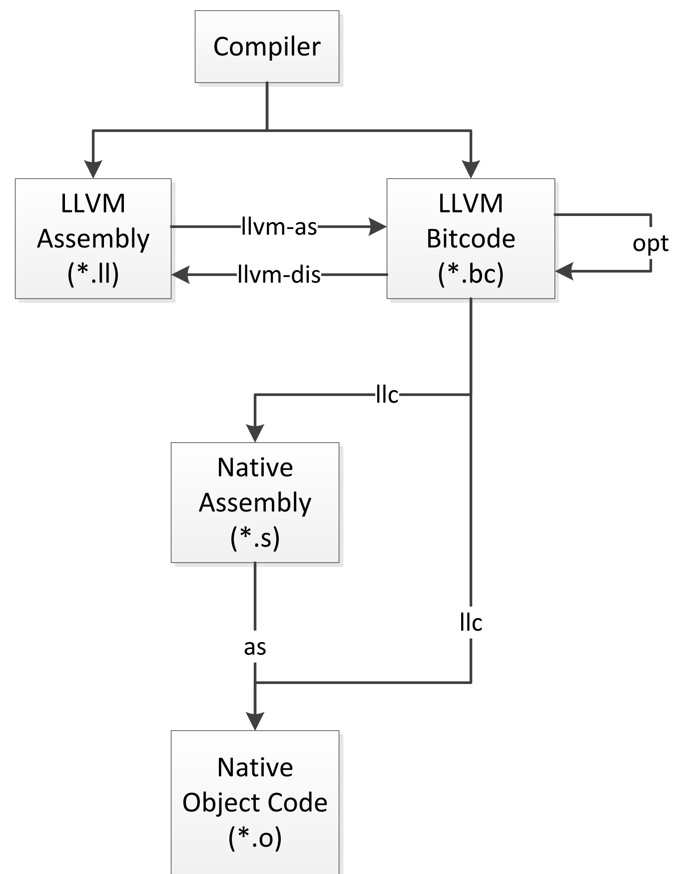

Figure 2.3: LLVM toolchain and compilation process

when this information is known. But unlike C, it lacks high-level control flow statements like “if” or “while”, instead the code is organized into a control flow graph of basic blocks.

The top level construct in LLVM IR is a Module, which describes a single translation unit. Modules contain declarations and definitions of types, global variables and functions, much like a source file would in C. LLVM functions are defined much like they are in C, with a list of parameters and a single return type (which may be `void`). Each function contains a collection of basic blocks, with one block being the designated entry block where execution starts when the function is entered. Each block has a label which identifies it and allows instructions in other blocks to reference it. Blocks contain a list of non-branching instructions, and end with a single branch instruction that can transfer control flow to one or more different blocks in the same function, or return from the function.

An important property of LLVM’s design is that its instruction set is in static single assignment (SSA) form. In SSA, instructions that produce values assign to a register that can only be written to once. SSA form helps simplify data-flow analysis, which makes it easier to implement more complex and powerful optimizations based on data-flow analysis.

Because of the SSA restriction, registers cannot directly represent mutable variables in higher level languages, as every time a variable is updated, a new register must be created. Handling these situations becomes more complex when control flow is involved. Consider compiling the pseudocode in listing 2.3 to LLVM IR. At the statement `y = x * 2`, the code generator cannot know if it should use the value of `x` that was assigned before the if statement, or inside the if block. The solution is to use a special operation in SSA instruction sets called the ϕ*-function*, implemented in LLVM as the phi instruction, which is put at the beginning of basic blocks with multiple predecessors. This instruction takes a list of value and basic block pairs, and chooses which value to assign to the register based on which basic block just executed before entering the current block. Listing 2.4 shows how the pseudocode would be compiled to LLVM IR.

Because of the SSA restriction, it can be hard for code generators to easily translate mutable variables to LLVM. Fortunately, LLVM provides a way of sidestepping this issue.

```
x = a + 1
if (cond) {
    x = x * 10
}
y = x * 2
```

Listing 2.3: Example of non-SSA pseudocode

```
bb1:
  %x = add i32 %a, 1
  br %cond, label %ifTrue, label %bb2
ifTrue:
  %x2 = mul i32 %x, 10
  br label %bb2
bb2:
  %x.phi = phi i32 [ %x, %bb1 ], [ %x2, %ifTrue ]
  %y = mul i32 %x.phi, 2
```

Listing 2.4: Code from listing 2.3 compiled to LLVM IR

Although registers in LLVM must be in SSA form, locations in memory do not. This means if the values of variables exist in a memory location, the value can be read and written to any number of times. LLVM provides an instruction, `alloca`, that allocates a space on a stack for a variable and gives a pointer to that location, and `load` and `store` instructions that read and write to a memory location given by a pointer.

2.3.3 Type System

LLVM has a type system that allows many optimizations to be performed more easily on the IR without any extra analyses. This type system by and large is based off of C's type system. There are two categories of types, primitive and derived, described in table 2.1.

LLVM has other types, such as vector types for SIMD operations and target-specific floating point types, but they are not used by the MLton's LLVM back-end.

Type	Syntax	Description
Integer	i1, i2, ..., i32, ...	Integer type of arbitrary bit width
Floating Point	float, double, ...	Floating point types of different standards
Void	void	Void type for functions that do not return a value
Label	label	Represents code labels for basic blocks
Array	[40 x i32]	Type that arranges elements sequentially in memory
Function	i32 (i8*, ...)	A function signature that consists of a return type and a list of formal parameter types.
Structure	{i32, i32, i32}	Represents a collection of data members together in memory
Opaque Structure	%t = type opaque	Represents a named structure that does not have a body specified
Pointer	[4 x i32]*	Used to specify memory locations

Table 2.1: Description of LLVM types

2.3.4 Instruction Set

LLVM's instruction set is RISC-like, with each instruction performing relatively simple operations on its operands. This section gives enough of an overview of the instructions relevant to the LLVM back-end to understand LLVM assembly and understand the design and implementation of the LLVM back-end for MLton. For a complete reference, see the LLVM Language Reference Manual [16].

Terminator Instructions

Terminator instructions are always located at the end of the block, and transfer control flow to elsewhere in the program.

- `ret`: Return to the caller function, returning a value if the current function is not `void`.
- `br`: Can be of one of two forms: unconditionally branch to a given block, or conditionally branch to one of two blocks based on an `i1` value.

- `switch`: Transfer control flow to one of any number of blocks, depending on the value of an integer. This is usually compiled to a jump table or series of conditional branches.

Binary Operations

These operations perform arithmetic computations. They take two operands which must be the same type, and compute a value which is also the same type.

- `add`, `sub`, `mul`, `udiv`, `sdiv`, `urem`, `srem`: Perform addition, subtraction, multiplication, and division on two integers. The `sdiv` and `udiv` instructions compute the unsigned or signed quotient, and `urem` and `srem` computes the unsigned or signed remainder of their operands. These instructions can also handle vectors of integers.

- `fadd`, `fsub`, `fmul`, `fdiv`, `frem`: Similar to the above operations, but for floating-point types or vectors of floating-point values.

Bitwise Binary Operations

Similar to the above binary operations, but perform bitwise computations. These instruction only take integers or vectors of integers.

