CDC HotSpot Implementation Dynamic Compiler Architecture Guide

Connected Device Configuration, Version 1.1.1

Foundation Profile, Version 1.1.1

Optimized Implementation

Sun Microsystems, Inc.
www.sun.com

August 2005

Contents

Code Samples

CHAPTER **1**

Dynamic Compiler Architecture

1.1 Introduction

This document describes the internal workings of the CDC-HI dynamic compiler. It is assumed that the reader is already familiar with Chapter 5, Dynamic Compiler, of the CDC Porting Guide, and section 4.3, Tuning Dynamic Compiler Performance, of the CDC Runtime Guide.

The CDC-HI Dynamic Compiler, also called a just-in-time compiler (JIT), dynamically converts Java bytecodes to native code during execution of a Java application. Compilation is per-method, meaning that a single method of a class is compiled at a time.

Compilation is triggered based on the specified compilation policy. The compilation policy determines how soon a method is compiled based on number of invocations and backwards branches. See the Dynamic Compilation Policy section of the CDC Runtime Guide for details on how to specify the compilation policy.

The JIT supports on stack replacement (OSR) of executing methods. This means that while a method is being interpreted, it can be compiled and then have execution resume in the compiled method. In order to simplify support for OSR, it is only triggered on backwards branches, and only when there are no values on the Java evaluation stack.

The JIT has two passes: a front end that converts Java bytecodes to an intermediate representation (IR), and a backend that converts the IR to native code. A build tool called JCS produces code used by the backend to parse the IR tree. It is similar to Yacc in both its purpose and syntax. There is also a large amount of runtime code to support compiled methods, such as for object allocation and invocations.

1.2 Compiler Front End – IR Creation

The goal of the front end is to convert the Java bytecodes to an intermediate representation (IR), which takes the form of a DAG (directed acyclic graph). For example:

```
x = y + 1000;
```

Produces the following bytecodes:

```
iload y

sipush 1000

iadd

istore x
```

The following represents the IR tree for this assignment statement:

```
          ASSIGN

          /   \

    LOCAL(X)  ADD

             /  \

       LOCAL(Y) CONSTANT(1000)
```

This shows an assignment (the ASSIGN node) to the local x (the LOCAL node on the lhs of the ASSIGN node) of the result of an add (the ADD node on the rhs of the ASSIGN node). The ADD node shows the adding of the local y (the LOCAL node on the lhs of the ADD node) to the constant 1000 (the CONSTANT node on the rhs of the add node).

The front end also handles other tasks, such as verifier and security checks usually made when the interpreter quickens opcodes, plus numerous optimizations on the IR, such as method inlining and null check eliminating. This is all covered in more detail in the JIT Intermediate Representation chapter.

1.3 Compiler Back End – JCS and IR Parsing

The back end is responsible for converting the IR generated by the front end into native instructions. To do this, the IR tree must first be parsed. This is handled by a parser produced by the JCS tool at build time. As input it takes files that contain the grammar of the IR, the syntax of which is also very similar to that of Yacc. For example, the following grammar rule could be used to describe the ADD IR node:

```
reg32: IADD32 reg32 aluRhs
```

Note – The ADD node has been renamed to IADD32. This is because of a transformation done in the back end, which will be explained in more detail later.

In this example, the rhs rule is an IADD32 node with a reg32 node its lhs and aluRhs node on its rhs, and it produces a reg32 node. This above rule can be viewed as converting an IADD32 node to a reg32 node, so it can then be consumed by other rules requiring a reg32 node.

A reg32 node represents an evaluated IR expression, either in memory or in a register. In the case of the example IR above, it would be the local `y`. An aluRhs node is either an evaluated IR expressions (a reg32) or an immediate value that can possibly be used in a native instruction without loading it into a register first. In the example IR above, it would be the constant 1000.

More details on JCS and grammar rules can be found in the `JavaCodeSelect` chapter.

1.4 Compiler Back End – Semantic Actions

Most JCS rules have semantic actions (in the form of C code), and this is where code generation takes place. For example, the semantic action for the IADD32 rule would contain something like the following:

```
CVMCPUemitBinaryALU(con,
CVMCPU_ADD_OPCODE,
CVMRMgetRegisterNumber(dest),
CVMRMgetRegisterNumber(lhs),
CVMCPUalurhsGetToken(con, rhs));
```

This is only a small part of the semantic action for the IADD32 rule. There is a lot of other bookkeeping work that is also done, but this is where code is actually emitted. An emitter function is called, specifying that an add instruction should be emitted, using the specified register numbers as the destination and the lhs operands. The aluRhs “token” will either be register number or a small immediate value which is encodeable by the add instruction. For our example above the token would be the constant 1000, and the following code would be emitted:

```
add r0, r1, #1000
```

1.5 Compiler Back End – Register Manager

The Register Manager is a key component of the compiler back end that is used to track the location of evaluated expressions. The location can be either in the compiled frame, in the constant pool, and/or in a register allocated for the evaluated expression. Each evaluated expression is mapped to a resource tracked by the Register Manager.

The Register Manager supports targeting of resources into registers allocated from a register set specified by the caller. Although register allocation is a key feature of the Register Manager, the tracking of evaluated expressions is the primary function, and therefore it could be more appropriately named the Resource Manager or Value Manager.

The Register Manager is described in more detail in the Register Manager chapter.

1.6 Compiler Back End – Code Emitters

Another key component of the compiler back end are the code emitters. The `CVMCPUemitBinaryALU()` emitter is used in the example above to emit the add instruction. Code emitters exist at the lowest layer of the dynamic compiler and usually are responsible for generating a very specific instruction or set of instructions, such as adding together the value of two specific registers and storing the result in a specific register. While much of the JIT source code is shareable across most processors, the code emitters are always processor specific. In fact, it is the presence of the code emitter APIs in the JIT HPI that make it possible to share much of the JIT source code. Code emitters are covered in section 5.3.6 of the *CDC Porting Guide*.

1.7 Other Compiler Components

There are many other components in the JIT. They are described in the following chapters:

- Code Cache Manager: Describes the management of memory that compiled methods reside in.
- JIT Memory Manager: Describes the management of memory allocated for use during compilation.

- Constant Pool Manager: Describes the management of constants referenced from compiled code.
- Stack Manager: Describes the management of method parameters on the Java stack.
- JIT Stack Maps: Describes the creation of GC stack maps for compiled methods.
- JIT Runtime Support: Describes the runtime support code used by compiled methods for doing things like invocations, object allocation, and object synchronization.
- In addition, more details on various compiler topics can be found in the following chapters:
- Phi Values: Explains phis, which are values left on the Java stack across basic blocks.
- Trap-based `NullPointerExceptions`: Explains the handling of null object references by allowing them to cause a trap (SEGV).
- GC Checks in Compiled Code: Explains how explicit GC check points at backwards branch targets work in compiled methods.
- Assembler Listings for Dynamically Compiled Code: Explains how to generate assembler listings for dynamically generated code.
- Code Generation Example: Steps through a few code generation examples from Java source to bytecodes to IR to native code.
- Intrinsic Methods: Explains how intrinsic methods work.
- JIT Debugging Support: Provides debugging tips.
- Profiling: Explains how JIT profiling works.

1.8 Compiler Porting Layers

One of the primary goals of the JIT is to make as much source code shareable as possible among as many processor ports as possible. As mentioned above, having a well defined set of emitter APIs aids in this goal. However, it is not possible to define a set of emitter APIs that is flexible enough to support all processors, while still allowing most JIT source to be shared. For this reason, it was decided that the emitter APIs would only support common RISC processors such as ARM, PowerPC, MIPS, Sparc, and SH. These processors all have many common characteristics, such as a large set of general purpose registers, uniform instruction sizes, and simple addressing modes. The differences between the processors, such as the number of registers and instruction encodings, are easily abstracted with or hidden behind a set of porting interfaces. Since some porting interfaces can be made common to all processor ports, the JIT porting layer is split into two parts:

- Shared layer: The shared porting layer is expected to be implemented by all processor ports. This porting layer is relatively small and is defined in `src/share/javavm/include/porting/jit/jit.h and ccm.h`.

- RISC layer: The RISC porting layer is where most of the porting effort is made, but is only implemented by RISC ports. It mostly consists of two parts: the low level emitter APIs and the CPU abstraction macros. The emitter APIs are responsible for emitting instructions and the CPU abstraction macros define characteristics of the processor.

Because of these two abstraction layers, all source for a JIT port that implements the RISC porting layer will fall into one of three categories:

- Processor specific code: This is where the porting effort for RISC ports is made, and includes the implementation of the RISC porting layer and part of the Shared porting layer. Code emitters and macros that characterize the processor make up most of the processor specific code.
- Shared RISC code: This code is shareable by all RISC ports and is expected to run on all RISC processors. Most of the Shared porting layer is implemented by Shared RISC code. Although there is very little to implement in terms of the number of APIs, one of the APIs is `CVMJITcompileGenerateCode()`, which is responsible for translating the IR into native instructions, so there is a large amount of Shared RISC code to handle this task. The Shared RISC code is implemented within the `src/portlibs/jit/risc` directory. The primary Shared RISC components are the JCS grammar and the Register Manager, and they are also the primary clients of the RISC porting layer. Shared RISC code is the client of the RISC porting layer.
- Shared code: This code is shareable by all ports and is expected to run on all processors. It includes the entire front end (IR generator) and some minor backend components such as the Code Cache Manager and JIT Memory Manager. Shared code is the client of the Shared porting layer.

More information on the porting layers can be found in Dynamic Compiler chapter of the CDC Porting Guide. It documents all the source files involved in doing a port. Details of the porting layers are documented in the header files.

1.9 CVMJITCompilationContext

Whenever a method is being compiled, a `CVMJITCompilationContext struct` is created. It maintains information about the state or context of the compilation, and is the main conduit of information between the various passes and components of the compilation.

A `CVMJITCompilationContext*` is passed to almost every API in the compiler. Usually it is referred to as `con`, both as the argument name and when referenced in this document.

1.10 Compiling a Method

`CVMJITcompileMethod()` is the entry point for the compilation of a method. It is called automatically by the interpreter loop when it detects that a method should be compiled. The actions taken by `CVMJITcompileMethod()` are summarized below:

```
CVMJITcompileMethod() {
CVMsysMutexLock(ee, &CVMglobals.jitLock);
CVMJITinitializeCompilation(&con);
CVMJITcompileBytecodeToIR(&con);
CVMJITcompileOptimizeIR(&con);
CVMJITcbufAllocate(&con, extraCodeExpansion);
CVMJITcompileGenerateCode(&con);
CVMJITwriteStackmaps(&con);
<intialize cmd>
CVMsysMutexUnlock(ee, &CVMglobals.jitLock);
}
```

- `CVMsysMutexLock()`: Only one compilation is allowed at a time, and this is accomplished by holding the `jitLock` during compilation.
- `CVMJITinitializeCompilation()`: initializes the `CVMJITCompilationContext` for use during the compilation.
- `CVMJITcompileBytecodeToIR()`: The front end pass. Converts the Java bytecodes to the intermediate representation (IR).
- `CVMJITcompileOptimizeIR()`: An optimization pass over the IR. Currently all IR optimizations are done when creating the IR, so `CVMJITcompileOptimizeIR()` does nothing at this point.
- `CVMJITcbufAllocate()`: Allocates a code buffer for the compiled method.
- `CVMJITCompileGenerateCode()`: The back end pass. Converts the IR to native instructions.
- `CVMJITwriteStackmaps()`: Writes GC stack maps for the compiled method.
- `<initialize cmd>`: A series of instructions that initializes the methods `CVMCompiledMethodDescriptor(cmd)`. This is where compilation specific information for a method is stored, such as the pointer to the start of the method and the pointer to stack maps for the method.
- `CVMsysMutexUnlock()`: Unlocks the `jitLock` so another thread can start compilation.

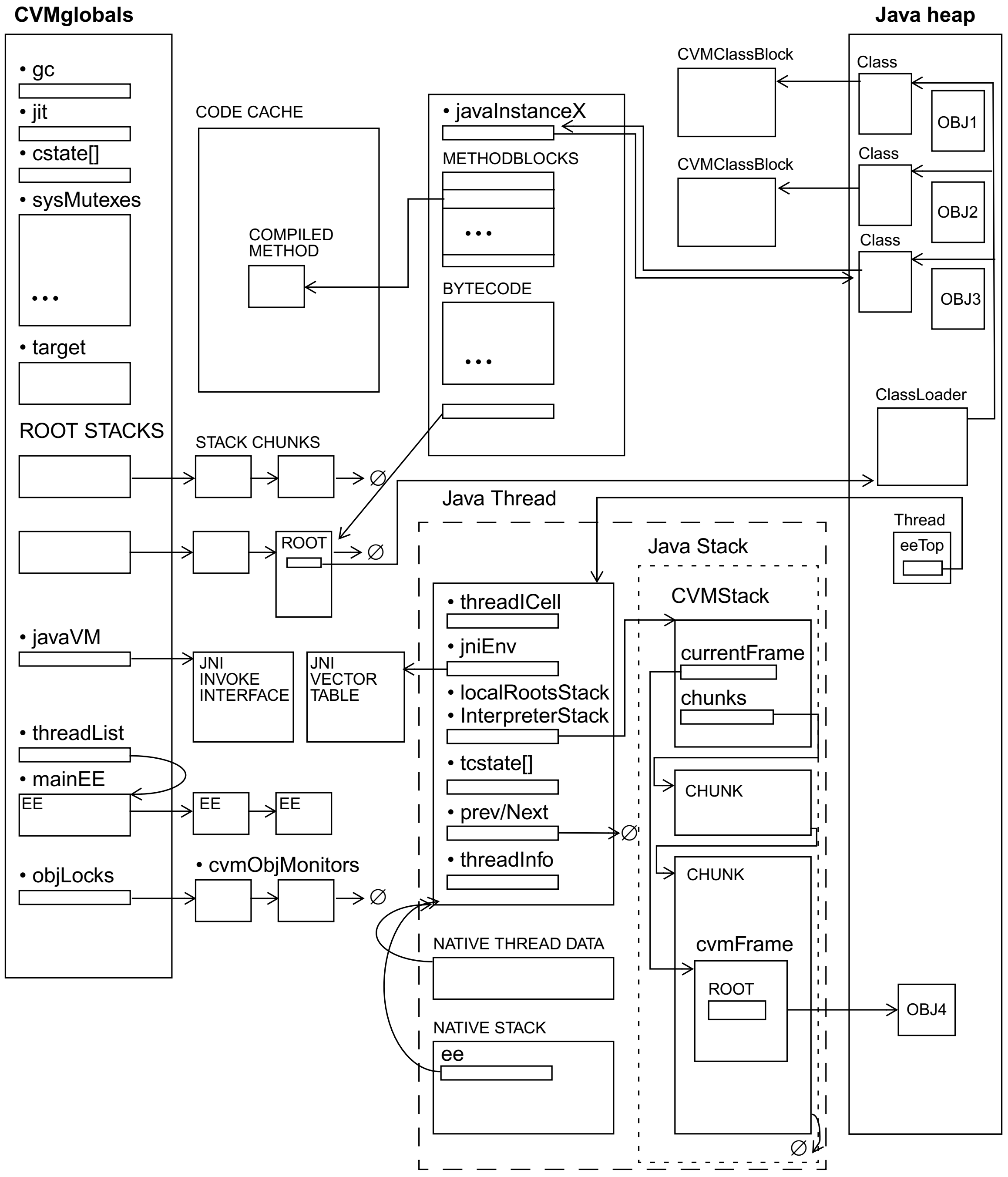
CHAPTER 2

VM Overview and Runtime Internals

2.1 CVM Internal Structure with Compiler

CVM is a fully compliant Java 2 VM. The following sections will go through a map of all the major data structures in the VM, and discuss the various sub-systems that are associated with those data structures.