- `shl`, `lshr`, `ashr`: Perform a left-shift, logical right-shift, or arithmetic right-shift operation respectively on the first operand by shifting the number of bits specified by the second operand.

- `and`, `or`, `xor`: Perform bitwise "AND", "OR", or "XOR" operations respectively on the two operands.

Aggregate Operations

These operations are for manipulating aggregate values such as arrays and structs directly. These are generally rarely used as aggregate values tend to exist in memory rather than in

registers, and are thus manipulated by memory access instructions.

- `extractvalue`: Extracts a value in an aggregate value specified by an index.
- `insertvalue`: Inserts a value into a location in an aggregate value, specified by an index.

Memory Access and Addressing Operations

These operations are for manipulating memory. Recall that in LLVM, only registers are in SSA form, so while the pointers contained in registers are immutable, all locations in memory are mutable.

- `alloca`: Allocates data of a specific type on the current stack frame, giving a pointer to that location.
- `load`: Loads the value pointed to by the given pointer operand.
- `store`: Stores a value into the memory location pointed to by the given pointer.
- `getelementptr`: Gets the address of a sub-element of an aggregate value in memory offsetting a given pointer operand with one or more indices. This operation performs address calculation only and does not access memory. The first operand is the pointer to be indexed. The second operand is an index for the pointer. Optional additional indices may follow to do further indexing if the pointer operand points to an aggregate value. This operation is generally used for getting the pointer to a member of an aggregate value to be used by a later load or store instruction, but this can be used for basic pointer arithmetic as well.

Conversion Operations

These operations convert values of one type to another, either by reinterpreting the value to be a different type (thus performing a no-op cast), or by casting it to a different type while keeping the value relatively the same by rounding.

- `truc`, `zext`, `sext`: Converts an integer to a different bit-width by truncating, zero-extending or sign-extending.
- `fptruc`, `fpext`: Converts a floating-point value to a different floating-point type by truncating or extending.
- `fptoui`, `fptosi`: Converts a floating-point value to its unsigned or a signed integer equivalent.
- `uitofp`, `sitofp`: Converts an unsigned or signed integer to its floating-point equivalent.
- `ptrtoint`, `inttoptr`: Converts values between integer and pointer types. If the integer is not the same bit-width as a pointer for the target platform, it will be truncated or zero-extended.
- `bitcast`: Converts a value from one type to another without changing any bits, meaning this is always a no-op cast. Both types must have the same bit-width. If the source type is a pointer, the destination type must also be a pointer.

Other Operations

These are operations that do not fit in any of the categories above.

- `icmp`, `fcmp`: Compares two integer or floating-point values of the same type. This operation takes an additional argument indicating what kind of comparison to do, *e.g.* `eq` for equality, `lt` for less-than. The result is a value with type `i1`, which can be used in the `br` instruction.
- `phi`: Perform the SSA ϕ-function operation, which selects a value from a list of alternatives depending on which basic block was previously executed. Phi instructions must come before all non-phi instructions in a basic block.

llvm.sqrt.*	Square root
llvm.sin.*	Sine function
llvm.cos.*	Cosine function
llvm.exp.*	Exponential function (*i.e.* e^x)
llvm.log.*	Natural logarithmic function
llvm.log10.*	Common logarithmic function
llvm.fabs.*	Absolute value function
llvm.rint.*	Round to nearest integer
llvm.fmuladd.*	Fused multiply-add
llvm.sadd.with.overflow.*	Signed addition with overflow detection
llvm.uadd.with.overflow.*	Unsigned addition with overflow detection
llvm.ssub.with.overflow.*	Signed subtraction with overflow detection
llvm.usub.with.overflow.*	Unsigned subtraction with overflow detection
llvm.smul.with.overflow.*	Signed multiplication with overflow detection
llvm.ubul.with.overflow.*	Unsigned multiplication with overflow detection

Table 2.2: LLVM intrinsic functions used by the LLVM back-end for MLton.

- `call`: Calls a function defined or declared elsewhere in the current module. Any necessary arguments depend on the function being called, and this may assign to a register if the function returns a value.

2.3.5 Intrinsic Functions

LLVM IR provides a number of *intrinsic functions*, which are functions that have well-known names and semantics and extends LLVM's capabilities without changing more fundamental aspects of the IR or any of its transformations. Intrinsic functions start with the prefix "`llvm.`" but otherwise behave the same as normal functions and must be declared before being used. The LLVM back-end uses several intrinsic functions to implement Prim operations that cannot be translated to any of LLVM's more basic instructions, mainly basic math functions that correspond to C math library functions, and arithmetic with overflow detection. The relevant intrinsic functions are listed in table 2.2. These functions are overloaded, so the asterisk is replaced by the actual type the function takes. For the math functions, it is `f32` for `float` and `f64` for `double`. For the arithmetic with overflow, it is `iN` where N is the integer size.

2.3.6 Optimizing LLVM IR

A major feature of LLVM is its ability to be optimized, both with target-independent optimizations where the IR is transformed to a more efficient but semantically equivalent version, and target-dependent optimizations which occur when translating the IR to assembly language or machine code. Optimizations are managed by having them be written as a *Pass*, which is code that performs analysis and transformations on an LLVM Module and some or all of its substructures. Passes are completely modular; they are designed to perform one type of analysis or transformation, and multiple passes can easily be applied to the same module in any order. This flexibility allows for passes to be easily added, removed, and replaced in an optimization pipeline.

LLVM passes can be applied to a module in its in-memory form by using the PassManager API, or be applied to LLVM assembly or bitcode files using the command-line `opt` tool. With `opt`, optimizations can be selected from a large collection that is included with LLVM by using a command-line flag. For example, applying the "simplifycfg" pass on prog.ll, `opt` would be invoked as `opt -simplifycfg prog.ll -o prog.ll`. Passes can also come come from a dynamic library that implements a pass by using the -load option and the library file. LLVM includes standard optimization sequences with the -ON flag, where N is a number between 1 and 3 indicating the optimization level. A higher number enables more aggressive optimizations but may take longer to run.

LLVM also supports *link-time optimization* (LTO) [17], which allows for additional optimization opportunities for programs that span multiple modules. LTO works by having a LTO-aware linker use LLVM bitcode files instead of normal object code files for linking. The idea is that with the extra information provided about a program in LLVM bitcode, the linker can perform aggressive optimizations while linking, especially ones that use interprocedural analysis, before compiling down to a native executable or library.

2.3.7 Using LLVM for MLton

Given an overview of the design of MLton and the design and features of LLVM, this project proposes that using LLVM as a back-end for MLton could bring a number of improvements to MLton's back-end design and implementation, and the quality of programs compiled by MLton.

An LLVM back-end would have several advantages over the existing C back-end for MLton. Although the C back-end is sufficient to compile any Machine IL program and grants portability for free, it suffers from some deficiencies. Because the compiler is outputting source code for what is technically another high-level language, this can causes a noticeable increase in compile times as the compiler must do operations such as parsing and type checking that has already been done on the original SML code. The C back-end is also difficult to learn and maintain, as it uses C preprocessor macros liberally to represent many common operations needed to translate from Machine IL. Additionally, the C code generated by MLton is very atypical compared to the code a human would write, thus the compiler has difficulty compiling this code quickly. In fact, the original purpose behind chunkifying was to improve compile times for very large programs by limiting the maximum amount of code that would be in a chunk. Using LLVM instead can solve many of these issues. Because LLVM assembly language is much simpler than the C language, less time can be spent parsing and type-checking. An LLVM back-end would be more maintainable as LLVM assembly is much closer to Machine IL, thus making translation more simple and straightforward.