FIGURE 2-1 Map of CVM Data Structures



2.2 Globals

`CVMglobals` (of type `CVMGlobalState`) is a consolidated struct for holding all global variables in the VM, and as the root of the tree of all runtime data structures used in the VM. A consolidated location for all global variables makes it easier to re-initialize the VM when needed in the absence of a process model.

2.3 The Java Heap

`CVMglobals.gc` (of type `CVMGCGlobalState`) is the global data structure for the GC implementation that is built into CVM, and its contents are defined by the GC implementation.

The Java heap is allocated and is managed by the GC implementation. It is allocated during GC initialization, and a pointer to it is usually stored somewhere inside `CVMglobals.gc`. The Java heap is used exclusively for storing Java objects only. Other VM and class meta-data constructs are stored in malloced memory.

2.3.1 The Garbage Collector

CVM implements a built time pluggable garbage collector interface that allows it to be used with different GC implementations if desirabed. The garbage collector choice is configured at build time and is not changeable at runtime. CVM comes with a generational collector which is built into CVM by default.

Regardless of the specific GC implementation, CVM assumes that the GC will be exact. This means that CVM will provide some point in the execution of code when the VM will be in a known state and all object references can be found. (See Chapter 6, "Creating a Garbage Collector" in the *CDC Porting Guide*.)

2.3.2 GC Consistency

Java threads will constantly alternate back and forth between a GC consistent (GC safe) and GC inconsistent (GC unsafe) state. When the thread is in a GC safe state, it will not hold onto any direct object pointers. Instead, all object references are kept in ICells (see below) which are indirect pointers to the Java objects. This allows the GC to run without fear of moving objects and invalidating pointers that are being used by the thread. When the thread is in a GC unsafe state, the thread may operate on direct object pointers without fear of GC invalidating the values of those pointers.

The thread voluntarily switches back and forth between GC safe and unsafe states. It never stays in a GC unsafe state for too long so as to not starve threads wanting to do GC. The thread running GC will request that all threads become safe before proceeding with GC. All threads are supposed to cooperate and become GC safe at their earliest convenience. When all threads are GC safe, the GC thread is notified and GC can commence.

Threads executing Java byte codes in interpreted mode will operate while mostly GC unsafe except at designated points such as:

- method invocations and returns
- backward branches
- calls into complex VM runtime functions

Threads executing JIT compiled code will operate while mostly GC unsafe until they need to call into certain JIT runtime functions. Backwards branches and method invocations can be made to become GC safe by use of a patching mechanism if available. Else, these locations will implement explicit GC checks which become GC safe if necessary.

Threads executing JNI native methods will operate while mostly GC safe. When the thread needs to operate on Java objects, it calls through JNI APIs which will become GC unsafe long enough to operate on the object, and then resumes being GC safe before returning to the native method.

Most complex VM runtime work needs to be done in a GC safe state for 2 reasons:

- The work being done may be lengthy, and this should not stop another thread from running GC while waiting for this work to complete.
- The VM runtime may need to acquire locks for its work. Hence, the current thread may block on those locks. The thread needs to make sure it is GC safe when blocked so as to not stop another thread from running GC.

GC safety within the VM is covered in more detail in the first few pages of the Creating a Garbage Collector section and the chapter “How To Be GC-Safe” in the *CDC Porting Guide*.

2.3.3 `CVMObjectICell`

A `CVMObjectICell` (or icells in general) is a construct for holding a pointer to an object. The location of every icell in the system is known to the VM. Hence, when GC needs to run, each icell will be updated with new object pointer values if the object that they refer to gets moved during the GC cycle.

GC safe code are not supposed to refer to Java objects using direct pointers. Instead, they should be using icell pointers (i.e. `CVMObjectICell*` or `jobject`).

2.4 The JIT Code Cache

`CVMglobals.jit`(of type `CVMJITGlobalState`) is the global data structure for storing control data structures and parameters for the dynamic compiler. A pointer to the JIT code cache is also stored there. The JIT code cache is a contiguous memory buffer which is used for storing compiled methods and their meta-data generated by the dynamic compiler. The amount of memory available for storing compiled methods and their meta-data is limited by the size of the code cache which can be specified on the VM command line. The code cache is allocated using malloc during VM initialization.

2.5 Java Classes

Java classes are represented in memory as two constructs that work together in pairs:

- `java.lang.Class object`
- `CVMClassBlock` struct

2.5.1 `java.lang.Class`

The `Class` object, like any Java objects, are allocated out of the Java heap, except for preloaded classes (see ROMized / Preloaded Classes below). Each Class object has a pointer to its `CVMClassBlock` counterpart. The Class object only serves the purpose of being the representation of the class that is observable from Java code. All the interesting information about the class itself is actually stored in the corresponding `CVMClassBlock`.

2.5.2 CVMClassBlock

The `CVMClassBlock` (or class block in general) is a C struct that is used to hold all the meta-data for a Java class. It is allocated using `malloc`, except for preloaded classes (see ROMized / Preloaded Classes below). There is one class block per Java class. The class block contains an icell pointer to an icell that refers to its corresponding `java.lang.Class` instance.

The class meta-data that is stored in or tracked by the class block includes:

- GC reference map for instances
- pointer to the virtual function table
- pointer to static fields
- method blocks
- other properties of the class (superclass, inner classes)

The GC reference map is used during garbage collection to identify the reference fields in the object instances of the class. This allows the GC to scan and update those reference fields if needed.

Static fields which are references are grouped together contiguously at the start of the block of static fields of the class. The class block also keeps track of how many reference static fields it has. This allows GC to scan and update these reference fields without the use of a map.

2.5.3 CVMMethodBlock

The `CVMMethodBlock` (or method block in general) is a C struct for holding meta-data about a method. It is allocated on the C heap as part of the class block allocation, i.e. the allocation of the class block asks for more memory to allow for the method blocks to be located immediately after the class block data itself. There is one method block struct for each method in the Java class (abstract, Java bytecode, or native).

The method block contains the following method data:

- name and signature
- exceptions
- pointer to more specific information for Java bytecode or native method

The method block is not immutable. When a method gets compiled, a pointer to the start of the compiled method is stored into the method block.

Java bytecode methods are commonly referred to simply as Java methods. For Java methods, the method block will have a pointer to a `CVMJavaMethodDescriptor` struct. The `CVMJavaMethodDescriptor` contains:

- the bytecodes for the method

- the method exception table
- debugging information like the line number table if applicable

The `CVMJavaMethodDescriptor` is immutable once it is initialized.

For native methods, the method block will have a pointer to the first instruction of the native method.

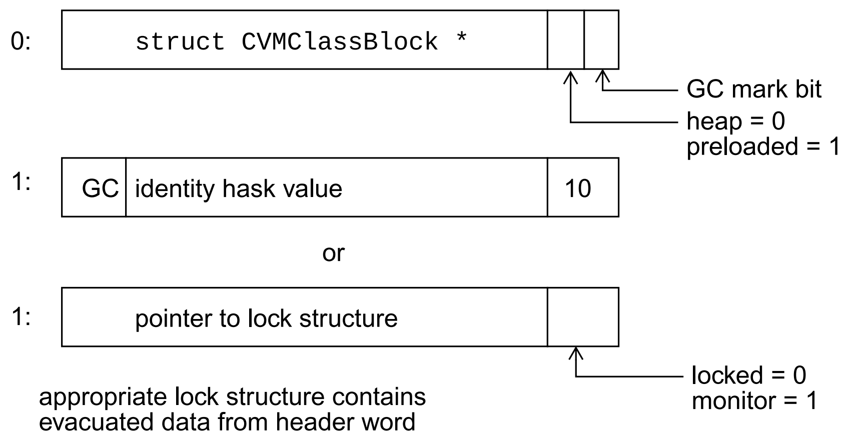
2.5.4 Preloaded Classes

CVM includes support for preloaded classes. Preloading of classes means to pre-digest the data for Java classes into `java.lang.Class` instances and class blocks (and all its sub-tree of information including the method blocks) at build time. Hence, classloading time will not be incurred at runtime for these preloaded classes. The preloaded `java.lang.Class` instances are allocated from the `.bss` segment (instead of from the Java heap), and the class blocks are allocated in read-only data (instead of using `malloc`).

2.6 Java Objects

Java objects are allocated out of the Java heap, except for preloaded objects. Preloaded objects are allocated from the `.bss` segment. The only objects which can be preloaded currently are `java.lang.String` objects.

All Java objects start with the following header structure:



2.7 Java Threads

Each Java thread executes in its own native thread, and is associated with the following data structures:

- a `CVMExecEnv` struct
- a `JNIEnv` struct
- some native thread data
- a native stack
- a Java stack

2.7.1 `CVMExecEnv`

The `CVMExecEnv` is the root data structure of the Java thread. A pointer to the current thread's `CVMExecEnv` (commonly known as the ee) is usually passed around to all VM functions. The `CVMExecEnv` serves as a node in the global link list of all threads in the VM. The head of the list is stored at CVMglobals.threadList.

The `CVMExecEnv` also serves to store the state of the thread. This includes:

- Java locks owned, system mutexes owned
- the Java stack
- exceptions

All `CVMExecEnvs` are allocated using `malloc` except for the one for the main thread. The main thread's `CVMExecEnv` is embedded in `CVMglobals` as `CVMglobals.mainEE`.

2.7.2 `JNIEnv`

The `JNIEnv` is the thread specific context for JNI native methods. The `JNIEnv` struct is actually embedded within the `CVMExecEnv` of the thread. The `JNIEnv*` pointer value can be computed from the `CVMExecEnv*` pointer value, and vice versa as illustrated by the following macros:

```
#define CVMjniEnv2ExecEnv(je)   \
((CVMExecEnv *)((char *)(je) - \
CVMoffsetof(CVMExecEnv, jniEnv)))


#define CVMexecEnv2JniEnv(ee)  (&(ee)->jniEnv)
```

2.7.3 Native Stack Data

The native stack data is a `CVMThreadID` struct which is embedded in the `CVMExecEnv`. `CVMThreadID` is target platform specific and is defined in the VM porting layer implementation.

2.7.4 Native Stack

Since each Java thread executes in its own native thread, it also has its own native stack. The shape and mechanics of the native stack depends on the underlying target platform.

2.7.5 Java Stack

Each Java thread has a Java stack (`CVMStack`) which is allocated as a link list of chunks. The Java stack is also commonly referred to as the interpreter stack. The stack is allocated using `malloc`. More chunks are added automatically to the stack as needed up to a limit. The stack is bounds checked as a new frame is pushed during method invocation. Chunk crossing can cause arguments or return value copying during method invocation or return.

2.7.6 Java Stack Frames

For each method invocation, a frame is pushed on the stack. The frames types include:

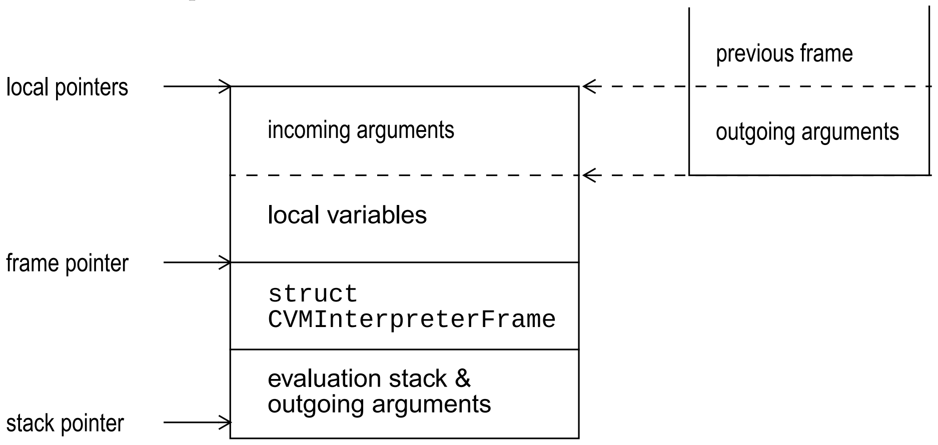
- Java frames (`CVMInterpreterFrame`)
- Compiled frames (`CVMCompiledFrame`)
- JNI native method frames (`CVMFreelistFrame`)

Stack frames generally consists of incoming arguments which are part of the local variables of the method (also commonly referred to as locals), a frame record, and an operand stack area. The outgoing arguments of a method usually reside in the operand stack region of the caller method frame, and this stack region is mapped as the start of the locals region in the callee method frame. No arguments copying is done except for a few exceptional cases.

2.7.6.1 `CVMInterpreterFrame`

The `CVMInterpreterFrame` is shaped as shown in FIGURE 2-2:

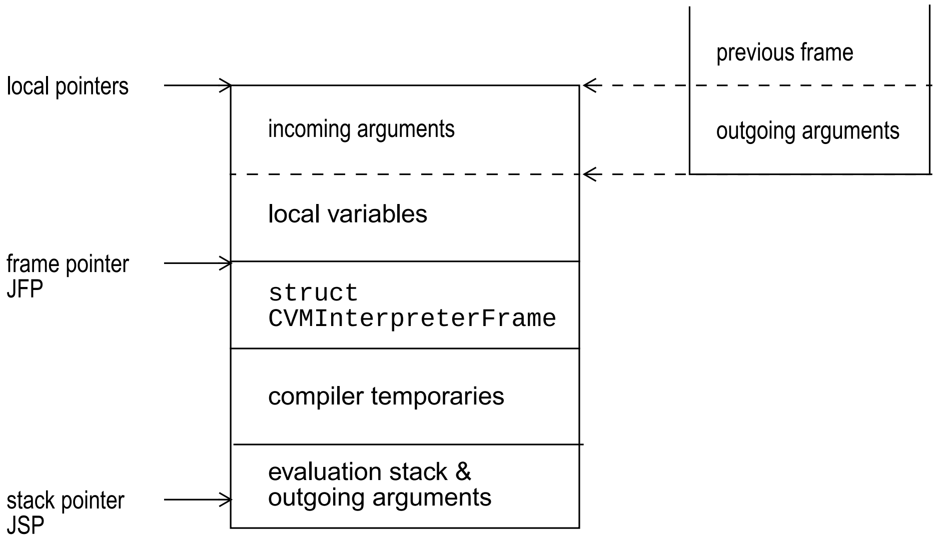
FIGURE 2-2 CVMInterpreterFrame



2.7.6.2 `CVMCompiledFrame`

The `CVMCompiledFrame` is shaped as shown in FIGURE 2-3:

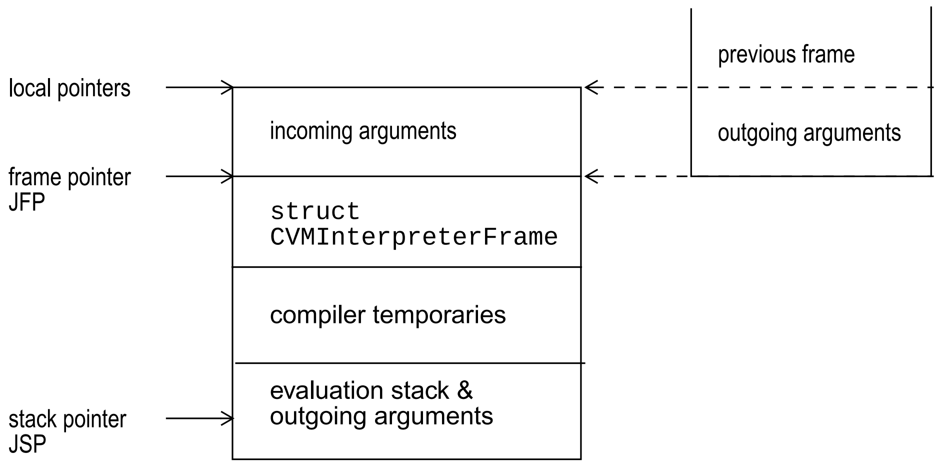
FIGURE 2-3 `CVMCompiledFrame`



2.7.6.3 CVMFreelistFrame

The `CVMFreelistFrame` is shaped as shown in FIGURE 2-4:

FIGURE 2-4 `CVMFreelistFrame`



2.7.7 Stackmaps

In order for the GC to be able to find object references on the Java stack, it makes use of stackmap data structures (`CVMStackMaps`) for the method being executed in the frame on the stack. For a given program counter values, the stackmap will indicate where in the object references are located within the stack frame. There only need to be stackmaps for program counter values which represent GC points in the code.