There are also advantages that an LLVM back-end would have over a native assembly back-end, with a major one being a drastically simpler implementation. On top of the work necessary to translate Machine IL constructs to native assembly, the native code generators have to handle low-level details such as register allocation and instruction scheduling to get optimal performance. Because LLVM is higher-level than native assembly, an LLVM back-end does not have to worry about these details. LLVM's rich libraries can handle optimization and native code generation aspects without having any of the heavy lifting be

done by MLton itself.

2.4 Related work

The LLVM project has been a focus of active research and development in the compiler community, and because of its modular and reusable design, it has been easy for external projects to take advantage of the features that the project provides. Some of these projects retrofit an LLVM back-end on an existing functional language compiler similar to the goal of this project.

2.4.1 Essential Haskell Compiler

The Essential Haskell Compiler (EHC) [4] is a research compiler for the Haskell language developed at Utrecht University. Its is designed around the idea of implementing the compiler as a series of compilers. Originally it used C as a back-end language, but support for an LLVM back-end was added as part of Master's Thesis by John van Schie [27] for many of the same motivating factors of this project. Resulting benchmarks show an average of 13.1% increase in performance with respect to execution time, and a 0.8% decrease in memory usage.

2.4.2 Glasgow Haskell Compiler

The most prominent compiler for the Haskell language is the Glasgow Haskell Compiler (GHC) [12], which was first released by the University of Glasgow in 1992. David Terei implemented an LLVM back-end [23] for GHC for many of the same reasons that motivate this project. GHC's compilation process first parses the source code, and then translates it through a series of intermediate representations similar to how MLton works, although not as many. The first is *Core*, a typed lambda calculus based on System F, where many simplifications and optimizations occur. Next it is transformed to a representation for an abstract called called the *Spineless Tagless G-Machine* (STG) [11], which is similar to Core but defines an explicit execution model. The final representation before the back-end

is *Cmm*, a modified version of the *C*-- language [10], which itself is a C-like language that supports runtime services such as garbage collection and exception handling.

From the *Cmm* language, a back-end completes the compilation process by translating to a native executable. Similar to MLton, GHC has back-ends for C as well as a native code generator (NCG) which emits assembly code for either the x86, PowerPC, or SPARC architectures. A new LLVM back-end was added, which translated from *Cmm* to LLVM IR. This translation was relatively straightforward, however handling intricacies in GHC's execution model, specifically with placement of certain STG registers when calling into the runtime, required implementing a new calling convention for LLVM functions.

In the end, the new LLVM back-end had a much simpler implementation than the existing back-ends, however the performance of the generated code was comparable but not significantly better than what was compiled by the existing back-ends.

2.4.3 Erlang

Erlang is a functional programming language with a concurrency system based on the Actor model, and is useful for building fault-tolerant distributed systems. The traditional Erlang execution model is similar to that of Java, where code is compiled into a bytecode format called BEAM and executed on a virtual machine. However, in an effort to increase the runtime performance of Erlang code, a project called HiPE (High Performance Erlang) [9] was started by Upsalla University in 2001. Under this system, Erlang code is compiled to native executables instead of bytecode. This back-end has three intermediate languages: Symbolic BEAM which is a symbolic representation of the BEAM bytecode, Icode, a minimal assembly language in SSA with infinite registers, and RTL, a three-address register transfer language. RTL is then translated by the back-end to native assembly code with support for architectures such as SPARC, x86, and PowerPC. Though this system works, it is simplistic and lacks the potential performance increases by doing aggressive optimizations and taking advantage of machine-specific code generation techniques.

ErLLVM [21] is project aimed at solving these issues by translating to LLVM instead

of native assembly, and using the optimizations and portability features of the project to have the compiler emit higher-performance executables. The result was near-equal execution speed and slightly higher binary file sizes, but the designers hope that since LLVM is constantly improving, the LLVM back-end for HiPE will improve as well.

2.4.4 Other Projects

In addition to being used as a back-end for compilers of other functional programming languages, LLVM has widespread use in other compiler and interpreter projects, some of which are listed below.

- *Clang*: A compiler for C, C++ and Objective-C that uses LLVM as a back-end [2]. It features fast compile times, clear error messages, and has a static analyzer that can automatically find bugs.

- *Pure*: A modern-style functional programming language based on term rewriting. The interpreter uses LLVM as a back-end to JIT-compile Pure programs to fast native code [8].

- *MacRuby*: A Ruby implementation on top of core Mac OS X technologies. It uses LLVM for optimization passes, JIT and AOT compilation of Ruby expressions [26].

- *Rubinius*: An environment for Ruby providing performance, accessibility, and improved programmer productivity. It leverages LLVM to dynamically compile Ruby code down to machine code using LLVM's JIT [20].

- *LDC*: A compiler for the D programming language, based on the DMD front-end. It uses LLVM for its back-end for modern and efficient code generation [24].

- *Emscripten*: A project that compiles LLVM code to JavaScript, which enables programs written in languages that can compile to LLVM to run inside a web browser. This shows the flexibility that the LLVM system has, as a back-end for LLVM can not only be built for CPU architectures but other high-level languages [28].

Chapter 3

The LLVM Back-end

This chapter describes the the design and implementation the LLVM back-end for MLton. It goes over the decisions made in the design and the rationale for those decisions. It also goes over the strategies used in the implementation of the LLVM back-end to overcome challenges encountered in the project.

3.1 Code Generation Strategies

As explained in section 2.3.1, LLVM has three different representations that are isomorphic and can be converted to and from each other easily. A compiler with an LLVM back-end must choose which of the three forms to emit. The LLVM FAQ [15] offers three suggestions that correspond to the three representations:

- Call into LLVM's API to build the LLVM IR. LLVM is written in C++, however the foreign function interface for most languages can only interface to C, so LLVM provides a C library to LLVM's core to handle this situation. This is the option recommended by the LLVM project as it adapts best a JIT context and eliminates the parsing and serialization overhead for transformations as the IR is kept in memory.
- Emit LLVM assembly directly. This is generally the most straightforward choice, as the LLVM assembly language is human-readable and easy to understand. However, parsing assembly files is slower than reading bitcode when using tools such as `opt` and `llc`.

- Emit LLVM bitcode directly. This is more difficult that emitting assembly as one must deal with the intricacies of working with a binary file format, and the LLVM module structure and bitcode writer must be re-engineered as part of the code generator. However, the bitcode format has the benefits of being more compact and faster to parse when given to the LLVM tools.

All three of the options are technically feasible for this project, as MLton supports a foreign function interface (FFI) that allows SML code to call C code. An important consideration when choosing one of the three options is how it fits in to the overall MLton back-end architecture. The existing C and assembly back-ends both emit their code as text to temporary files, and rely on external compilers, assemblers, and linkers to finish the compilation process.