Stackmaps for interpreted frames are generated at GC time as needed using a stackmap computer. Stackmaps for compiled frames are generated at the JIT compilation time of the method being executed.

JNI native frames do not require stackmaps. Their incoming arguments are scanned as part of the caller frame. Their local refs are contiguous and the bounds of this region is known. Their outgoing arguments are scanned as part of the callee frame.

The native stack is not scanned by the GC. This is because there are no direct object references on the native stack. All references to objects on the native stack are made through icells.

2.8 GC Roots

During garbage collection, the GC needs to scan all object references that are live. These object references can reside in the following locations:

- object instance fields
- static fields
- Java stack references and local refs
- global refs

Object instance fields are found using the GC reference maps stored in the class block of the object, and static fields are always located contiguously at the beginning of the class static fields (see `CVMClassBlock` above). Java stack references are found using stackmaps. Local refs are found using specific knowledge about the shape of JNI native frames.

Global refs are allocated from global root stacks which are similar to the interpreter stack. Unlike interpreter stacks, the global root stacks are not associated with a specific thread but are part of the VM globals (e.g. `CVMglobals.globalRoots`). The global root stacks only have one `CVMFreelistFrame` that can hold as many global refs (also known as roots) as is limited by space in the stack. Like the JNI native frames, no stackmap is needed here because the global refs are all contiguous and the bounds of the region containing these references are known from the `CVMFreelistFrame`.

2.9 Synchronization

Synchronization on Java objects is done using a fast lock mechanism using light-weight lock records (referred to as fast locks) in most cases and only done using a real lock mechanism (referred to as inflated locks) when needed. The premise behind this implementation is that contention on Java locks are rare. Hence, there is no need to associate an inflated lock with the object until contention occurs. Using an inflated lock tends to be slower than just using a fast lock.

A Java object can be in one of 3 lock states at any one time:

```
enum {
  CVM_LOCKSTATE_LOCKED = 0,
  CVM_LOCKSTATE_MONITOR = 1,
  CVM_LOCKSTATE_UNLOCKED = 2
};
```

The lock state of the object is recorded in the least significant 2 bits of the 2nd word in the object header (see Java Objects notes above). These bits are commonly referred to as the object sync bits. For simplicity, this 2nd word in the object will be referred to as the header word in the following.

By default, an object is unlocked, and the sync bits are set to `CVM_LOCKSTATE_UNLOCKED`.

When a thread attempts to lock the object, it uses atomic operations to check and set the sync bits as well as the header word. If the sync bits are `CVM_LOCKSTATE_UNLOCKED`, then there is no contention on this object yet. Within the same atomic operation, the header word will be replaced with a pointer to a fast lock record and the sync bits are set to `CVM_LOCKSTATE_LOCKED`. The original header word value is saved in the fast lock record (`CVMOwnedMonitor`).

If the same thread attempts to re-enter the lock on this object, it will find that the sync bits are already set to `CVM_LOCKSTATE_LOCKED`, and check to see if it is the owner of the fast lock. Since the current thread does own this fast lock, it simply increments the reentry count in the fast lock record and proceed with code execution.

If a different thread attempts to acquire the lock on this object, it will check and see that it is not the owner of the fast lock record. This is considered a contention case which will trigger the inflation of the lock. The lock inflation process is synchronized with a global system mutex (`CVMglobals.syncLock`). Hence, only one thread can do lock inflation at any one time.

To inflate the lock, the thread first replaces the header word and sync bits with values that make the object appear as if it is locked with a fast lock but no thread owns it. Once the header word and sync bits have been set appropriately, the thread can proceed with associating the object with an inflated lock (`CVMObjMonitor`). The original header word and sync bits are saved in the inflated lock. The object header word will now be a pointer to the inflated lock, and its sync bits will be set to `CVM_LOCKSTATE_MONITOR`. Ownership of the inflated lock will be assigned to the thread that is the owner of the fast lock record previously associated with this object. After inflation is complete, the thread will try to acquire the inflated lock. Since the inflated lock is owned by another thread, this thread will block.

Other threads (including the actual lock owner thread) that try to lock this object while inflation is in progress will detect a contention case and attempt to inflate the lock. In so doing they will block on `CVMglobals.syncLock`. Once the inflater thread is done inflating the lock and releases `CVMglobals.syncLock`, these other threads will detect that the lock is already inflated and contend directly on the inflated lock. The owner thread will be able to re-enter the inflated lock and continues execution since it already owns the lock. Other threads will block on the lock until the owner releases it completely (i.e. re-entry count goes back down to 0).

The atomic operations used depend on the fastlock type (`CVM_FASTLOCK_TYPE`). The fastlock type choice is a build time option and depends on the target platform port. The choices are:

- `CVM_FASTLOCK_NONE`
- `CVM_FASTLOCK_ATOMICOPS`
- `CVM_FASTLOCK_MICROLOCK`

`CVM_FASTLOCK_NONEis` basically a scheme where every lock on an unlocked object is considered a contention case, and triggers inflation. Hence, an inflated lock will always be used to synchronize on the object.

`CVM_FASTLOCK_ATOMICOPS` is a scheme where the object header word and sync bits are checked and modified using an atomic compare and swap instruction.

`CVM_FASTLOCK_MICROLOCKis` a scheme where the object header word and sync bits are checked and modified only within critical regions that are protected by mutexes called microlocks. By default, microlocks are implemented as mutexes supplied by the underlying target platform port. The platform may choose to implement the microlock using custom mechanisms. An example of this is by using a scheduler lock that disables context switching of threads.

See Chapter 4, "Fast Locking" in the *CDC Porting Guide*.

2.10 Other VM Components

Other components of the VM include:

- the Java bytecode interpreter.
- the class loading, linking, preparing, and verification sub-systems.
- the dynamic compiler.
- glue logic to enable transitions between interpreted and compiled code.
- `Misc.runtime` libraries for interpreted and compiled code.

2.11 The Bootstrapping Process

The following listing shows the flow of execution from the main C function until the main Java method is executed. Each line in the listing signifies a forward step in time. Some details are left out for succinctness. In essence, the bootstrap process starts with initializing the VM. After the VM is initialized, the JNI APIs are used to invoke the main method of the Java application.

We see that the call to JNI's `CallStaticVoidMethod()` for invoking the main Java method eventually leads to `CVMgcUnsafeExecuteJavaMethod`. This is true regardless of whether the main Java method is interpreted, compiled, or native. `CVMgcUnsafeExecuteJavaMethod` is the VM's interpreter loop function.

New threads are bootstrapped in a similar fashion by invoking the thread's start method using the JNI `CallVoidMethod` API which ultimately calls `CVMgcUnsafeExecuteJavaMethod`.

```
main(argc, argv) {
  ansiJavaMain(argc, argv) {
    // Initialize the VM:
    JNI_CreateJavaVM() {
      CVMinitVMGlobalState(gs) {
        CVMpreloaderInit();
        CVMinitExecEnv(ee, CVM_TRUE) {
          // Init the thread's Java stack:
          CVMinitStack(... ee->interpreterStack);
          CVMjniFrameInit(...);
          CVMinitStack(... ee->localRootsStack) ;
          CVMinitJNIEnv(ee->jniEnv);
        }
        CVMinitGCRootStack(... gs->globalRoots);
        ...
        CVMinitGCRootStack(... gs->classLoaderGlobalRoots);
        CVMinitStack(... gs->classTable);
        CVMfreelistFrameInit(gs->classTable.currentFrame);
        ...
        CVMinitVMTargetGlobalState(gs->target);
        CVMgcImplInitGlobalState(gs->gc);
        CVMinitJNIJavaVM(gs->javaVM);
        ...
        CVMtypeidInit(ee);
        ...
        CVMID_allocNewInstance (...,
          CVMsystemClass (java_lang_OutOfMemoryError));
        CVMID_allocNewInstance (...,
          CVMSystemClass (java_lang_StackOverflowError));
      }
```

```
        }

        ...
        // Invoke the main Java method:
        (*env)->CallStaticVoidMethod(env, cvmClass, runMainID) {
          ...
          // Use assembly glue to call native method:
          CVMjniInvoke(...) {
            // Marshall arguments onto Java stack:
            ...
            // Invoke the interpreter:
            CVMgcUnsafeExecuteJavaMethod(...);
            // Marshall return value:
            ...
          }
        }
      }
    }
```

CHAPTER 3

JIT Intermediate Representation

3.1 Overview

Java source files are compiled into Java byte-codes, as defined in the JVM specification. The JVM execution model is stack based. There is a runtime stack which is used by byte-code instructions. Values are popped from the stack, computation results are pushed back on the stack, method arguments and return values are passed on the stack. So all "communication" between instructions and all computation happens via the stack.

Our aim in the JIT is to translate these stack-oriented byte-codes into native code. In the process, we'd like to analyze and manipulate code, optimize it, and remove redundant computation. We'd like this manipulation to be fairly platform independent, so we can share most of it between our ports. In addition, we aim to expose some of the implicit Java language semantics inherent in the byte-codes by expressing them via explicit lower-level operations (among these implicit semantics are runtime checks and strict left-to-right evaluation order of expressions). Finally, on architectures where it makes sense, we'd like to eliminate stack overhead as much as possible, and use registers for most computations.

So we choose to translate byte-codes into a platform-independent intermediate representation (IR) that is more amenable to manipulation than stack-oriented byte-codes. Most code manipulation happens on the IR. The resultant IR is passed on to the code generator for conversion into native code.

The purpose of this section is to outline what the IR looks like, and how the byte-code to IR conversion is done. We outline the data structures, the conversion passes, and the APIs involved in this intermediate conversion. The resultant set of data structures make up the IR that represents the original byte-codes.

3.2 From Stack-Oriented to Value-Oriented

Our IR is a list of expression trees, which when traversed in postfix order represent the original byte-code computation. All implicit Java semantics, such as strict left-to-right evaluation order and implicit bounds checks and null checks, are made explicit in the IR. This structure is most convenient to represent our stack-oriented byte-codes for conversion into register-oriented low-level machine code.

Let's start with an example:

```
class c {
 int _foo;
 public int f(int k) {
  return _foo + k;
 }
}
```

The stack-based byte-codes for method `c.f()` are:

```
  aload_0   // this
  getfield 2 // _foo
  iload_1   // k
  iadd    // _foo + k
  ireturn
```

Our IR would represent this statement as an expression tree:

```
             IRETURN (7)
               |
              IADD (6)
             /   \
         (4) FETCH32  LOCAL 1 (5)
             |
         (3) FIELDREF32
            /   \
       (1) LOCAL 0 CONST 2 (2)
```

The original stack semantics are replaced with a value-based system. Each node has zero, one or two arguments. These are linked together in a tree to reflect the original computation.

The code generator receives this tree, and traverses it in a postfix manner. The traversal order is marked on the above graph. Each subnode traversal could involve code generation as a side effect. The result of each computation can be associated with the subnode and passed on to other nodes for use for their code generation. When the traversal is complete, the program's translation is complete. Note that the traversal order is key to satisfying Java evaluation order and runtime check requirements, so the "shape" of the IR tree produced by the front end cannot be altered very easily.

3.3 IR Basics

First off, let's define a few terms and concepts.

Our JIT compiles Java methods. Each method is split into "extended basic blocks" -- sequences of byte-codes in which flow control always enters at the beginning (i.e. there are no branches into the middle of an extended basic block). Once extended basic blocks are determined, the front-end works block by block, and generates the IR for each block.

Expressions are built into expression trees made up of "IR nodes." IR expression trees are collected under "root IR nodes." Root nodes represent statements, such as an assignment, branch or void method invocation.

At the end of the bytecode-to-IR conversion, a list of root IR nodes hangs off of each block, representing the code for that block. And blocks are linked together as well. This means that the code generator can go over each block, iterate over the root nodes of each block, and generate code for the trees found within.

Now we will explain the format of an IR node, a root node, and a block.

3.4 IR Node Format

Each IR node is represented by a C struct. The main node type is `CVMJITIRNode`. In object-oriented terms, `CVMJITIRNode` is the "super-class" of all IR nodes. "Subclasses" of nodes include binary nodes, unary nodes, lookupswitch and tableswitch nodes, among others.

For reference, here's the full declaration of a `CVMJITIRNode`:

```
struct CVMJITIRNode {
#if defined(CVM_DEBUG) || defined(CVM_TRACE_JIT)
```

```
  CVMUint32 nodeID; /* for IRdump purpose */
  CVMBool  dumpTag; /* Each IR node is dumped only once. */
#endif
  CVMUint16 tag;            /* encodes node type, opcode, and typeid
*/
  CVMUint16 refCount;
  CVMUint16 curRootCnt; /* number of nodes built in current node */
                         /* also used by code gen for state tag */
  CVMUint16 flags;   /* See CVMJITIRNodeFlags enum list below. */
  CVMInt32  regsRequired;/* code generation synthesized attributes */
  /*
   * Nodes can be decorated with some extra information so the
   * backend can do a better job of register allocation. For example,
   * we like to pass phi values in registers, but often the phi value
   * is evaluated into the wrong register before the backend sees its
   * DEFINE node. The solution is to have the creation of the DEFINE
   * node store the stackIdx in the value node so the backend knows
   * which register to put it in.
   */
  CVMJITIRNodeDecorationType decorationType;
  union {
                 CVMUint16 phiStackInfo;/* Stack index for outgoing
phi values */
                 CVMUint16 argNo; /* argument # for outgoing
parameter values */
  } decorationData ;
  CVMJITIRSubclassNode type_node;
};
```

Each node has several fields we care about. The most important shared fields are:

- An id: a unique block-local identifier for the node. (’nodeID’)
- A tag: A 16-bit value that encodes the opcode, basic type and subclass type of the node. (’tag’)
- A set of flags for the node. (’flags’)
- A field of type `CVMJITIRSubclassNode` that is a union of all subclass specializations of an IR node (’type_node’).

Node identifiers are assigned sequentially as the IR nodes for a block are created. A node created during the translation of a block can only be referred to by other nodes created within the block. So the id is unique for the nodes of a block. Note that id's are only created in debug builds, for memory efficiency reasons.

Tags are an encoding of the opcode, node type, and basic type of the node. The <opcode,node,type> triple uniquely defines the kind of a node.

Opcode tags are picked from the enumerated type `CVMJITIROpcodeTag`, and indicate what operation this opcode represents. Examples are `CVMJIT_GET_VTBL`, `CVMJIT_LOCAL`, `CVMJIT_RETURN`, and `CVMJIT_NEW_OBJECT`.

For reference, here's the definition of a `CVMJITIROpcode` tag with a few highlighted items (for full defintion see `jitirnode`.h).

```
typedef enum CVMJITIROpcodeTag {
  [...]
  /*****************************************************************
   * Constants - opcodes starting with CVMJIT_CONST are all opcodes
   * for CVMJITConstant32 or CVMJITConstant64 nodes.
   *****************************************************************/
  /* The follow are all genreted by ldc opcodes */
  CVMJIT_CONST_JAVA_NUMERIC32, /* java Integer and Float */
  /* CVMJITLocal, reference to local variables */
  CVMJIT_LOCAL,             /* {i|f|a|l|d|l}load */
  /* CVMJITUnaryOp */
  CVMJIT_ARRAY_LENGTH,
  CVMJIT_NEW_OBJECT,
  /* NOTE: It is important that the following NEW_ARRAY
    opcodes are sorted in the same order as the sequence
    of corresponding types in the CVMBasicType enum list
    (see basictypes.h). */
  CVMJIT_NEW_ARRAY_BOOLEAN,
  /* CVMJITUnaryOp, arithmethic and conversion ops */
  CVMJIT_CONVERT_I2B,    /* convert i2b. */
  /* CVMJITUnaryOp */
  CVMJIT_RET,                  /* opc_ret */
  CVMJIT_RETURN_VALUE,   /* {a|i|f|l|d}return */
  /* CVMJITBinaryOp */
  CVMJIT_ASSIGN,            /* {i|f|a|l|d|c|s|b}store */
  CVMJIT_INVOKE,
  /* CVMJITBinaryOp, arithmetic ops */
  CVMJIT_ADD,            /* {i|f|l|d}ADD */
  /* CVMJITBranchOp */
  CVMJIT_GOTO,
  /* CVMJITConditionalBranch */
  CVMJIT_BCOND,
  [...]
} CVMJITIROpcodeTag;
```

Node tags (subclass indicators) are picked from the enumerated type `CVMJITIRNodeTag`, and indicate what "concrete subclass" this node is. Examples are `CVMJIT_ROOT_NODE`, `CVMJIT_UNARY_NODE` and `CVMJIT_BINARY_NODE`.

Here's the full list of node tags:

```
typedef enum CVMJITIRNodeTag {
  CVMJIT_ROOT_NODE,
  CVMJIT_CONSTANT_NODE,
  CVMJIT_NULL_NODE,
  CVMJIT_LOCAL_NODE,
  CVMJIT_UNARY_NODE,
  CVMJIT_BINARY_NODE,
  CVMJIT_BRANCH_NODE,
  CVMJIT_CONDBRANCH_NODE,
  CVMJIT_LOOKUPSWITCH_NODE,
  CVMJIT_TABLESWITCH_NODE,
  CVMJIT_PHI_NODE,     /* DEFINE and USED */
  CVMJIT_PHI_LIST_NODE,   /* LOAD_PHIS, RELEASE_PHIS */
  CVMJIT_MAP_PC_NODE,
  /* NOTE: CVMJIT_TOTAL_IR_NODE_TAGS must always be at
    end of this enum list. It is used to ensure that the
    number of types of IROpcodeTags don't exceed the
    storage capacity alloted for it in the encoding of
    node tags.
  */
  CVMJIT_TOTAL_IR_NODE_TAGS
} CVMJITIRNodeTag;
```

And finally, type tags are picked from a CVM_TYPEID_* set of #define's, and indicate which basic Java type this node represents. Examples are CVM_TYPEID_VOID, CVM_TYPEID_INT and CVM_TYPEID_OBJ.

Here's a full list of basic types:

```
#define CVM_TYPEID_NONE     0

#define CVM_TYPEID_ENDFUNC1

#define CVM_TYPEID_VOID2

#define CVM_TYPEID_INT3

#define CVM_TYPEID_SHORT4

#define CVM_TYPEID_CHAR5

#define CVM_TYPEID_LONG6

#define CVM_TYPEID_BYTE7

#define CVM_TYPEID_FLOAT8

#define CVM_TYPEID_DOUBLE9

#define CVM_TYPEID_BOOLEAN10

#define CVM_TYPEID_OBJ11
```

For reference, the entire set of node definitions can be found in `src/share/javavm/include/jit/jitirnode.h`

And the basic type definitions can be found in `src/share/javavm/include/typeid.h`

There are macros that encode tags to fit into an IR node.

The basic macro is `CVMJIT_TYPE_ENCODE(opcode,node,type)`, which encodes a 16-bit tag based on the parameters -- the triple that uniquely defines the 'kind' of an IR node.

Here are a few examples on how this works.

The first example is an integer type binary node. For that we have:

```
#define CVMJIT_ENCODE_IBINARY(opcodeTag) \
  CVMJIT_TYPE_ENCODE(opcodeTag,
   CVMJIT_BINARY_NODE, CVM_TYPEID_INT)
```

Here we are setting the type to int, and the subclass to binary. But we are leaving the opcode open.

Here's how an integer addition would be encoded:

```
CVMJIT_ENCODE_IBINARY(CVMJIT_ADD)
```

The `CVMJIT_BINARY_NODE` subclass tag would tell the JIT to select the `CVMJITBinaryOp` part of the `CVMJITIRSubclassNode` union of the IR node. This is defined as:

```
 typedef struct {
  CVMJITIRNode* lhs;
  CVMJITIRNode* rhs;
  CVMUint16              data; /* e.g. true argsize of invoke node */
  CVMUint8    data2; /* more node specific data. */
  CVMUint8    flags; /* CVMJITCMPOP_UNORDERED_LT|
                               CVMJITBINOP_ALLOCATION */
 } CVMJITBinaryOp;
```

Ignoring the last word, the binary operation has two arguments, the left-hand side node and the right-hand side node. The tag already encodes an integer addition operation. The arguments to the integer addition are the nodes pointed to by 'lhs' and 'rhs', which themselves may be composed of other subnodes.

In another example, to encode a local variable node kind, we would use `CVMJIT_LOCAL` as the opcode and `CVMJIT_LOCAL_NODE` as the node subclass. The type of local (integer vs. float, for example) would be indicated by the type_tag field.

So we would have:

```
#define CVMJIT_ENCODE_LOCAL(typeTag) \
 CVMJIT_TYPE_ENCODE(CVMJIT_LOCAL,
  CVMJIT_LOCAL_NODE, typeTag)
```

and now we can specialize an integer local node in the following way:

```
  CVMJIT_ENCODE_LOCAL(CVM_TYPEID_INT)
```

Here, `CVMJIT_LOCAL` is the opcode, and `CVMJIT_LOCAL_NODE` is the subclass selector.

The `CVMJIT_LOCAL_NODE` subclass tag would tell the JIT to select the `CVMJITLocal` part of the `CVMJITIRSubclassNode` union of the IR node. This is simply defined as:

```
  typedef struct {
               CVMUint16   localNo;
  } CVMJITLocal;
```

3.5 Conversion

Now we outline how byte-codes are converted to IR nodes. We first treat this phase for individual expressions. In the next section, we define how the compiler performs its passes to initiate this conversion for a given method.

The conversion of byte-codes to IR nodes involves emulation of the stack-based byte-code architecture, and extracting a value-based intermediate representation from the emulation.

To go back to our initial example, let's consider the method `c.f(I)`:

```
aload_0   // this
getfield 2 // _foo
iload_1   // k
iadd    // _foo + k
ireturn
```

We emulate the byte-codes' effect on the stack. But our emulation serves a tracking purpose only. We maintain an internal "stack" of IR nodes to figure out what the byte-codes would be doing to the runtime stack were they to be executed at runtime.

So `aload_0` accesses local 0, and pushes it onto the stack. When the translator sees `aload_0`, this is what it does, in effect:

```
loc = CVMJITirnodeNewLocal(CVMJIT_ENCODE_LOCAL
 (CVM_TYPEID_OBJ), 0);
CVMJITirnodeStackPush(loc);
```

This in effect creates an object-typed local node, sets its local number to 0 (since this node represents local 0), and pushes it on our emulation stack.

The next byte-code is getfield 2, which is defined as popping the top-most item on the stack, and accessing field #2 out of it. So we pop the top item on the emulation stack, which is "LOCAL 0" (the node), and we create a field read node off of it.

The '2' becomes a "constant node":

```
fieldOffsetNode = CVMJITirnodeNewConstantFieldOffset(2);
```

The popped "LOCAL 0" is set to be `objrefNode`:

```
objrefNode = CVMJITirnodeStackPop();
```

And now we create:

```
fieldRefNode =
 CVMJITirnodeNewBinaryOp(CVMJIT_ENCODE_FIELD_REF
    (CVM_TYPEID_32BITS), objrefNode, fieldOffsetNode)
```

So basically we have created a binary node:

```
 fieldRefNode --->  FIELDREF32
/   \
LOCAL 0 CONST 2
```

We now want to do a read off of this field, since that is what getfield is supposed to do. So we create a read:

```
fetchNode =
CVMJITirnodeNewUnaryOp(
CVMJIT_ENCODE_FETCH(CVM_TYPEID_32BITS),
fieldRefNode);
```

The read is a 32-bit read off of an object, which is represented as a unary operation. So we now have:

```
   fetchNode --->  FETCH32
                    |
                FIELDREF32
                /   \
              LOCAL 0 CONST 2
```

So now fetchNode is a pointer to the head of this "compound" node. We now push fetchNode back on our emulation stack:

The iload_1 instruction that follows pushes "LOCAL 1" onto the emulation stack. And the addition operation simply creates a binary integer addition, and sets the arguments of the binary node to be the two top values on the stack. Which leaves us with the following, which is pushed back onto the stack:

```
                  IADD
                 /   \
               FETCH32  LOCAL 1
                 |
               FIELDREF32
                /   \
              LOCAL 0 CONST 2
```

And finally, we get the return statement. We pop the IADD expression of the emulation stack, we attach to it IRETURN, and we push it back.

```
                 IRETURN
                   |
                  IADD
```

```
          /   \
        FETCH32  LOCAL 1
          |
        FIELDREF32
         /   \
       LOCAL 0 CONST 2
```

So we basically iterate over the byte-codes to be translated, we keep track of the stack state of the byte-codes using our own emulation stack, and we compose expression trees that represent the original computation. By composing the tree in the correct order (and expecting traversal to happen in the right order), we make sure that Java semantics and evaluation order are followed correctly.

3.6 Example Expression Trees

Now we can go through a few common byte-code sequences, and show what the resultant IR looks like. Before we do that, a few words about tracing bytecode to IR translation (enable when the build option `CVM_TRAE_JIT=true`). The command line option “`-Xjit:trace=bctoir`” traces the IR nodes created for compiled methods. The display is hierarchical, where each level in an expression tree is indicated with indentation.

So the following tree:

```
       NODE1
       /  \
    NODE2 NODE3
    /  \
  NODE4 NODE5
```

would appear as:

```
  <(ID: 1) NODE1
    <(ID: 2) NODE2
     <(ID: 3) NODE3
     <(ID: 4) NODE4
    <(ID: 5) NODE5
```

3.6.1 Arithmetic Operation

Arithmetic operations are the easiest type of expression trees to construct. Here's an example:

```
 public static int arith(int x, int y, int z) {
  return x * y + z;
 }
```

The bytecodes for this are:

```
<0> iload_0
<1> iload_1
<2> imul
<3> iload_2
<4> iadd
<5> ireturn
```

Where local 0, 1, and 2 correspond to x, y and z respectively.

The corresponding IR tree is:

```
  <(ID: 7) RETURN_VALUE (int)
   <(ID: 6) ADD (int)
     <(ID: 4) MUL (int)
      <(ID: 2) LOCAL (int)  0>
      <(ID: 3) LOCAL (int)  1>
     <(ID: 5) LOCAL (int)  2>
```

which corresponds to:

```
     RETURN (int)
      |
     ADD(int)
    /   \
  MUL(int)  LOCAL2
  /  \
LOCAL0  LOCAL1
```

3.6.2 Object Access With Null Check

Here’s an example of an access to a field:

```
class C {
  Object f;
  [...]
}
class C2 {
  public static Object accessField(C x)
  {
    return x.f;
  }
}
```

The corresponding byte-codes are:

```
<0> aload_0
<1> getfield <C.x>
<4> areturn
```

Assume for now that the offset of the field C.x is 2 words from the header (i.e. the first data word in the object). So the `getfield` below would refer to the constant 2 as the field offset.

The generated IR tree (ignore for now the TEMP and IDENTITY nodes. They are going to be explained later on).

```
  <(ID: 7) TEMP (NONE)
   <(ID: 6) IDENTITY (reference) (ref count: 2)
     <(ID: 9) FETCH (reference)
      <(ID: 5) FIELD_REF (reference)
        <(ID: 4) NULL_CHECK (reference)
         <(ID: 2) LOCAL (reference)  0>
        <(ID: 3) CONST_FIELD_OFFSET (NONE) (2)
  <(ID: 8) RETURN_VALUE (reference)
   <(ID: 6) IDENTITY (reference) (ref count: 2)
```

This corresponds to:

```
      TEMP-------------RETURN_VALUE
        |                    |
     IDENTITY <--------------/
        |
     FETCH
        |
     FIELD_REF
      /     \
  NULL_CHECK  CONST 2
    |
   LOCAL0
```

Note here that there are two root nodes in this statement. Traversing the first one forces the evaluation of the field access. The second one points to the first (which when traversed already holds the value of the field read), and returns that value.

Also note that we've introduced a unary node called "NULL_CHECK." This makes explicit what's only implicit in the byte-code semantics - that *before* a field access is performed, an explicit null check has to be performed. The traversal order of this tree preserves the original order of the implicit runtime check and the evaluation of the read. First the local node is evaluated. Then the NULL_CHECK node is evaluated, and causes code to be generated that tests the local node for null and throws an exception if it is. The field reference is only evaluated after the null check.

There are ways to eliminate explicit null checks. This will be explained later on.

3.6.3 Object Access With Lazy Class Initialization Check

The above example referred to a field reference in a class that was already loaded and initialized in the system. The use of the constant 2 for the field offset implies the field is well known at JIT-time (i.e. the class C is loaded and initialized when the method `C2.accessField()` is JIT'ed). But Java semantics require that a class not be initialized until its first active use. So it is now worth examining what would happen if the class C above were not initialized, and the offset of the field C.x was not known at JIT time. The JIT would then have to generate code that would resolve the field reference at runtime and figure out its offset. We can't "aggressively" resolve this offset at JIT-time, because that would involve actually initializing the class C, which is forbidden by Java semantics.

So here's the IR tree (the byte-code is unchanged):

```
<(ID: 8) RETURN_VALUE (reference)
   <(ID: 7) FETCH (reference)
     <(ID: 6) FIELD_REF (reference)
```

```
      <(ID: 5) NULL_CHECK (reference)
        <(ID: 2) LOCAL (reference)  0>
      <(ID: 4) RESOLVE_REFERENCE (NONE)
        <(ID: 3) CONST_GETFIELD_FB_UNRESOLVED (NONE)
         (cpIndex 2)
The tree looks similar to the above:
    RETURN_VALUE
        |
     FETCH
        |
     FIELD_REF
      /     \
  NULL_CHECK  RESOLVE_REFERENCE
    |         |
   LOCAL0   CONST_GETFIELD_FB_UNRESOLVED (cpIndex: 2)
```

Note that the constant field reference has been replaced with a RESOLVE_REFERENCE opcode that refers to a constant pool entry for the field to be accessed. The code generator will replace this with a call to a runtime routine that will resolve the required reference "lazily," only when it's first executed.

Note once again that implicit Java semantics have been replaced with explicit operations that preserve the original order of operations. The null check is done first. The reference resolution is done next. The field reference is executed only if the two succeed without an exception thrown. An exception due to a null check has to happen before an exception due to a reference resolution. All this is implicit in 'getfield' but explicit in the IR.

3.6.4 Array Access With Runtime Checks

Now an example of an array access expression:

```
public static Object accessArray(Object[] arr, int idx)
{
 return arr[idx];
}
```

The bytecodes for this are:

```
<0> aload_0
<1> iload_1
```

```
<2> aaload
<3> areturn
And the IR tree is:
  <(ID: 10) RETURN_VALUE (reference)
   <(ID: 9) FETCH (reference)
     <(ID: 7) INDEX (int)
      <(ID: 2) IDENTITY (reference) (ref count: 2)
        <(ID: 8) LOCAL (reference)  0>
      <(ID: 6) BOUNDS_CHECK (NONE)
        <(ID: 3) LOCAL (int)  1>
        <(ID: 5) ARRAY_LENGTH (int)
         <(ID: 4) NULL_CHECK (reference)
           <(ID: 2) IDENTITY (reference) (ref
            count: 2)
```

which corresponds to:

```
            RETURN_VALUE
                |
              FETCH
                |
              INDEX
             /   \
       /---> IDENTITY   BOUNDS_CHECK
       |    |       /    \
       |   LOCAL 0   LOCAL 1  ARRAY_LENGTH
       |                 |
       |               NULL_CHECK
       |                 |
       \-----------------------------/
```

Note once again that this expression explicitly reflects what's implicit in `aaload`. The array reference 'arr' first has to be evaluated (it can be composed of an expression with side effects so its evaluation might throw an exception). Then the index expression has to be evaluated (it too might be an expression with side effects, so its evaluation might throw an exception). Then a `NULL_CHECK` has to be performed on the array reference. Finally a bounds check has to be performed for the `arr[idx]` reference, so the `BOUNDS_CHECK` node is generated, against the index expression and the length of the array reference 'arr'.