I made the choice to have MLton emit LLVM assembly code and rely on the external command-line tools to handle optimization and complete the compilation process. One reason for this choice is that it follows the same pattern as the existing back-ends, allowing for a unified strategy for all of MLton's back-ends. This has the additional benefit of minimizing the amount of changes needed for the rest of the MLton code, specifically in the `Main` and `Control` structures which orchestrate the entire compilation process. Another reason is that it makes for easier development when the project is a work in progress, as it is easy to catch errors in the generated LLVM assembly. When an LLVM assembly file is passed in to a program like `llc`, the module is parsed and type-checked, and if there is any apparent error, it will print out the problem and where in the file the error occurred. This strategy does suffer from the downside that it is not as fast as the alternative options, thus making compilation times longer than they could be, however MLton has relatively long compilation times regardless, and so it does not make too much a difference in the grand scheme of things.

As stated before in the background, MLton can choose which code generator is used to compile a program with the `-codegen` flag. Adding support for a new code generator is relatively simple. In the `Main` and `Control` structures, there is a variant type called

`Codegen` that defines the different code generators, and adding the new LLVM code generator means adding a new variant for LLVM and modifying all other locations where the the `Codegen` value is used.

3.2 Back-end Compilation Process

For all of MLton's code generators, including the one for LLVM, the compilation process is finished by delegating to external tools to transform the code to machine code and link it with external libraries. The process for each code generator is shown in figure 3.1. For the C back-end, all of the chunks are compiled with `gcc` to object files. For the native back-ends, `gcc` is also used for purpose of assembling the code to object files. The final stage is the linking step, where all of object files for the chunks are linked together with the external dependencies with an invocation of `gcc` that looks like the following:

```
gcc -o program program.0.o program.1.o ... \
    -lmlton -lgdtoa -lm -lgmp
```

Here, `program.0.o` is the compiled main module and all subsequent object files are the chunks are the compiled chunks outputted by the compiler or assembler. The program is also linked with a few external dependencies listed below:

- `-lmlton` is the compiled MLton runtime library.
- `-lgdtoa` is David M. Gay's floating-point conversion library [7] which is for "exact" conversions between floating point and decimal; meaning that the conversions follow the current rounding mode and produce as few digits as necessary.
- `-lm` is the C standard math library.
- `-lgmp` is the GNU Multiple Precision Arithmetic Library [6], which powers the underlying computation for SML's IntInf types.

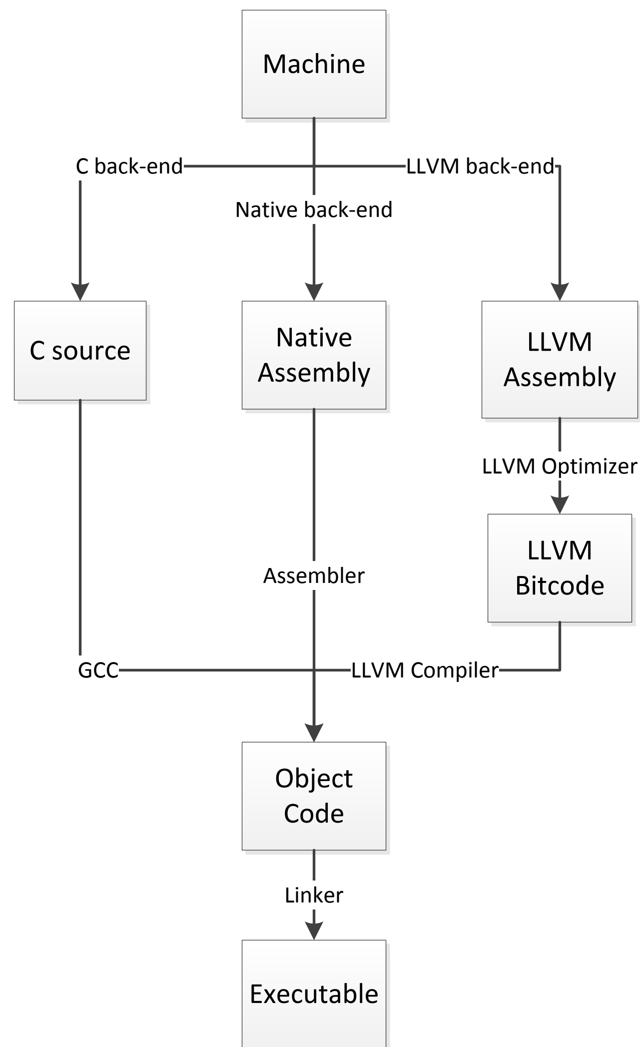

Figure 3.1: MLton Back-end Pipeline

```
signature LLVM_CODEGEN =
sig
    include LLVM_CODEGEN_STRUCTS

    val implementsPrim: 'a Machine.Prim.t -> bool
    val output: {program: Machine.Program.t,
                 outputC: unit -> {file: File.t,
                                   print: string -> unit,
                                   done: unit -> unit},
                 outputLL: unit -> {file: File.t,
                                    print: string -> unit,
                                    done: unit -> unit}
                } -> unit
end
```

Listing 3.1: Signature for MLton's LLVM code generator

3.3 LLVM Code Generation

This section describes the design and implementation details of MLton's LLVM code generator. The bulk of the work for this project is done in the `LLVMCodegen` structure, which provides a couple of high-level functions to the rest of the compiler which does code generation of the Machine IL program and outputs it to two or more files.

3.3.1 Signature and Structure

To adhere with the module system that Standard ML provides and the design used by existing MLton components, the code for LLVM code generation is split between a *signature*, which defines the interface of the module that can be called from outside the module, and a *structure*, which contains the implementation of the module. The signature for the LLVM back-end, which is called `LLVM_CODEGEN`, follows the same pattern as the existing back-ends by defining two functions, given in listing 3.1.

The first function, `implementsPrim`, is used by the SSA-to-RSSA conversion pass earlier in the compilation process to decide which `Prim` operations should appear directly in the Machine IL. The `Prim` type includes operations which may not be implementable directly with a particular code generator, such as 64-bit arithmetic on a 32-bit platform, or

certain math operations not supported by LLVM intrinsics in the LLVM back-end. If the function returns false for Prim operation, the operation is translated to a CCall to a function contained in the runtime. This provides an implementation for that operation, but at the overhead cost of a function call. In the LLVM back-end, most of the Prim operations are implemented, with the math operations using the equivalent LLVM intrinsic functions.

The second function, `output`, is called once by the compiler, and this is the function that does the translation from Machine to LLVM. The function passes in a record with three members which provide the code generator with all necessary data and functionality to output to files. The first member, `program`, is a Machine IL program, which as discussed before, is the top level structure in Machine and contains all other program data. The remaining two members, `outputC`, and `outputLL` are closures that, when called, return a record containing data and closures used for outputting text to a file. The record has three members: `file`, which is an SML file descriptor type, `print`, which is used for writing to the file, and `done`, which signals that output for the current file has finished and closes the file. The output closures can be called multiple times, and each time they return output operations for a different file. This is what allows different chunks to be written to different files. The difference between the two output functions is the kind of source file it opens: `outputC` is used for writing the main module, which must always be written as C code regardless of code generator, and `outputLL` is used for writing the translated Machine IL code in LLVM assembly.