When all these checks are performed, we can compute `&arr[idx]` (the INDEX node) and do a read off of that (the FETCH node). Any exceptions thrown in the evaluation of the array and index expressions, and the null and bounds checks all have to be performed in the order prescribed by the JVM specification.

3.6.5 Method Invocation With Parameters and Return Value

And finally, here's an example of a method invocation, with parameter passing.

```
class C {
  public static int f(int arg1, int arg2, int arg3)
  {
    return arg1 + arg2 + arg3;
  }
  public static int g(int v1, int v2, int v3) {
    int result = f(v1, v2, v3);
    return result * result;
  }
}
```

The bytecodes for g() involve an invocation to f():

```
<0> iload_0
<1> iload_1
<2> iload_2
<3> invokestatic #3  // f(III)
<6> istore_3
<7> iload_3
<8> iload_3
<9> imul
<10> ireturn
```

The IR tree is:

```
  <(ID: 12) TEMP (NONE)
   <(ID: 11) IDENTITY (int) (ref count: 3)
     <(ID: 14) INVOKE (int)
      <(ID: 10) PARAMETER (int)
        <(ID: 2) LOCAL (int)  0>
        <(ID: 9) PARAMETER (int)
         <(ID: 3) LOCAL (int)  1>
         <(ID: 8) PARAMETER (int)
           <(ID: 4) LOCAL (int)  2>
           <(ID: 7) NULL_PARAMETER (NONE)
      <(ID: 6) RESOLVE_REFERENCE (NONE)
        <(ID: 5) CONST_STATIC_MB_UNRESOLVED (NONE) (cpIndex 3)
  <(ID: 15) RETURN_VALUE (int)
   <(ID: 13) MUL (int)
     <(ID: 11) IDENTITY (int) (ref count: 3)
     <(ID: 11) IDENTITY (int) (ref count: 3)
```

which corresponds to:

```
TEMP------------------------------------>  RETURN_VALUE
     |                         |
     |                        MUL
   IDENTITY< <--                    / \
     |   \  \-----------------------------------------
    INVOKE  ----------------------------------------
    /  \
    /   RESOLVE_REFERENCE
   /         |
 PARAMETER  CONST_STATIC_MB_UNRESOLVED(3)
  /   \
LOCAL0  PARAMETER
     /   \
   LOCAL1  PARAMETER
        /    \
     LOCAL2  NULL_PARAMETER
```

A few things to note: Once again, there is a TEMP root node that evaluates the invocation. The next statement, the return statement, points to the evaluated invocation nodes.

Parameters are passed using a list composed of multiple binary PARAMETER nodes. The left-hand side of a PARAMETER node points to an individual parameter, and the right hand side points to either the next parameter in line, or the NULL_PARAMETER node. Evaluating this list evaluates each parameter, and the code generator can set up the parameter list in preparation for the invocation.

3.7 Conversion Passes

This section talks about the implementation of byte-code to IR translation: the entry point for the translation, the passes that are performed, and functions of interest.

The entry point for compilation is the function `CVMJITcompileMethod()` in `src/share/javavm/runtime/jit/jitcompile.c`. This function is called by the runtime system on a compilation trigger. In pseudo-code, it does roughly the following:

```
LOCK(jitLock);   // Allow one compilation at a time
                  InitializeContext(&con);
                  CompileBytecodeToIR(&con);
                  CompileOptimizeIR(&con);
                  AllocateCodeBufferSpace(&con);
                  GenerateCode(&con);
                  WriteStackmaps(&con);
                  FinalizeMethod();
                  FreeWorkingMemory();
UNLOCK(jitLock);
```

The byte-code to IR translation is done in the `CompileBytecodeToIR()` stage. The entry point function is `CVMJITcompileBytecodeToIR()` in `src/share/javavm/runtime/jit/jitir.c`. The IR is created at this stage.

The bulk of the work is done in `jitir.c:translateMethod()`.

The function `firstPass()` handles the preparation for the translation. This is the block discovery phase. The byte-codes are scanned. Extended basic blocks are identified. These are defined to be sequences of byte-codes with the only incoming flow of control at the head, but with potentially multiple outflows of control. It is possible to trivially identify extended basic blocks by iterating over the code and following branches. An incoming branch makes an instruction a block header. When there are no more branches to traverse, all blocks have been discovered.

In the Java world, there are also exception handler blocks. These are not discovered via the traversal of branches, but by looking at exception tables in class files. Exception handlers are also marked as blocks.

Each block is represented by the data structure `CVMJITIRBlock`. Blocks are linked together in a list, and arranged in byte-code pc order. So iterating over the blocks follows the original order of the byte-codes.

After the first pass is over, we have a list of ordered blocks. Machine code generation follows this block order as well, which ensures that generated code follows the original byte-code order.

The second pass involves the actual translation. Here we don't actually follow the pc order, but flow order instead. We have a queue of blocks to be compiled. The first block is placed on the queue. Translation then starts by retrieving the head block from the queue. Every branch to a new block causes it to be placed on the compilation queue. When there are no more queued blocks to process we are done.

Each block is translated in `translateBlock()`. Due to inlining (more on that later), the translation unit is the "range" -- a range of byte-codes that could belong to the method currently being compiled, or any of the methods it inlines. translateRange() is the function that drives the byte-code to IR conversion. It handles each byte-code in a byte-code range and translates it to IR trees. It emulates byte-code execution via the use of an "IR node stack" to facilitate IR tree creation.

The root nodes that are created are all added to the block being translated. Once all blocks are completely translated, byte-code to IR translation is completed, and code generation takes over.

3.8 Simple Inlining

Assume we have a simple accessor method:

```
  private int getSize() {
                return size;
  }
```

And a block of code that calls it using a nonvirtual invocation:

```
Block B0
  [ ... ]
L0:
  x = getSize();
L1:
  [ ... ]
```

Inlining effectively rewrites the invocation, resulting in an equivalent list of inlined statements:

```
Block B0
  [ ... ]
L0:
  x = size;
L1:
  [ ... ]
```

Conceptually, the bytecodes for the inlinable target method are translated as if they encountered at the call site. This is very similar to macro expansion in the C preprocessor. In the above example, there are no branches in the target method, so inlining did not cause new blocks to be created. But in general, inlining can cause new blocks to be created.

3.9 Simple Multi-block Inlining

Suppose the target method is more complicated, requiring branches:

```
  private int getSize(int n) {
               if (n > 0) {
                 return size;
               } else {
                 return 0;
               }
  }
Block B0
L0:
  [ ... ]
L1:
  x = getSize(1);
L2:
  [ ... ]
```

To correctly inline the reference at L1, Block B0 must be split:

```
Block B0
```

```
L0:
  [ ... ]
  /* fall through to next block */
Block B1
L1:
  /* getSize() prologue */
  ARG1 = 1
  /* fall through to next block */
Block B2
  /* expansion of getSize() */
  if (ARG1 > 0) {               // if (n > 0) {
               RETVAL = size; //   return size;
               goto R0;
  } else {                      // } else {
Block B3
R1:
               RETVAL = 0;//  return 0;
               goto R0;
  }                         // }
Block B4
R0:
  /* getSize() epilogue */
  /* fall through to next block */
Block B5:
  x = RETVAL;                   // x = getSize(1);
L2:
  [ ... ]
```

In the example, the current block is split into 6 different blocks. First, the block is split into three parts, pre-invocation, invocation, and post-invocation. Finally, the invocation block is split into separate prologue, expansion, and epilogue blocks. Because the method contains an if-then-else, the expansion block is again split into two blocks. For this example, there is no more block splitting required. For nested inlining, the process would be repeated inside the expansion block if it contained inlinable invocations.

The prologue block is where incoming arguments are copied into local variables. Also, if the target method is synchronized, a monitor-enter operation is performed.

The epilogue block is where the target method returns to. All “returns” in the inlined method are converted into “gotos” to the epilogue block. A monitor-exit will be performed if the inlined method was synchronized.

During translation, split blocks are rejoined wherever possible. In the above example, blocks B0, B1, and B2 will be rejoined, and block B4 and B5 will also be rejoined, resulting in:

```
Block B0
L0:
  [ ... ]
L1:
  /* getSize() prologue */
  ARG1 = 1
  /* expansion of getSize() */
  if (ARG1 > 0) {              // if (n > 0) {
               RETVAL = size; //   return size;
               goto R0;
  } else {                     // } else {
Block B3
R1:
               RETVAL = 0;//  return 0;
               goto R0;
  }                        // }
Block B4
R0:
  /* getSize() epilogue */
  x = RETVAL;                  // x = getSize(1);
L2:
  [ ... ]
```

3.10 Virtual Inlining

In the above examples, the invocation was always nonvirtual, because the target method was known as compile time. For virtual invocations, the actual target method may not be known at compile time. The following is an example of a virtual invocation due to method overriding.

```
class A {
  int getSize(int n) {
                if (n > 0) {
                  return size;
                } else {
                  return 0;
                }
  }
}
class B extends A {
  int getSize(int n) {
                if (n > 2) {
                  return size;
                } else {
                  return 0;
                }
  }
}
Block B0
L0:
  A a;
  [ ... ]
L1:
  x = a.getSize(1);         // virtual invocation
L2:
  [ ... ]
```

The inline a virtual invocation, the call site is rewritten to the following:

```
L1:
  // compare method block pointers
  if (a.getSize == A.getSize) {
L3:
                x = a.getSize(1);// nonvirtual invocation
  } else {
L4:
                x = a.getSize(1);// virtual invocation
  }
```

```
L2:
```

We “guess” that `a.getSize()` is calling `A.getSize()` instead of `B.getSize()`. If our guess is right, we perform a nonvirtual invocation, which gets inlined. If the guess is wrong, we perform a virtual invocation. For performance reasons, the virtual invocation block at L4 is moved “out-of-line” to the end of the method, resulting in the following before inlining:

```
Block B0
L0:
  A a;
  [ ... ]
L1:
  // compare method block pointers
  if (a.getSize != A.getSize) {
               goto L4;
  }
  x = a.getSize(1);         // nonvirtual invocation
L2:
  [ ... ]
Block B1
L4:
  x = a.getSize(1);         // virtual invocation
  goto L2;
```

And finally, after inlining:

```
Block B0
L0:
  A a;
  [ ... ]
L1:
  // compare method block pointers
  if (a.getSize != A.getSize) {
               goto L4;
  }
  // expansion of "x = a.getSize(1);" nonvirtual invocation
  /* getSize() prologue */
  ARG1 = 1
  /* expansion of getSize() */
  if (ARG1 > 0) {               // if (n > 0) {
                RETVAL = size; //   return size;
                goto R0;
  } else {                      // } else {
Block B2:
R1:
                RETVAL = 0;//  return 0;
                goto R0;
  }                         // }
Block B3
R0:
  /* getSize() epilogue */
  x = RETVAL;                   // x = getSize(1);
L2:
  [ ... ]
Block B1                // out-of-line virtual invocation
L4:
  x = a.getSize(1);         // virtual invocation
  goto L2;
```

3.11 Nested and Recursive Inlining

In the above examples, the method being inlined did not call any other methods. But suppose method A calls method B, and we inline method A. During the translation of method A, we encounter the invocation of method B. If method B is also inlinable, the inlining process begins again, recursively. A stack of these inlined method contexts is maintained during this process.

```
  int B() {
                return 5;
  }
  int A() {
                return B();
  }
Block B0
  [ ... ]
```

```
L0:
  x = A();
L1:
  [ ... ]
```

inlined to:

```
Block B0
  [ ... ]
L0:
  // begin inlined method A
  RETVAL_A = B();
  // end inlined method A
  x = RETVAL_A;
L1:
  [ ... ]
```

and recursively inlined to:

```
Block B0
  [ ... ]
L0:
  // begin inlined method A
                // begin inlined method B
                RETVAL_B = 5;
                // end inlined method B
  RETVAL_A = RETVAL_B;
  // end inlined method A
  x = RETVAL_A;
L1:
  [ ... ]
```

If `A()` had called itself instead of `B()`, the invocations would have been recursive. The expansion could potential have an unlimited depth. The front end has tunable parameters for controlling inlining depth, and uses heuristics to sometimes stop inlining before the hard limits are reached.

3.12 Method Contexts

The front end uses a `CVMJITMethodContext` data structure to keep track of inlining information each time an inlinable method is expanded. In the above example, block B0 contains tree method contexts. The outer context belonging to the original method being compiled, the context for inlined method A, and the context for inlined method B. When an inlined method refers to LOCAL N, the method context is used to map it to the appropriate local in the outermost context. So LOCAL 0 in method A might map to LOCAL 5 in the generated code.

3.13 Argument Handling and Locals

Incoming arguments for an inlined methods are converted into locals. They are assigned in the prologue block. As each new method context is pushed, the range of locals mapped to that context increases. When translation reaches the end of the inlined method, the range of locals is popped so that they can be reused by another inlined method. So inlined method A might use LOCAL 5 through 6. Method B, inlined from A, might use LOCAL 7 through 8. If A also called another method C, that method might use LOCAL 7 through 10. After calling A, the outmost method (call it X) might call D, using locals 5 through 9:

```
/* local map XXXXX */
  // begin inlined method A, locals 5 - 6
/* local map XXXXXAA */
                 // begin inlined method B, locals 7 - 8
/* local map XXXXXAABB */
                 // end inlined method B
/* local map XXXXXAA */
                 // begin inlined method C, locals 7 - 10
/* local map XXXXXAACCCC */
                 // end inlined method C
/* local map XXXXXAA */
  // end inlined method A
/* local map XXXXX */
  // begin inlined method D, locals 5 - 9
/* local map XXXXXDDDDD */
```

```
  // end inlined method D
/* local map XXXXX */
```

It should be noted that passing a constant to an inlinable method might allow extra optimizations to be performed. For example, getSize(1) would expand to

```
if (1 > 0) {
  return size;
} else {
  return 0;
}
```

which would be simplified into:

```
return size;
```

Likewise, returning a constant from an inlined method will also be optimized. Sometimes the method did not return a liternal constant, but due to other optimizations, return value turned into a constant:

```
if (n == 0) {
  return size;
} else {
  return n;
}
```

turns into:

```
if (1 == 0) {
  return size;
} else {
  return 1;
}
```

and simplifies into:

```
return 1;
```

when n == 1. These are some of the many benefits of inlining.

3.14 Limiting Inlining

The depth of inlining is one tunable parameter (`-Xjit:maxInliningDepth`) that effects the amount of code generated for each compiled method. Because any method can be inlined, unlimited inlining would quickly fill up the code cache. The front end use heuristics to decide how much inlining to do. Sometimes, especially if recursive inlining is involved, the IR generated by the front end results in generated code that is too big. In this case, we can ask for a bigger code buffer. But there is a reasonable limit on code buffer size. In order to allow the method to be compiled into, the front end will sometimes need to retry the compile with a smaller inlining depth limit. So if the previous attempt failed at inlining depth 5, the next attempt might limit the depth to 4.

3.15 Runtime Inlining Information and `BEGIN/END_INLINING` Nodes

After a method is compiled, inlined method will not push a new frame on the stack. Instead they use the frame for the outermost method. But there are times when runtime system still needs to know, for a given program counter, what the call chain (or stack trace) would look like, including inlined methods. To support this, the front end generates `BEGIN_INLINING` and `END_INLINING` IR nodes. The `END_INLINING` node is attached to the return value for inlined methods that return a value. So in the example where we had

```
  x = RETVAL_A;
```

this will be represented in the IR as if it `RETVAL_A` was wrapped in an `END_INLINING` node:

```
  x = END_INLINING(RETVAL_A);
```

This ensures that the end of the inlined method is correctly identified.

The back end is required to match these IR nodes with the appropriate logical PC range in the generated code and store the range in a lookup table. At runtime, this lookup table is used to map (PC) to (inlining depth, method block), for a given compiled frame, allowing complete stack back trace and caller information to be reconstructed.

```
// begin method X
L0:
  // begin inlined method A, locals 5 - 6
```

```
L1:
                 // begin inlined method B, locals 7 - 8
L2:
                 // end inlined method B
L3:
                 // begin inlined method C, locals 7 - 10
L4:
                 // end inlined method C
L5:
  // end inlined method A
L6:
  // begin inlined method D, locals 5 - 9
L7:
  // end inlined method D
L8:
// end method X
```

If the PC values were as labeled above, the mappings for this frame would be:

```
L0 --> (depth 0, method X)
L1 --> (depth 1, method A)
L2 --> (depth 2, method B)
L3 --> (depth 1, method A)
L4 --> (depth 2, method C)
L5 --> (depth 1, method A)
L6 --> (depth 0, method X)
L7 --> (depth 1, method D)
L8 --> (depth 0, method X)
```

CHAPTER 4

JavaCodeSelect

4.1 Introduction

When a method is being compiled, CDC-HI parses the method's bytecodes and builds an intermediate representation (IR) that can be further analyzed and optimized. Details of the IR are described elsewhere, but here's a quick overview: the method is a linked list of basic blocks; each basic block contains a linked list of directed acyclic graphs (DAGs), which can contain edges connecting one rooted DAG to another. The DAGs are carefully constrained, so that in general they can be viewed as trees.

Java Code Select (JCS) is a parser generator similar in purpose to YACC or Bison. Where YACC or Bison build parsers that perform pattern matching with streams, JSC produces parsers that performs pattern matching within tree-based data structures. JCS is based on the work of Graham, Henry, and Pelegri-Llopart [Pel88, Hen89].

Patterns are organized as a set of JCS rules similar to those of a parser generator such as YACC or Bison. These rules are processed by JCS when CDC-HI is built, resulting in files of C source code and initialized data. These are compiled at build time and linked into CDC-HI.

Code generation using rule-based pattern matching on trees is described in [Muchnick] and [AhoG89]. The principle difference between JCS and the pattern matching described there is that JCS determines which rules to apply using static costs, rather than dynamically computed costs. This means that JCS make more of the decisions at build time rather than when the compiler is running.

Code generation using rule-based pattern matching is easier to program when the set of instructions is at all complex or at all flexible (as it is for a retargetable code generator). This is because the code generator generator takes care of making sure that the best rule is used at any point, based on the cost metric.

4.2 Concepts

The following sections describe some of the basic concepts of JCS. Understanding these will help with the JCS Syntax section that follows.

4.2.1 Tokens, Terminals, and Nonterminals

JCS patterns are made up of grammatical terminals and grammatical nonterminals. The terminals represent nodes in the IR tree which matches a pattern. The nonterminals represent the result of some other rule. That is, a nonterminal indicates that another rule of a certain sort matched a subtree of the one matching this rule.

A terminal represents one of:

- a leaf node, which is an IR node having no subtrees, such as a local variable or constant
- a unary node, which is an IR node having one subtree, such as a unary negation operator
- a binary node, which is an IR node having two subtrees, such as a subtraction operator

Terminals and non-terminals are represented by <word>s in the JCS input. A <word> begins with a letter and is followed by zero or more letters, numerals, or underscore characters.

```
[a-zA-Z][a-zA-Z_0-9]*
```

The <word>s representing terminals are specified as part of the JCS input. All other <word>s used as parts of patterns or the result of rules (see below) are implicitly declared as nonterminals.

The JCS statements that specify terminals are these:

```
<statement> ::= '%leaf' <word>// specify a leaf-terminal

<statement> ::= '%unary' <word>// specify a unary-terminal

<statement> ::= '%binary' <word>// specify a binary-terminal
```

The <word>s specified as terminals are used in the C files produced by JCS as labels on switch statements.

4.2.2 Patterns, Pattern Specification, and Rules

A pattern is represented by a left-to-right, prefix form of the subtree it matches, as specified here:

```
<pattern> ::= <subtree>

<subtree> ::= <non-terminal>

<subtree> ::= <leaf-terminal>

<subtree> ::= <unary-terminal> <subtree>

<subtree> ::= <binary-terminal> <(left) subtree> <(right)
subtree>
```

A rule consists of several parts, the first three of which specify the rule's result (a nonterminal), its pattern, and a small number giving the rule's relative cost. Costs are used to break ties when two or more sets of rules could be used to match the same tree. A rule's parts are separated by colons.

```
<statement> ::= <rule>

<rule> ::= <result> ':' <pattern> ':' <cost> ':' ...

<result> ::= <non-terminal>

<cost> ::= <number>
```

The number of non-terminals in a pattern is sometimes called its arity. Although IR nodes are constrained to being at most binary, there is no limit on the arity of a pattern. A rule is called a chain rule if its pattern consists solely of a single non-terminal.

4.2.2.1 Example 1

Here is a simple example of partial JCS input. It specifies simple arithmetic on and assignment to variables. Note that the full form of a rule has not yet been fully specified.

```
%leaf   LOCAL32

%unary INEG32

%binary IADD32

%binary ISUB32

%binary ASSIGN
```

```
statement : ASSIGN LOCAL32 reg32 : 1 : ... // 1: assignment of
value to variable

reg32: LOCAL32 : 1 : ... // 2: a variable's value

reg32: INEG32 reg32 : 1 : ...  // 3: the negation of a value

reg32: IADD32 reg32 reg32 : 1 : ... // 4: the sum of two values

reg32: ISUB32 reg32 reg32 : 1 : ... // 5: the difference of two
values
```

What Do These Names Represent?

The examples here follow the naming conventions of the production grammar. Grammatical nonterminals are in lower or mixed case, grammatical terminals are upper case.

- `‘reg32’` represents any 32-bit value -- conceptually it is in a register during execution of a RISC instruction, but the Register Manager (q.v.) is actually allowed to move it between registers and temporary locations on the stack.
- `‘IADD32’` is a 32-bit integer add operation. In general, the prefix `I` indicates an integer operation, and the suffix `32` represents a 32-bit operation. ’INEG32’ and ’ISUB32’ follow this pattern.
- `‘LOCAL32’` is a 32-bit local variable of indeterminate type.

4.2.2.2 Example 1a

So far, the costs have not helped us, since there is only one rule for each operation. Consider this input tree:

```
                ASSIGN(a)
                  /     \
               LOCAL32(b) IADD32(c)
                           /   \
                     LOCAL32(d) INEG32(e)
                                   \
                                LOCAL32(f)
```

According to the costs and rules given above, this tree will take 5 units of cost to evaluate: rule 2 is applied to nodes d and f, rule 3 to node e, rule 4 at node c, and rule 1 at node a. If the addition of a negation appears often in the input, it may be worthwhile to recognize this as an opportunity to do subtraction. By adding this rule:

```
reg32: IADD32 reg32 INEG32 reg32 : 1 : ... // 6: difference of
2 values
```

the same tree can be matched with cost 4: rule 2 applied at nodes d and f, rule 6 at node c, then rule 1 at node a.

4.2.3 Code Generator Operation: Parsing the Tree

The parser generated by JCS does a bottom-up pattern match on a tree. This takes multiple passes over the tree.

4.2.3.1 Match Phase

Each node is labeled with a state number. A state represents the set of possible partial and full rules matches of the subtree rooted at the node.

Each leaf node is labeled with a state number based only on the node's own information, such as its operator and type.

Each interior node (binary or unary) is labeled with a state number based on the node's own information together with the state number(s) of its direct descendent(s).

4.2.3.2 Rule-based Phases

The rule-based phases are driven by node states and submatch goals. In order to successfully match any rule with nonterminals in its pattern, other rules having those nonterminals as results must first be matched. Thus at each step, nonterminals in the rule's pattern are used as the goals for subtree matches. Since a node's state represents a set of full and partial matches, knowing the goal for that node lets us select a fully-matched rule from that set which will result in the goal nonterminal. Costs are used to make the best selection.

Thus we can start at the top of the tree and recursively select rules which allow us to parse the entire tree.

During the rule-based phases, the only nodes visited are those which matched rules. Thus not every node is visited, and nodes which participate in chain rules may be visited multiple times.

Synthesis Phase

The synthesis phase gives us an opportunity to propagate information up the tree from children to parents. It is a bottom-up rule-based tree traversal. This is generally used for register targeting but there is nothing built into JCS specific to this task. Each rule specifies a C expression which will be applied to the node rooting each subtree which that rule will match.

Action Phase

The action phase is also a rule-based traversal of the tree. Each node rooting a submatch will be visited twice: once before visiting all submatches, and once after.

The previsit is called the inheritance action. Like synthesis, it is used for register targeting. It allows a rule to propagate information down to its children before those nodes are visited. It is specified as a C expression which will be applied to the node.

The postvisit is called the semantic action. It is used for code generation and other semantic bookkeeping. It is specified as a C statement.

Example 2

Given the grammar of Example 1 above, the match states are described below. In the following [...] represents the part of a rule's pattern matched by the state. If [...] encloses the entire pattern, it is a full match of that rule. Through closure, a full match of a rule induces us to add partial matches of further rules.

```
state #1: // for the leaf node LOCAL32
                 statement : ASSIGN [LOCAL32] reg32 // partial match
of rule 1
                 reg32: [LOCAL32]    // full match of rule 2
                 // the following added by closure
                 statement: ASSIGN LOCAL32 [reg32]
                 reg32: INEG32 [reg32]
                 reg32: IADD32 [reg32] reg32
                 reg32: IADD32 reg32 [reg32]
                 reg32: ISUB32 [reg32] reg32
                 reg32: ISUB32 reg32 [reg32]
state #2: // for the unary node INEG32 and its reg32 submatches
                 reg32: [INEG32 reg32] // full match of rule 3
                 // the following added by closure
                 statement: ASSIGN LOCAL32 [reg32]
                 reg32: INEG32 [reg32]
                 reg32: IADD32 [reg32] reg32
                 reg32: IADD32 reg32 [reg32]
                 reg32: ISUB32 [reg32] reg32
                 reg32: ISUB32 reg32 [reg32]
state #3: // for the binary node IADD32 and its reg32 submatches
                 reg32: [IADD32 reg32 reg32] // full match of rule 4
                 // the following added by closure
                 statement: ASSIGN LOCAL32 [reg32]
                 reg32: INEG32 [reg32]
                 reg32: IADD32 [reg32] reg32
                 reg32: IADD32 reg32 [reg32]
                 reg32: ISUB32 [reg32] reg32
                 reg32: ISUB32 reg32 [reg32]
state #4: // for the binary node ISUB32 and its reg32 submatches
                 reg32: [ISUB32 reg32 reg32] // full match of rule 5
                 // the following added by closure
                 statement: ASSIGN LOCAL32 [reg32]
                 reg32: INEG32 [reg32]
                 reg32: IADD32 [reg32] reg32
                 reg32: IADD32 reg32 [reg32]
                 reg32: ISUB32 [reg32] reg32
                 reg32: ISUB32 reg32 [reg32]
state #5: // for the binary node ASSIGN and its reg32 submatch
                 statement : [ASSIGN LOCAL32 reg32] // full match of
rule 1
```

Here are the match phase transition tables. (-1) is the error state.

The leaf node LOCAL32 requires no transition table: it is immediately labeled with state #1.

```
INEG32: descendent| result
     state # | state #
      1                     2
                   2       2
                   3       2
                   4       2
                   5      -1
```

```
IADD32:               right descendent state #
                     1  2  3  4  5
     \_____________________________
left  1 |  3  3  3  3 -1
desc. 2 |  3  3  3  3 -1
state 3 |  3  3  3  3 -1
 #  4 |  3  3  3  3 -1
    5 | -1 -1 -1 -1 -1
```

ISUB32: same as IADD32 but substitute state #4 for #3 in the result values.

```
ASSIGN:                right descendent state #
                     1  2  3  4  5
     \_____________________________
left  1 |  5  5  5  5 -1
desc. 2 | -1 -1 -1 -1 -1
state 3 | -1 -1 -1 -1 -1
 #  4 | -1 -1 -1 -1 -1
    5 | -1 -1 -1 -1 -1
```

This is the table mapping the state number and goal into rule numbers:

```
                       STATE #
                     1 2 3 4 5
  GOAL \________________
                   |
  reg32 | 2 3 4 5 -1
                   |
statement | -1 -1 -1 -1 1
```

Here is the state assignment for the example tree:

```
                       ASSIGN(5)
                       /     \
                    LOCAL32(1) IADD32(3)
                           /   \
                         LOCAL32(1) INEG32(2)
                                 \
                                LOCAL32(1)
```

Example 2a

Adding rule 6 from example 1a causes a lot of changes in the above states and tables. Changes are marked with a *. State #2 is now as follows:

```
state #2: // for the unary node INEG32 and its reg32 submatches
                 reg32: [INEG32 reg32] // full match of rule 3
*                reg32: IADD32 reg32 [INEG32 reg32] // partial match
of rule 6
                 // the following added by closure on full match of
rule 3
                 statement: ASSIGN LOCAL32 [reg32]
                 reg32: INEG32 [reg32]
                 reg32: IADD32 [reg32] reg32
                 reg32: IADD32 reg32 [reg32]
                 reg32: ISUB32 [reg32] reg32
                 reg32: ISUB32 reg32 [reg32]
*                reg32: IADD32 [reg32] INEG32 reg32
*                reg32: IADD32 reg32 INEG32 [reg32]
```

To states #1, 3 and 4 the partially matched states now include the following due to closure:

```
*   reg32: IADD32 [reg32] INEG32 reg32

*   reg32: IADD32 reg32 INEG32 [reg32]
```

and state #6 is added as:

```
*  state #6: // for a full match of rule #6
*                reg32: [IADD32 reg32 INEG32 reg32]
*                // the following added by closure on full match of
rule 6
*                statement: ASSIGN LOCAL32 [reg32]
*                reg32: INEG32 [reg32]
*                reg32: IADD32 [reg32] reg32
*                reg32: IADD32 reg32 [reg32]
*                reg32: ISUB32 [reg32] reg32
*                reg32: ISUB32 reg32 [reg32]
*                reg32: IADD32 [reg32] INEG32 reg32
*                reg32: IADD32 reg32 INEG32 [reg32]
```

The transition tables are as follows:

LOCAL32 remains as state #1

```
INEG32: descendent| result
     state # | state #
                    1       2
                    2       2
                    3       2
                    4       2
                    5      -1
                   *6      *2
IADD32   right descendent state #
                     1  2  3  4  5 *6
     \______________________________
left  1 |  3 *6  3  3 -1 *3
desc. 2 |  3 *6  3  3 -1 *3
state 3 |  3 *6  3  3 -1 *3
 #  4 |  3 *6  3  3 -1 *3
    5 | -1 -1 -1 -1 -1 *3
   *6 | *3 *6 *3 *3 *-1 *3
ISUB32:   right descendent state #
                     1  2  3  4  5 *6
     \______________________________
left  1 |  4  4  4  4 -1 *4
desc. 2 |  4  4  4  4 -1 *4
state 3 |  4  4  4  4 -1 *4
 #  4 |  4  4  4  4 -1 *4
    5 | -1 -1 -1 -1 -1 *-1
   *6 | *4 *4 *4 *4 *-1 *4
ASSIGN:             right descendent state #
                     1  2  3  4  5 *6
     \______________________________
left  1 |  5  5  5  5 -1 *5
desc. 2 | -1 -1 -1 -1 -1 *-1
state 3 | -1 -1 -1 -1 -1 *-1
 #  4 | -1 -1 -1 -1 -1 *-1
    5 | -1 -1 -1 -1 -1 *-1
   *6 | *-1 *-1 *-1 *-1 *-1 *-1
```

This is the table mapping state number and goal into rule numbers:

```
                        STATE #
                    1 2 3 4 5 *6
  GOAL \___________________
                  |
  reg32 | 2 3 4 5 -1 *6
                  |
statement | -1 -1 -1 -1 1 *-1
```

And finally, we can make a table of prerequisite actions for each rule. These can easily be derived from inspecting the rules alone. In this table, the path is an encoding of a path to follow from the root node of the subtree we want to match with the rule:

```
                 L -- left subtree
                 R -- right subtree
                 U -- unary subtree
```

The goal gives the nonterminal we want to be the result of a subtree match at that point. Note that a rule can have no prerequisites, or multiple prerequisites.

```
                    path   goal
  rule # \_______________
   1  | R   reg32
   3  | U   reg32
   4  | L   reg32
                  & | R   reg32
   5  | L   reg32
                  & | R   reg32
   6  | L   reg32
                  & | RU   reg32
```

Here is the state assignment for the example tree:

```
                       ASSIGN(5)
                      /      \
                    LOCAL32(1) IADD32(*6)
                          /   \
                        LOCAL32(1) INEG32(2)
                              \
                             LOCAL32(1)
```

To parse this tree the algorithm would proceed as follows:

At the top of the tree, we have a node in state 5 which we want to evaluate to goal statement. We will use rule 1 which requires that we first evaluate its right subtree into a reg32. So we recursively visit the IADD32 node to do so.