3.3.2 Code Generation Strategy

The translation of a Machine program to LLVM in `LLVMCodegen` is handled by several functions which handle the translation of each major Machine IL component. These are supported by utility functions that handle common tasks in code generation.

As decided in section 3.1, the choice was made for the new back-end to emit LLVM assembly. However, since LLVM is a highly structured language, care needs to be taken in the design to ensure the assembly emitted is always correct and able to be accepted by the

LLVM tools. There are two ways this issue can be handled:

- Implement much of LLVM's type hierarchy for representing LLVM code (Modules, Functions, BasicBlocks, Instructions, etc.) as SML types, and have the code generator build up a symbolic representation of the LLVM code with those types. At the end, the symbolic representation would be pretty-printed to text which can be done by having a `toString` function associated with each type. The advantage of this way allows code generation to focus entirely on the high-level and symbolic translation aspects and not have to deal with things like string manipulation to build assembly code. It also makes code generation easier when the translation is not as direct.

- Translate Machine IL constructs directly into LLVM assembly and be directly outputted to the file. This way is much more straightforward as there is no need to re-engineer the LLVM type hierarchy for the code generator, but exposes issues like assembly syntax and string manipulation directly to the code generator.

In the end, I ended up choosing the second way, as Machine IL and LLVM are similar enough in structure that the advantages of the first way did not matter too much. Also, this is the method that the existing C code generator uses, so there is already proof that this method works in practice.

3.3.3 Translating Machine IL to LLVM

This section describes the design and implementation of the important functions defined in LLVM that drives the code generation process.

Chunks

As stated before, a Machine IL program is divided into one or more *chunks*, each of which contain the implementation of one or more SML functions. Because of the trampolining aspect of MLton's execution model, the C back-end translates each chunk into a C function,

and so the LLVM back-end does the equivalent and generates an LLVM function for each chunk.

At the beginning of a chunk, the code generator must allocate space for variables whose scope is throughout the entire chunk. This includes local caches of the `gcState` members `frontier` and `stackTop`, the `nextFun` value used by the switch statement to jump to the currently executing function, and all of the Machine registers that were created for this chunk. All of these variables are mutable in Machine, which means that they cannot map directly to LLVM registers due to the SSA restriction. Instead, these variables are put on the stack with the `alloca` instruction, and because memory is not in SSA form, they can read and written to any number of times with the `load` and `store` instructions. This is not the ideal solution for performance, as it is always faster to access registers than to access memory. LLVM can mend this inefficiency with the `mem2reg` optimization pass, which promotes memory locations to registers and inserts `phi` instructions wherever necessary to enforce the SSA constraints.

Blocks

Machine blocks map directly to LLVM basic blocks, so this aspect of code generation is relatively simple and straightforward. The code generator just needs to translate each statement to its equivalent in LLVM, and then translate the transfer. However, for blocks that execute right after a ML function call or return (*i.e.* have kind `Cont`), the code generator must emit stack manipulation code at the beginning of the block to push or pop a stack frame.

Statements

Statements in Machine are also straightforward to handle in code generation. As described in section 2.2.6, a Machine statement is one of only four variants, two of which are trivial or ignored. Move statements have two Operands, and the code generator translates the Operands and stores the source value into the destination. PrimApp statements are for Prim

operations, and they are translated to one or more LLVM instructions or calls to intrinsic functions.

Transfers

Some transfer instructions simply map to a terminator instruction in LLVM, but some do not as they perform some kind of computation and then branch depending on the result of that computation. Also, because LLVM cannot implicitly manipulate the SML stack the way it does the C stack, the inter-chunk transfers have some stack manipulation. A description of how the code generator handles each kind of transfer function is listed below.

- `Arith`: This instruction performs a Prim instruction, specifically an arithmetic computation with overflow checking, and does a conditional branch to one of two labels, depending on whether an overflow occurred. This is done by calling one of the overflow-checking intrinsic functions described in table 2.2. These functions return a struct containing the computed value and a value of type `i1`, which is 1 if overflow occurred. The result of the Arith operation is stored into its destination Operand, regardless of outcome, and a conditional branch (`br`) is used on the `i1` value.

- `CCall`: This instruction calls a C function defined by the MLton runtime. In addition to generating code for the operands to the function and doing the call with a `call` instruction, the code generator needs to cache and flush the `frontier` and `stackTop` when calling a function that may manipulate them (*e.g.* garbage collection), as their values will be changed in `gcState`. Lightweight threads are supported, and when the runtime switches threads, upon returning to the SML/chunk code, the code jumps to the top of the chunk in order to trampoline to the block of the switched-to thread, otherwise it does an unconditional branch to a given label.

- `Call`: This transfer instruction is for performing a call to another SML function. It pushes a frame on the stack unless it is a tail call, in which case it reuses the current stack frame. If the function is in another chunk, it stores the chunk number in the

local `cont` variable which will be return by the chunk function, and updates the global `nextFun` variable which is used by the switch statement in the next chunk to jump the function being called.

- `Goto`: This transfer instruction just does an unconditional branch to a block with the `br` instruction.

- `Raise`: This transfer instruction is for raising an exception. This pops off all stack frames up to the nearest enclosing exception handler by setting `stackTop` to `stackBottom` plus the `exnStack` member in `gcState`. It then loads a "return address", which is really a label index, from the current stack frame, stores it into the `l_nextFun` local variable, and then does an unconditional branch to the top of the chunk.

- `Return`: This is for returning from an SML function. Like `Raise`, it loads the label index from the stack into `l_nextFun`, but only pops off one stack frame. If the label index is a different chunk, the default case of the switch statement will return to the trampoline.

- `Switch`: This is a simple conditional branch operation that transfers control to one of two or more possible basic blocks, based on the value of an integer. If there are exactly two blocks that can be branched to, and the values the condition variable must match are 0 and 1, then the switch can be simplified to a conditional `br` instruction. Otherwise, it is translated to an LLVM `switch` instruction.

Operands

Operands are the least straightforward part of Machine to translate to LLVM, for two main reasons: first, they are the only type in Machine that is recursively defined. That is, some variants of Operand are defined in terms of another Operand. This has the effect of making the code that translate Operands recursive as well. Second, certain operands may be either r-values or l-values (*e.g.* offsets of a pointer) whereas others may only be r-values (*e.g.*

word or real literals). What this means is the actual meaning of an Operand depends on the context in which it is used. To give an example, take the following code:

```
a[i] = b[i];
```

Although the expressions on both sides of the assignment look the same, the actual operation being done differs: on the right, a value is being loaded from memory, and on the left, a value is being stored into memory. It is apparent that this aspect of Machine was designed with a C back-end in mind, as the C code generator simply exploits this l-value/r-value duality in the C language when translating Operands, however this cannot be the case in the LLVM code generator as LLVM IR has explicit loads and stores.

In the LLVM code generator, this means that the function that translates Operand values needs some kind of additional context to know if the Operand will be read or written to. This issue can be solved in a few ways:

- The function receives an additional parameter indicating if the operand is an l-value or r-value. If it is an l-value, it will return a pointer which the caller will use for building a store instruction, otherwise, it returns the value that the operand means to reference. This method somewhat convolutes the implementation of the function by having it require an extra argument.