At the IADD32 node, we have a node in state 6 which we want to evaluate to goal reg32. We will use rule 6. Rule 6 requires that we first evaluate the left subtree into a reg32, then evaluate the unary subtree of the right subtree as a reg32. So we first recursively visit the left subtree.

At the LOCAL32 node we have a node in state 1 which we want to evaluate to goal reg32. We will use rule 2. Rule 2 has no prerequisites so we can execute its semantic action immediately and return.

Evaluation of the IADD32 node next recursively visits the unary subtree of the right subtree to evaluate it as a reg32.

At the LOCAL32 node we have a node in state 1 which we want to evaluate to goal reg32. We will use rule 2. Rule 2 has no prerequisites so we can execute its action immediately, and return.

All prerequisites of rule 6 have been met, so its action can be executed. We return.

All prerequisites of rule 1 have been met, so its action can be executed.

Having completed the parse, we return.

4.2.4 DAG Support

Thus far the examples have all been trees. JCS has very simple support for DAGs. When a rule is prefaced with '`%dag`', JCS generates code to tag the rule's root with additional state information. This allows it to avoid traversing the subtree multiple times in each of its phases. Multiple traversals would cause JCS to be very confused about the state of the subtree and the propagation of its information.

```
<statement> ::= '%dag' <rule>
```

Encountering a DAG rule subsequent to the first time causes recursion to be cut off. Though the synthesis and semantic actions occur, inheritance does not. A test is available allowing the semantic code to determine whether or not it is executing the rule in the context of this node for the first time. See the section on Working with DAGS in Code Generation for more information.

4.2.5 Conditional Compilation

Because one of the rule-selection criteria is cost, it is possible to write a grammar with rules that JCS would never use. This most often happens when a platform-specific rule overrides a higher-cost generic rule supplied by the RISC porting layer of CDC-HI.

In normal operation, JCS will not emit the code for such unreachable rules, and can be directed to generate #define symbols based on the rule's use. This allows the programmer to allow helper functions or data structures to compile or not, possibly reducing the footprint of the resulting code generator.

Conditionally excluded rules and helpers can be unconditionally added by `#define CVM_CG_EXPAND_UNREACHED_RULE`.

4.3 JCS Syntax

Now that all JCS's important features have been introduced, the full input syntax of JCS input files can be given.

- Blank lines are ignored.
- JCS comments are // to EOL.
- Comments in C statements are /*...*/.
- Each statement ends with a new line.
- Newlines may be embedded within a C expression or statement.
- A <word> is [a-zA-Z][a-zA-Z_0-9]*

```
<statement> ::= '%name' <word>
```

Specify <name>, which is used to compose the names of generated symbols in the output. This allows multiple JCS generated parsers in the same program.

```
<statement> ::= '%type' <word>
```

Specify <type>, the typedef name to be used for pointer-to-nodes in the generated output.

```
<statement> ::= '%goal' <word>
```

Specify the nonterminal which is the top-level goal of an entire tree match.

```
<statement> ::= '%opcode' <word>
```

Specify the name of an int-valued function or macro used to retrieve a node's operator. A pointer to the node is passed as its parameter. This can be a simple field-access macro or a function that combines opcode with type or other information to provide a different view of the IR.

```
<statement> ::= '%left' <word>
```

Specify the name of a function or macro used to retrieve a pointer to a binary node's left subtree or a unary node's only subtree. A pointer to the node is passed as its parameter.

```
<statement> ::= '%right' <word>
```

Specify the name of a function or macro used to retrieve a pointer to a binary node's right subtree. A pointer to the node is passed as its parameter.

```
<statement> ::= '%setstate' <word>
```

Specify the name of a function or macro used to store state to a node. A pointer to the node and an integer state number are passed as parameters.

```
<statement> ::= '%getstate' <word>
```

Specify the name of a function or macro used to retrieve a node's state. A pointer to the node is passed as its parameter.

```
<statement> ::= '%leaf' <word>

<statement> ::= '%unary' <word>

<statement> ::= '%binary' <word>
```

Specify terminals which are leaves, unary operators, or binary operators.

Hereafter, these are referred to as <leaf-terminal>, <unary-terminal> and <binary-terminal>. These are used in the generated output as switch statement case labels.

The above <statements> make up the declarative statements. All declarative statements must precede all rules.

```
<statement> ::= <rule>
<statement> ::= '%dag' <rule>
<rule> ::= <result> ':' <pattern> ':' <cost> ':' <synthesis-action>
':'
                    <inheritance-action> ':' <macro-list> ':'
<semantic-action>
<result> ::= <non-terminal>
<non-terminal> ::= <word>
```

A <non-terminal> is a <word> which is no a <leaf-terminal>, <unary-terminal> or <binary-terminal>.

```
<pattern> ::= <subtree>

<subtree> ::= <non-terminal>

<subtree> ::= <leaf-terminal>

<subtree> ::= <unary-terminal> <subtree>

<subtree> ::= <binary-terminal> <(left) subtree> <(right) subtree>

<cost>  ::= <number>
```

Decimal numbers only. Must be ? 0. Should be small.

```
<synthesis-action>  ::= <C-expr>    // a C expression

<synthesis-action>  ::=             // may be empty
```

If omitted, a default is used. It is one of the following:

i. For 0-ary rules, there is no default action

ii. DEFAULT_SYNTHESIS_CHAIN(con, $$) for chain rules

iii. DEFAULT_SYNTHESIS_ACTIONn(con, $$) for n-ary rules,

See below for the meaning of '$$' within a C-expr

```
<inheritance-action> ::= <C-expr>   // a C expression

<inheritance-action> ::=            // may be empty
```

If omitted, a default is used. It is one of the following:

i. For 0-ary rules, there is no default action

ii. DEFAULT_INHERITANCE_CHAIN(con, $$) for chain rules

iii. DEFAULT_INHERITANCE_ACTIONn(con, $$) for n-ary rules,

See below for the meaning of '$$' within a C-expr

```
<macro-list> ::=         // may be empty

<macro-list> ::= <word> <macro-list>
```

This is the list of macros associated with the rule that will govern conditionally compiled helper code. For each macro associated with a reachable rule, the following will appear in the generated header file (see section on Output):

```
#ifndef <word>

#define <word>

#endif
```

For each macro associated with an unreachable rule, the following will appear in the generated header file:

```
#ifdef CVM_CG_EXPAND_UNREACHED_RULE
#ifndef <word>
#define <word>
#endif
#endif
<semantic-action> ::= <C-stmt>
<statement>::= <C-block>
<C-block> ::= '%{' .* '\n%}'
```

A <C-block> is any block of text that begins with '%{' and ends with '%}'. These delimiters should be on lines by themselves. The containing text is immediately passed through without modification to the C program file produced by JCS. A <C-block> usually contains helper code and data structures. Note that it will precede any C code evaluated in the context of rules, such as <C-expr> or <C-stmt>. As such, functions in a <C-block> don't need forward declarations to be visible to that code.

`<C-expr> and <C-stmt>`

A <C-expr> is a well-formed C expression or statement. It may contain calls, /**/ comments ?: expressions, balanced () {} [], and generally anything except a naked colon.

A <C-stmt> is a well-formed C statement. It is terminated by a semicolon or is a sequence of statements enclosed by '{' and '}'. A C statement may contain /**/ comments, quoted strings and newlines.

`'$$'`

The code included in <C-expr> and <C-stmt> is evaluated in the context of a specific rule. The special symbol '$$' is expanded to the name of a pointer variable which addresses the node that is the root of the subtree which this rule matches.

4.4 Other Input

In addition to the input grammar and associated actions, the programmer must provide additional definitions to allow the operation of the program produced by JCS. These should be defined in a <C-block> statement, or in a header file included by one.

4.4.1 Default Actions

If any non 0-ary rules are specified with empty synthesis or inheritance expressions, JCS will cause default actions to be called. These must be defined by the programmer:

```
DEFAULT_SYNTHESIS_CHAIN(CVMCompilationContext*, <type>)
DEFAULT_SYNTHESIS_ACTIONn(CVMCompilationContext*, <type>)
DEFAULT_INHERITANCE_CHAIN(CVMCompilationContext*, <type>)
DEFAULT_INHERITANCE_ACTIONn(CVMCompilationContext*, <type>)
```

To make it easier to write default synthesis rules, the variable

```
<type> submatch_roots[<MAX-ARITY>];
```

(for <MAX-ARITY> the maximum rule arity) contains the roots of submatches to the current rule. The synthesis action can use this to examine subtrees and their properties.

For the mechanisms available during inheritance, see the section on Synthesis, Inheritance, and Register Targeting.

4.4.2 Managing Recursion

The match, synthesis and inheritance algorithms all require traversing the IR tree recursively. In order to minimize C stack depth, we avoid simple programmatic recursion and use auxiliary data structures instead. The types of these data structures are declared as

`struct <name>_match_computation_state and`

`struct <name>_rule_computation_state`

Though the form of these structures is fixed, their allocation is not.

These macros must be provided for the match phase:

`INITIALIZE_MATCH_STACK`

No parameters. Sets up the stack of struct <name>_match_computation_state. Should declare a stack pointer.

`MATCH_PUSH(<type> this, int op, <type> l, <type> r, int n, int arity)`

Fill in a new <name>_match_computation_state record and push it on the stack. Assuming that mcp is the stack pointer declared in `INITIALIZE_MATCH_STACK`, it could be defined like this:

```
#define MATCH_PUSH( _p, op, l, r, n, arity ){ \
  mcp->p = (_p); \
  mcp->opcode = (op); \
  mcp->subtrees[0] = (l); \
  mcp->subtrees[1] = (r); \
  mcp->which_submatch = (n); \
  mcp++->n_submatch = (arity); \
}
```

`MATCH_POP( p, op, l, r, n )`

The pop operation corresponding to `MATCH_PUSH`. Must be defined as a macro since values from the top record are stored in the parameter values.

`GET_MATCH_STACK_TOP`

No parameters. Returns a (struct <name>_match_computation_state *) which is a pointer to the topmost recursion record. Used for direct access to this record.

`MATCH_STACK_EMPTY`

int valued macro. No parameters. Returns 1 if stack is empty, else 0.

These must be provided for the synthesis and action phases:

`INITIALIZE_GOAL_STACK`

No parameters. Sets up the stack of struct <name>_rule_computation_state. Does not need to declare a stack pointer, but must initialize the stack pointer

`struct <name>_rule_computation_state* goal_top`;

The goal stack will grow forward. Generated code increments goal_top to point just after the top record.

`GOAL_STACK_TOP`

No parameters. Returns a (struct <name>_rule_computation_state *) which is the limit of goal stack growth. Used to test stack integrity by calls of `validateStack()` q.v.

`GOAL_STACK_EMPTY`

int valued macro. No parameters. Returns 1 if stack is empty, else 0.

Part of the definition generated for struct <name>_rule_compuatation_state is

```
#define <name>_MAX_ARITY            <MAX-ARITY>
                <name>_attribute *   curr_attribute;
                <name>_attribute    attributes[<name>_MAX_ARITY];
```

The typedef for <name>_attribute must be provided. This is used for managing

inheritance in the presence of DAGs. See the section on Synthesis, Inheritance and Register Targeting.

4.5 Debugging

The following macros are for debugging and instrumentation. Though they must have definitions, the definitions may be empty.

The macro `validateStack` is called with a boolean expression and a word. The word is an indicator of which stack is being verified. Here are examples of its use:

```
 validateStack((goal_top < GOAL_STACK_TOP), Synthesis);

   validateStack((goal_top < GOAL_STACK_TOP), Action);
```

and here’s an example definition:

```
#define validateStack(condition, stackName) { \
 CVMassert(condition);          \
 if (!(condition)) {           \
  CVMJITerror(con, CANNOT_COMPILE, #stackName " stack too small"); \
 } \
}
```

`validateStack` can also be used by other code. For instance a call to `validateStack` could be added to the `MATCH_PUSH` macro defined above.

These macros are for collecting stack depth statistics.

`INITIALIZE_STACK_STATS(name)`

A macro that defines and initializes a counter which will be manipulated by the following routines. The name parameter is used to compose the counter name, and may be used to pass information to any further collection mechanism.

`statsPushStack(name)` indicates a push to the named stack.

`statsPopStack(name)` indicates a pop of the named stack.

The following macros or functions are for debugging the execution of semantic actions.

`CVMJITdoStartOfCodegenRuleAction(`

```
CVMJITCompilationContext* con,
```

```
int ruleno, const char* ruleString, <type> nodePtr)
```

is called immediately before executing each rule's semantic action.

`CVMJITdoEndOfCodegenRuleAction(CVMJITCompilationContext* con)` is called immediately after.

Defining `CVM_DEBUG` or `CVM_TRACE_JIT` causes additional information to be retained in case of a parse error (which should never occur in a production system).

Finally, the oldest and crudest form of debugging is to define the following:

```
#define <name>_DEBUG1
```

```
#define id(p)... get an identifying tag (e.g. node number) from
<type> p
```

will cause messages to stderr at various points in rule processing.

4.6 Output

The program produced by JCS defines three entry points. They are

```
int <name>_match( <type> root, CVMJITCompilationContext* con );
```

```
int <name>_synthesis( <type> root, CVMJITCompilationContext*
con );
```

```
int <name>_action( <type> root, CVMJITCompilationContext* con
);
```

Each phase returns 0 on success, -1 on failure.

The phases can be called independently, as long as they are called in the right order. CDC-HI uses this capability by calling the match and synthesis phases on every rooted DAG in the block before running the action phase on any of them. Since DAGs can be connected, this can allow improved register targeting across expressions.

JCS produces three output files:

- **Header file -** Contains struct definitions shared between the other two files, extern declarations for the above three entry points, and any conditional compilation macro definitions produced as a result of processing a <macro-list>.

- **Data file -** Contains constant data such as the transition tables, the table mapping state and goal into the applicable rule, and the rule prerequisit table. JCS goes to some lengths to reduce the size of these data. During state construction, JCS reduces the number of states by combining those that are indistinguishable. And when writing the output tables, identical rows and columns are coalesced using a simple mapping process. This causes table lookup to take more time, but we believe it is worth it.
- **Code file -** Starts with the output of all <C-block>s, in the order given. Helper routines should be bracketed with `#ifdef` if they are to be conditionally compiled. This is followed by the three phases of pattern matching and rules application. The declarations of these three entry points is given above.