- The function always returns a pointer to the Operand, and the caller either loads or stores with that pointer depending on what the context the Operand is needed for. For literal values such as words and reals, they can be put in a stack-allocated location with the `alloca` instruction and the caller can load from that. Although its possible for the unnecessary `alloca` instructions to be optimized away by LLVM's optimizer, this method greatly convolutes the generated code and makes it spend unnecessary time optimizing.

- The function for translating operands can instead be two mutually recursive functions: one for getting the address of an operand and one for getting the value. The

caller decides which function to call depending on what the operand is used for. This method is similar to the first, but does not force the functions to take an extra parameter.

The solution chosen was the third, with the code generator using two mutually recursive functions: `getOperandAddr` and `getOperandValue`. Since not all operands (*e.g.* literals) could have an address, the `getOperandAddr` function would cause a runtime error in the compiler if it received a variant it could not handle, but that never happens in practice as the compiler never generates Machine IL code where that could occur. Also, for the variants of Operand that can have an address, the implementation of `getOperandValue` simply calls `getOperandAddr` with that operand and adds a load instruction, to simplify its implementation and minimize redundancy.

3.3.4 Main Module and Label Indices

As described before, there needs to be a main chunk for every MLton program to be the entry point of the program and the module where all the global data shared with all the chunks is defined, do all of the necessary boilerplate and setup for the runtime, and invoke the chunk functions via the trampoline. Because the existing C code generator can already handle generating a main chunk, it exports a function called `outputDeclarations` in the `CCodegen` signature, which is used by `LLVMCodegen` to generate the main chunk.

One other important aspect in back-end is the generation of label indices. As described before, an unique integer known as a label index is assigned to each block that can be jumped to via the chunk's switch statement. This includes blocks that are the entry point to SML functions as well as blocks where control flow returns to after a called SML function returns. In Machine IL, blocks that have a label index (which are known as "entry labels") have a `FrameInfo` property containing that value, which comes from the translation from RSSA to Machine.

3.4 Other Stages

This section describes the rest of the compilation phases in the LLVM back-end to translate the program down to object code.

3.4.1 LLVM Optimizer

After the module is compiled to a *.ll file, it is optimized by the LLVM optimizer tool, which is called `opt`. This tool is used with one of the standard optimization sequences (`-O1`, `-O2`, or `-O3`), which is given by the user when invoking MLton. It also applies the `-mem2reg` pass which promotes stack-allocated variables to registers. The result is an LLVM bitcode file (*.bc) with the optimized LLVM code.

3.4.2 LLVM Compiler

The bitcode file is compiled down to object code by invoking `llc`, the LLVM compiler. This tool can also compile to assembly as well, but doing that would require an extra step of invoking the assembler to get object code.

Chapter 4

Evaluation

This chapter evaluates the implementation of the LLVM back-end for MLton by comparing it to the existing C and native back-ends, based on qualitative and quantitative analysis of the implementation complexity, and quantitative analysis of the performance of executables generated by MLton's LLVM back-end.

4.1 Implementation Complexity

In this section, the complexity of implementation of the LLVM back-end will be compared to the existing C and native back-ends. The easiest and simplest way to do this is to compare the code size of the different back-ends by measuring *source lines of code* (SLOC). The measurements are shown in table 4.1. Note that the code size for the LLVM back-end is "bloated" by a large amount of boilerplate string data (specifically, declarations of types and intrinsic functions). Also, the C back-end relies on a number of header files that define macros used by the code generator to create simplified representations of common operations.

Back-end	**Size**
C	1226 over 2 files (+ header files)
x86	28544 over 29 files
amd64	26139 over 28 files
LLVM	1557 over 2 files

Table 4.1: SLOC comparisons of different back-ends

CPU	Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz
RAM	8GB
OS	Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-49-generic x86_64)
MLton version	20130715
GCC version	4.6.4
LLVM version	3.3

Table 4.2: Specifications of the computer running the benchmarks

4.2 Executable Quality

For quantifiable metrics about the executables generated by the compiler, MLton comes with a benchmark suite which measures compilation time, running time, and file size of executables compiled from a wide array of benchmark programs. These are programs that perform computationally intensive tasks such as fast-Fourier transforms or ray tracing.

For analysis of the quality of code emitted by the different back-ends, the benchmark suite was run on a computer with specifications described in table 4.2. Also, the way chunkifying a program is done can have a noticeable effect on compilation times and execution times, especially for larger programs. MLton has three different chunkify strategies that can be selected with the `chunkify` flag: *OneChunk* puts all of the code into one chunk, *ChunkPerFunc* creates a chunk for each function, and *Coalesce* limits the size of a chunk and puts the code into as few chunks as possible, grouping functions that call each other frequently into the same chunk to minimize the trampoline overhead. For these benchmarks, the native codegen used the *Func* strategy, the C codegen used *Coalesce* with limit 4096, and the LLVM codegen used *Coalesce* with limit 16384.

4.2.1 Code Size

Code size measures the file size of the resulting executable in bytes, with smaller being better. Note that file sizes for MLton-compiled executables are relatively large compared to the executables that come from compilers for other languages, even for small and trivial programs, as MLton always compiles the full ML Basis Library with the program, and the runtime is statically linked with every executable. The results are shown in table 4.3, with

the smallest measurement for each benchmark in bold.

The results show that the code sizes emitted by the back-ends are generally close, however the LLVM back-end was the smallest in all but two of the benchmarks. This shows a clear advantage of using the LLVM back-end, at least on the amd64 platform.

4.2.2 Compilation Time

Compilation time measures the time it takes for MLton to compile a program from invocation to executable generation. As mentioned before, MLton compiles the entire ML Basis library for every program, meaning compile times are always at least a few seconds even for trivial programs. However, the main goal of MLton is to aggressively optimize its programs the best that it can, even at the sacrifice of compilation times. Still, however, it is good to have compile times to be as small as possible. The compilation times for the programs in the benchmark suite are shown in table 4.4.

It turns out that the native (amd64) code generator is the clear winner for every benchmark. This is not too much of a surprise, as from a design perspective, translating directly from Machine to assembly is the shortest path to getting native code. Looking at the LLVM back-end's performance, it is not too much slower that the other back-ends, and it beats the C code generator in many of the benchmarks. This demonstrates a hypothesis from the beginning that, because it takes less computational effort to compile and optimize a high-level assembly language like LLVM to machine code than a high-level programming language like C, the LLVM back-end will result in shorter compilation times than the C back-end.

It is important to keep in mind that the compilation time for larger, non-trivial programs can vary greatly depending on the chunkify strategy used. As mentioned before, the purpose of partitioning SML code in to chunks is to keep files emitted by the code generator a reasonable size, otherwise it will take very long for an external compiler or optimizer to process a large program if all of its code is put in a single translation unit. The trade-off is in run-time performance, as programs split into more chunks will have to jump back to the trampoline more often.

Benchmark	**amd64 back-end**	**C back-end**	**LLVM back-end**
barnes-hut	178,552	181,200	**172,624**
boyer	247,644	251,036	**229,588**
checksum	115,340	122,668	**114,500**
count-graphs	143,564	148,380	**136,500**
DLXSimulator	214,087	224,391	**207,439**
fft	142,521	148,483	**132,134**
fib	115,180	114,812	**114,468**
flat-array	114,956	114,636	**114,100**
hamlet	1,447,923	1,540,699	**1,407,795**
imp-for	114,988	114,492	**114,148**
knuth-bendix	190,183	198,831	**184,159**
lexgen	299,078	332,974	**281,774**
life	138,972	136,492	**135,700**
logic	196,012	192,564	**179,460**
mandelbrot	115,036	118,300	**114,196**
matrix-multiply	117,356	120,252	**115,748**
md5	148,183	157,623	**146,143**
merge	116,668	123,724	**115,668**
mlyacc	663,622	701,270	**638,470**
model-elimination	831,284	893,044	**776,676**
mpuz	121,660	129,356	**120,052**
nucleic	289,107	**259,547**	283,931
output1	156,043	164,411	**154,355**
peek	152,903	160,231	**151,167**
psdes-random	119,404	118,876	**117,684**
ratio-regions	142,812	152,084	**138,644**
ray	255,258	267,682	**236,594**
raytrace	385,806	394,002	**326,918**
simple	351,027	380,324	**335,483**
smith-normal-form	312,119	**305,951**	329,295
tailfib	115,036	114,668	**114,228**
tak	115,164	114,780	**114,500**
tensor	180,774	188,750	**172,558**
tsp	160,398	167,172	**149,789**
tyan	228,343	242,215	**215,647**
vector-concat	116,652	116,044	**115,716**
vector-rev	116,620	116,172	**115,460**
vliw	520,276	604,252	**494,428**
wc-input1	182,469	191,709	**178,541**
wc-scanStream	192,485	204,477	**186,957**
zebra	228,215	234,791	**208,079**
zern	151,328	155,941	**141,293**

Table 4.3: Code size results of benchmarks in bytes

Benchmark	amd64 back-end	C back-end	LLVM back-end
barnes-hut	**2.76**	3.26	3.06
boyer	**2.92**	4.14	4.32
checksum	**2.18**	2.26	2.24
count-graphs	**2.39**	2.73	2.61
DLXSimulator	**2.92**	3.95	3.92
fft	**2.31**	2.49	2.42
fib	**2.18**	2.24	2.22
flat-array	**2.18**	2.25	2.23
hamlet	**12.34**	25.33	38.50
imp-for	**2.19**	2.26	2.24
knuth-bendix	**2.65**	3.49	3.32
lexgen	**3.36**	4.99	5.99
life	**2.31**	2.60	2.49
logic	**2.68**	3.47	3.46
mandelbrot	**2.20**	2.26	2.24
matrix-multiply	**2.20**	2.30	2.27
md5	**2.38**	2.80	2.59
merge	**2.18**	2.26	2.24
mlyacc	**7.10**	12.49	14.11
model-elimination	**6.85**	12.98	21.73
mpuz	**2.23**	2.36	2.31
nucleic	**3.76**	9.55	4.88
output1	**2.39**	2.89	2.65
peek	**2.38**	2.84	2.62
psdes-random	**2.22**	2.32	2.28
ratio-regions	**2.52**	2.88	2.79
ray	**3.09**	4.34	4.76
raytrace	**4.12**	6.28	7.68
simple	**3.52**	5.54	7.00
smith-normal-form	**3.20**	17.71	8.21
tailfib	**2.19**	2.26	2.23
tak	**2.22**	2.29	2.27
tensor	**2.78**	3.50	3.25
tsp	**2.45**	2.94	2.68
tyan	**3.02**	4.24	4.39
vector-concat	**2.18**	2.27	2.25
vector-rev	**2.19**	2.27	2.25
vliw	**5.12**	8.81	11.94
wc-input1	**2.60**	3.35	3.13
wc-scanStream	**2.66**	3.44	3.31
zebra	**3.01**	4.12	4.10
zern	**2.34**	2.61	2.50

Table 4.4: Compilation time results of benchmarks in seconds

4.2.3 Execution Time

The final and perhaps most important measurement of code quality from a compiler is the execution time of the programs it compiles. The execution times of the benchmark programs for MLton's various back-ends are shown in table 4.5.

The results show that the best execution times for a program are varied among the different back-ends. The amd64 codegen had the best performing executable for 13 of the 42 benchmarks, the C codegen had the best for 7, and the LLVM codegen had the best for 26. This means the majority of the benchmarks run the fastest when compiled with the the LLVM code generator. This fulfills one of the most important goals of this project, which is that utilizing LLVM can increase the performance of generated executables compared to the existing code generators. There is still room for improvement though, as there are still many benchmarks where one of the existing code generators produce the fastest executable.

Somewhat surprising is that for some benchmarks, the version compiled with the C back-end is the fastest, and for many of the others it is not much slower. When the native back-ends were first implemented, they generated substantially faster code that what was compiled with the C back-end, but this is no longer the case, as over the years GCC has improved quite a bit.

One fascinating result is the LLVM back-end's execution time for the *flat-array* benchmark (reproduced in listing 4.1), which is effectively instant. This is not deemed to be a bug, as when it is compiled with LLVM with no optimization, the program takes several seconds to run. Rather, what I suppose is happening is that LLVM is doing optimizations like constant propagation and loop unrolling so aggressive that it ended up computing the result of the call `loop` at compile time, thus skipping pretty much all of the computational work. This shows the power of some of LLVM's optimizations and that utilizing LLVM in a compiler can bring about some great benefits.

Benchmark	amd64 back-end	C back-end	LLVM back-end
barnes-hut	3.37	**3.33**	3.46
boyer	14.77	14.82	**14.74**
checksum	5.75	17.75	**4.27**
count-graphs	6.56	**6.33**	6.69
DLXSimulator	6.01	6.00	**5.75**
fft	3.26	3.18	**2.88**
fib	**13.93**	18.91	18.70
flat-array	6.42	12.73	**0.00**
hamlet	**9.79**	21.07	14.36
imp-for	9.63	9.53	**7.96**
knuth-bendix	**5.91**	7.76	6.67
lexgen	5.29	4.89	**4.50**
life	**7.07**	7.39	7.49
logic	**6.22**	6.50	6.70
mandelbrot	12.80	11.52	**3.55**
matrix-multiply	4.69	**3.11**	4.15
md5	12.83	46.70	**12.78**
merge	9.99	10.08	**9.98**
mlyacc	**5.83**	6.51	7.11
model-elimination	**10.17**	14.82	15.32
mpuz	6.25	9.29	**4.94**
nucleic	3.99	**3.74**	3.76
output1	**8.26**	8.58	9.49
peek	16.26	**7.87**	**7.87**
psdes-random	7.41	7.18	**5.05**
ratio-regions	20.78	22.30	**20.31**
ray	2.61	2.65	**2.41**
raytrace	**3.81**	3.93	4.12
simple	**4.43**	7.45	5.79
smith-normal-form	**1.29**	**1.29**	**1.29**
tailfib	13.70	13.73	**8.24**
tak	**7.00**	7.78	8.72
tensor	17.43	14.56	**11.58**
tsp	9.43	6.67	**6.25**
tyan	**6.76**	7.28	7.26
vector-concat	10.91	11.42	**4.33**
vector-rev	10.05	9.13	**8.16**
vliw	6.11	7.59	**5.89**
wc-input1	6.84	**6.72**	**6.72**
wc-scanStream	7.43	9.00	**6.02**
zebra	7.85	8.02	**6.70**
zern	5.31	6.88	**4.88**

Table 4.5: Execution time results of benchmarks in seconds

```
structure Main =
struct
fun doit n =
    let
        val v = Vector.tabulate (1000000, fn i => (i, i + 1))
        fun loop n =
            if 0 = n
            then ()
            else
                let
                    val sum = Vector.foldl (fn ((a, b), c) =>
                              a + b + c handle Overflow => 0) 0 v
                in
                    loop (n - 1)
                end
    in
        loop n
    end
end
```

Listing 4.1: Source of the *flat-array* benchmark

4.3 Summary

The analyses shown in this chapter show support for the hypotheses given at the beginning of this report, that LLVM would be a feasible back-end target for the MLton compiler. To give answers to the questions proposed by the hypotheses:

- *How complex is the implementation of the LLVM back-end compared to the existing back-ends, with respect to code-size and simplicity of design?*

  The complexity in terms of the amount of code needed to implement an LLVM back-end is shown to be substantially less then either of the native code generators, and also less than the C code generator when taking into account C header files that are included. The code generator is also reasonably easy to learn and should be maintainable enough, given the relatively straightforward mapping of Machine to LLVM as described in chapter 3.

- *How much does the new back-end affect MLton's overall compilation times?*

For most programs as demonstrated by the benchmark suite, there is not much of a noticeable impact on compilation times compared to the other code generators, and it even gives faster compilation times than using the C back-end on many programs.

- *How good is the quality of executables generated the different back-ends, based on file size and running time of various benchmark programs?*

  The benchmark suite showed that for most programs, the executable sizes and running times of programs compiled with LLVM is either the best out of all the back-ends, or close to the versions compiled with either of the other code generators. This is generally considered the most important criteria from the perspective of MLton's users, and the fact that LLVM yields the best executables based on the simple and straightforward implementation detailed in this report shows the overall quality of LLVM system and the amount of positive impact it can have for a compiler back-end.

It is important to consider that the benchmarks used in this chapter only tested on one architecture. Although it is one of the more commonly used architectures, a more complete evaluation would have benchmarks for other architectures, including ones that the native code generator does not support, as one of the features of LLVM is that it is architecture-independent and can support a multitude of platforms.

Chapter 5

Conclusions

This report described the motivations, design, and implementation of a new LLVM back-end for the MLton SML compiler. The primary goal of the project was to see if leveraging the benefits an LLVM back-end could bring to a compiler would result in a new back-end design that was much simpler yet resulted in outputting excellent quality code, and was not tied to one specific architecture.

Background information was presented on the broad design of MLton, its compilation pipeline, and execution model, to give understanding as to where LLVM would fit in the picture and what kind of work would have to be done. The design and implementation of the LLVM back-end was then presented, showing all of the challenges that were faced in its implementation and how they were overcome.

After observing the benchmark results of code generated from the new LLVM back-end compared to code generated by the existing back-ends, the LLVM back-end is deemed to be a viable alternative as it produces code that is comparable if not superior to what is produced by the existing code generators. Given that it can produce better code than the C code generator with faster compilation times, along with a small implementation and being portable to any of the many architectures that LLVM supports, it has the potential of being MLton's primary code generator moving forward.

5.1 Future Work

Although the project met many of the goals that it was set to accomplish, there is still plenty of room for future improvement, as shown by certain benchmark results where LLVM performs worse than the other code generators. These opportunities stem from the relatively simple implementation of the LLVM back-end and that it may not be taking full potential of the optimization opportunities that LLVM offers.

A paper on improvements to GHC's LLVM back-end [3] details ideas that could be applied to MLton's LLVM back-end. One of these is the use of alias analysis to improve operations that deal with memory. Memory in LLVM IR does not have types, so it cannot be used for type-based alias analysis (TBAA) as it is. To solve this, metadata is added to certain nodes to describe the type system for a higher-level language and enable more detailed alias analysis for more aggressive optimizations.

Other potential improvements include using LLVM function tail calls when switching to a different chunk, which can avoid the overhead of jumping back to the trampoline. Also, the compiler could utilize LLVM link-time optimization (LTO) features [17], which applies interprocedural optimizations at link-time when linking together translation units that are in LLVM bitcode format. Because the MLton runtime and the libraries it uses are written in C, they can be compiled to LLVM bitcode with clang which will allow all parts of an executable to be linked together with LTO.

Bibliography

[1] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Programming Languages and Systems*, pages 56–71. Springer, 2000.

[2] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[3] Robert Clifton-Everest. Optimisations for the LLVM back-end of the Glasgow Haskell Compiler. Bachelors Thesis, Computer Science and Engineering Dept., The University of New South Wales, Sydney, Australia, 2012.

[4] Atze Dijkstra, Jeroen Fokker, and S Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. In *Implementation and Application of Functional Languages*, pages 57–74. Springer, 2008.

[5] DragonEgg - Using LLVM as a GCC backend. `http://dragonegg.llvm.org/`.

[6] Free Software Foundation. The GNU MP Bignum Library. `http://gmplib.org/`.

[7] David M. Gay. gdtoa Library. `http://www.netlib.org/fp/`.

[8] Albert Gräf. Pure Programming Language. `http://purelang.bitbucket.org/`.

[9] E. Johansson, M. Pettersson, and K. Sagonas. A high performance erlang system. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 32–43. ACM, 2000.

[10] S. Jones, N. Ramsey, and F. Reig. C-: A portable assembly language that supports garbage collection. *Principles and Practice of Declarative Programming*, pages 1–28, 1999.

[11] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *J. Funct. Program.*, 2(2):127–202, 1992.

[12] S.L.P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93. Citeseer, 1993.

[13] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[14] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[15] LLVM Frequently Asked Questions (FAQ). `http://www.llvm.org/docs/FAQ.html`.

[16] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`.

[17] LLVM Link Time Optimization: Design and Implementation. `http://llvm.org/docs/LinkTimeOptimization.html`.

[18] Robin Milner. *The definition of standard ML: revised*. The MIT press, 1997.

[19] MLton. `http://mlton.org`.

[20] Evan Phoenix. Rubinius. `http://rubini.us/`.

[21] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. ErLLVM: an LLVM backend for Erlang. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, Erlang '12, pages 21–32, New York, NY, USA, 2012. ACM.

[22] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ml to c. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(2):161–177, 1992.

[23] David A. Terei and Manuel M.T. Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM.

[24] The LLVM-based D compiler. https://github.com/ldc-developers/ldc.

[25] The LLVM Compiler Infrastructure Project. http://www.llvm.org.

[26] MacRuby. http://macruby.org/.

[27] John van Schie. Compiling Haskell to LLVM. *Master's thesis, Department of Information and Computing Sciences, Utrecht University*, 2008.

[28] Alon Zakai. Emscripten. https://github.com/kripken/emscripten/wiki.