4.7 References

- [AhoG89] Aho, Alfred V., Mahadevan Ganaparhi, and Steven W.K. Tijiang. “Code Generation Using Tree Pattern Matching and Dynamic Programming,” ACM TOPLAS, Vol. 11, No. 4, Oct. 1989, pp. 491-516.
- [Hen89] Henry, Robert R., “Encoding Optimal Pattern Selecection in a Table-Driven Bottom-Up Tree-Pattern Matcher” Technical Report #89-02-04, Computer Science Department, University of Washington, Seattle, WA.
  [Muchnick] Muchnick, Steven S., Advanced Compiler Design and Implementation, Morgan Kauffman Publishers, San Francisco, CA, 1997.
- [Pel88] Pelegri-Llopart, Eduardo “Rewrite Systems, Pattern Matching, and Code Generation” Technical Report UCB/CSD 88/423, Department of Electrical Engineering and Computer Sciences, UC Berkeley.

CHAPTER 5

Code Generation Mechanics

A few points on the mechanics of code generation are introduced here:

- Code generation for DAGs in the IR
- The code generator's semantic stack and its relation to JCS rules
- Synthesis, inheritance and register targeting
- Simple rule overriding using rule costs

5.1 Working with DAGs in Code Generation

5.1.1 Introduction

Although DAGs are a special case in the CDC-HI IR, their handling must be fully understood for correct code generation. Here we discuss three aspects of DAGs: their properties, the mechanism of traversal using JCS, and their annotations in the semantic actions of code generation.

5.1.2 Properties and semantics Of DAGs in the CDC-HI IR

- The only nodes which can have multiple parents are the IDENTITY operator the CONSTANT node.
- All IR nodes have a refCount field. Only the IDENTITY and CONSTANT nodes can have a refCount > 1.

- Java has very strict left-to-right semantics, which the code generator must obey. A DAG must be fully evaluated when it is first encountered using a first-to-last traversal of a block's roots, depth-first left-to-right traversal of each rooted DAG. Thereafter it must not be reevaluated. This obviously doesn't matter for CONSTANT nodes, but it does for IDENTITY operators.
- Due to the way the IR is generated, an IDENTITY node must be no larger than the smallest node. If you don't modify the node type definitions this should not be a concern.

5.1.3 JCS handling of DAGs

When a DAG rule is present in a JCS grammar, special code is generated for the pattern matching phases. In addition to the state numbers attached to every node using the`%setstate` function, the code stores a state flag which tells it whether a node has been visited the match phase, the synthesis phase, or the action phase. Once a node has been Matched, it will not be traversed and marked again. Code like this is generated at the top of the match loop:

```
stateno = IRGetState(p);
if (stateno >= CVMJIT_JCS_STATE_MATCHED){
 goto skip_match;
}
```

As a side note, the field we use to keep state is the same one that the front end uses to keep subtree-size count

```
CVMUint16curRootCnt;
```

At the end of front-end processing of a method, this is summarized and used to estimate the size requirements of some of the back end's compile-time structures, so this field is available for back end use. However, we don't bother to rezero it before entering the back end, counting on the small size of trees to be less than the int value of this flag. If we ever did wish to reparse the block, this field would have to be zeroed to clear all the state bits.

Similarly, the synthesis phase flags those nodes it visits. If it comes across a node which has already been visited by synthesis and the rule being matched is marked as a `%dag` rule, then synthesis processing of this node and any subtree matches is cut off.

Finally, the action phase acts similarly to the synthesis phase, cutting off recursive processing when a flagged node rooting a DAG rule is found. The semantic action part of the rule is still executed. The semantic action is responsible for determining whether or not it has been executed before, and determining what to do on reentry. The macro `CVMJIT_DID_SEMANTIC_ACTION` is available to help it do this. Here's an example:

```
%dag reg32: IDENT32 reg32 : 0 : : : : {
                 /* COMMON DECLARATIONS AND ACTIONS */
                 ...
                 if (!CVMJIT_DID_SEMANTIC_ACTION($$)){
                   /* FIRST VISIT ACTIONS */
                   ...
                 } else {
                   /* SUBSEQUENT VISIT ACTIONS */
                   ...
                 }
                 /* COMMON ACTIONS */
                 ...
               };
```

5.1.4 Decoration of IDENTITY Nodes

In addition to all the fields shared with the Unary node type, the IDENTITY node has one unique field, `identDecl`, which is a pointer to `CVMJITIdentityDecoration`, declared as follows:

```
typedef struct {
#ifdef CVM_DEBUG_ASSERTS
               CVMJITIdentityDecorationType decorationTag;
#endif
               CVMInt32  refCount;
} CVMJITIdentityDecoration;
```

This is an example of faking a hierarchical type system using C: the `identDecl` field must point to a struct the first element of which is a `CVMJITIdentityDecoration`. We should know by context – which rule we are processing -- what the actual type is, but when debugging we also have the decorationTag to test for correctness.

The simplest and most common subtype of `CVMJITIdentityDecoration` is the `CVMRMResource`. This is an object representing a value either in a register or in a temporary memory location. These are managed by calls to the register manager, which allocates registers and manages their contents by generating load and store instructions. The refCount field is used to determine when a value still has future uses and indicate that it should be retained. When the semantic action of a rule producing a `'reg32'` is executed, a pointer to a `CVMRMResource` is pushed on a compile-time semantic stack for consumption by later rules. With this background, we can flesh out the above example:

```
  %dag reg32: IDENT32 reg32 : 0 : : : : {
                   CVMRMResource* src;
                   if (!CVMJIT_DID_SEMANTIC_ACTION($$)){
                     /* FIRST EVALUATION */
                     /* get subtree's CVMRMresource from stack */
                     src = popResource(con);
                     /*
                      * point $$->identDecl to src using
                      * CVMJITidentitySetDecoration(con, src, $$);
                      *  this should set src->dec.refCount to $$-
>refCount using
                      *  CVMJITidentitySetDecorationRefCount(con,
&(src->dec),
                      *              CVMJITirnodeGetRefCount($$));
                      * Do other management tasks, too.
                      */

CVMRMoccupyAndUnpinResource(CVMRM_INT_REGS(con), src, $$);
                   } else {
                     /* SUBSEQUENT EVALUATIONS */
                     /*
                      * fetch $$->identDecl using
                      * CVMJITidentityGetDecoration(con, $$);
                      */
                     src = CVMRMfindResource(CVMRM_INT_REGS(con),
$$);
                     if ( src == NULL ){
                       /* something went unexpectedly wrong!! */
                       return JIT_RESOURCE_NONE_ERROR;
                     }
                   }
                  /* Our parent rule expects to find a CVMRMResource*
on the
       stack */
                   pushResource(con, src);
                 };
```

The only other type of IDENTITY node decoration we use is the ScaledIndexInfo. This is a structure that bundles up the information needed to use the commonly-available array addressing modes:

```
[basereg+const]
```

```
[basereg+(indexreg*stride)] for stride in {1,2,4,8}
```

These are more intricate than CVMRMResources because of the number of elements and choices that can be involved, but they are managed in the same manner.

5.2 Code Generator's Semantic Stack

The semantic stack is used during code generation to communicate the product of a subtree rule match to its parent match. The products of subtree matches become input for their parents. Tree rule semantic actions are executed in depth-first, left-to-right order. Thus a stack discipline can be used to manage rule products. When there are multiple subrules, the left-most is deepest, and the right-most is on the stack top. Each rule pops from the stack any products of subtree matches, does its own processing, and pushes on the stack its own product.

The code generator's semantic stack does not mimic the Java interpreter's evaluation stack.

Here is an example which assumes tree-formed expressions. This example shows how the non-terminals of a JCS grammar define a simple type system, and how that type system is reflected in the semantic stack.

```
%name  SimpleExample
%leaf  ICONST_32
%leaf  LOCAL32
%binary INDEX
%binary IADD32
%binary IDIV32
%binary ASSIGN
%goal  statement
// nonterminals are: statement, reg32, aluRhs
statement: ASSIGN INDEX reg32 reg32 reg32 : 1 : : :
                                  semantic_action;
statement: ASSIGN LOCAL reg32 : 1 : : : semantic_action;
reg32: ICONST_32 : 1 : : : semantic_action;
```

```
  reg32: LOCAL32 : 1 : : : semantic_action;
  reg32: IADD32 reg32 aluRhs : 1 : : : semantic_action;
  reg32: IDIV32 reg32 aluRhs : 4 : : : semantic_action;
  aluRhs: ICONST_32 : 0 : : : semantic_action;
  aluRhs: reg32 : 0 : : : semantic_action;
```

Note the forms of `aluRhs` here: a constant or a computed value. These correspond to operand modes found in some instruction sets.

In the examples below, a `CVMRMresource` is the the data structure used by the register manager (q.v.) to model the location of a value. The value can be in a register, in a temporary location in memory, or a constant value to be instantiated on demand.

Assume that each rule producing a reg32 computes this value into a temporary cell, as represented by a `CVMRMResource`. An appropriate semantic stack element type would look something like this:

```
typedef enum {
  REG_ELEMENT, ALU_RHS_ELEMENT;
} StackElementType;
typedef enum {
  CONST_RHS, VALUE_RHS
} AluRhsType;
typedef struct {
  AluRhsType                    rhsTag;
  ConstVal                      rhsConstVal;
  CVMRMResource *               rhsRegVal;
} AluRhs;
typedef struct {
  StackElementType              tag;
  union {
                     CVMRMResource *val;
                     AluRhs     rhsVal;
  }u;
} SemanticStackElement;
```

Here, the `StackElementType` tag is only for debugging: when a reg32 nonterminal is part of a pattern, the stack element must be a `REG_ELEMENT`; when an `aluRhs` nonterminal is part of a pattern, the stack element must be an `ALU_RHS_ELEMENT`.

But the `AluRhsType` tag is required here, since the distinctions among the two types of `aluRhs` nonterminals has been lost by the grammar, but must be rediscovered when generating code.

Note that for a chain rule, you often have to pop an element from the stack, manipulate the data, and re-push it in another form. The rule `aluRhs: reg32` is an example of a chain rule.

We can now fill in the example informally:

```
%{
  /*
   * In the production system, this is dynamically allocated
   * as part of the CVMJITCompilationContext.
   */
  static SemanticStackElement sstack[MAX_SS_DEPTH];
  static SemanticStackElement * sptr = &sstack[0];
  void
  pushResource(CVMJITCompilationContext* con, CVMRMResource* v){
                sptr->tag = REG_ELEMENT;
                sptr->u.val = v;
                sptr++;
  }
  CVMRMResource *
  popResource(CVMJITCompilationContext* con){
                sptr--;
                assert(sptr->tag == REG_ELEMENT);
                return sptr->u.val;
  }
  void
  pushAluRhs(
                CVMJITCompilationContext* con,
                AluRhsType tag,
                ConstVal constant,
                CVMRMResource* val
  ){
                sptr->tag = ALU_RHS_ELEMENT;
                sptr->u.rhsVal.rhsTag = tag;
                sptr->u.rhsVal.rhsConstVal = constant;
```

```
                    sptr->u.rhsVal.rhsRegVal = val;
                    sptr++;
    }
    void
    popAluRhs(CVMJITCompilationContext* con, AluRhs* val){
                    sptr--;
                    assert(sptr->tag == ALU_RHS_ELEMENT);
                    val->rhsTag    = sptr->u.rhsVal.rhsTag;
                    val->rhsConstVal = sptr->u.rhsVal.rhsConstVal;
                    val->rhsRegVal  = sptr->u.rhsVal.rhsRegVal;
    }
  %}
    statement: ASSIGN INDEX reg32 reg32 reg32 : 1 : : : {
                      CVMRMResource* valToStore;
                      CVMRMResource* baseReg;
                      CVMRMResource* indexReg;
                      valToStore = popResource(con);
                      indexReg = popResource(con);
                      baseReg = popResource(con);
                      generateIndexedAssignment(baseReg, indexReg,
  valToStore);
                    }
    statement: ASSIGN LOCAL32 reg32 : 1 : : : {
                      CVMRMResource* val;
                      LocalInfo* l;
                      l = extractLocalInfo($$->left);
                      val = popResource(con);
                      generateLocalAssignment(l, val);
                    }
    reg32: ICONST_32 : 1 : : : {
                      CVMRMResource* val;
                      ConstantValue conval;
                      conval = extractConstValue($$);
                      val  = getResource();
                      generateLoadConstantValue(val, conval);
                      pushResource(con, val);
```

```
                 }
  reg32: LOCAL32 : 1 : : : {
                   CVMRMResource* val;
                   LocalInfo* l;
                   l = extractLocalInfo($$);
                   val = getResource();
                   generateLoadLocal(val, l);
                   pushResource(con, val);
                 }
  reg32: IADD32 reg32 aluRhs : 1 : : : {
                   AluRhs     rhsVal;
                   CVMRMResource* lhsVal;
                   CVMRMResource* resultVal = getResource();
                   popAluRhs(con, &rhsVal);
                   lhsVal = popResource(con);
                   generateALUInstruction(OPCODE_ADD, resultVal,
lhsVal,
                                  &rhsVal);
                   pushResource(con, resultVal);
                 }
  reg32: IDIV32 reg32 aluRhs : 4 : : : {
                   AluRhs     rhsVal;
                   CVMRMResource* lhsVal;
                   CVMRMResource* resultVal = getResource();
                   popAluRhs(con, &rhsVal);
                   lhsVal = popResource(con);
                   generateALUInstruction(OPCODE_IDIV, resultVal,
lhsVal,
                                  &rhsVal);
                   pushResource(con, resultVal);
                 }
  aluRhs: ICONST_32 : 0 : : : {
                   ConstantValue conval = extractConstValue($$);
                   pushAluRhs(con, CONST_RHS, conval, NULL);
                 }
  aluRhs: reg32 : 0 : : : {
                   CVMRMResource* val = popResource(con);
                   pushAluRhs(con, VALUE_RHS, 0, val);
                 }
```

5.3 Synthesis, Inheritance, and Register Targeting

Here we will walk through an example of using JCS's synthesis and inheritance phases to implement simple register targeting. For the sake of simplicity, this example will assume tree formed expressions without DAGs. In the next section we'll revisit the example, showing how to target using the subtree evaluation target stack managed by JCS.

This example is a continuation of the above example on the semantic stack. The main difference is that in this case, we assume that integer division is implemented

by a subroutine call, that its parameters and return value are in registers, and that the subroutine will use a larger set of registers without saving them. Let

`Rl`	-- set containing the target register for left operator of division
`Rr`	-- set containing the target register for right operator of division
`Rd`	-- set containing the register containing the result of a division
`RdivSet`	-- set of registers used by division. Assumed to superset Rl + Rr + Rd.
`RanySet`	-- set of usable registers
`RbaseSet`	-- set of registers usable for base of indexed addressing.
`Rempty`	-- empty register set

`SetAnnotation(node, which, value)` -- annotate (decorate) the indicated node. 'which' indicates which annotation to set. For this example, TARGET or USED 'value' is the set value to use for the annotation.

- **USED -** the attribute naming the set of registers which must be used by this tree and its subtrees. For example, if there's a divide in here, it will include RdivSet.
- **TARGET -** the attribute naming the set of registers acceptable for the result of this subtree's evaluation.
- `GetAnnotationValue(node, which)` **-** retrieve the value of the indicated annotation.

In this example, a node’s USED attribute is synthesized from those of its children plus information about the node operation itself. The TARGET attribute is inherited from its parent.

No default macros are used in these examples. In an actual JCS grammar there would certainly be enough redundance that you would want to do so.

Ultimately, it is up to the consumer of a value to ensure that the value is in the required register, but if a subrule match can cause a result to be delivered in the proper register in the first place, excessive shuffling will be avoided.

```
  // target the base value to be in an acceptable base register
  // if possible
  statement: ASSIGN INDEX reg32 reg32 reg32 : 1 :
                 : SetAnnotation($$->left->left, TARGET,
                     RbaseSet - GetAnnotation($$->right, USED)
                     - GetAnnotation($$->left->right,USED));
                  SetAnnotation($$->left->right, TARGET,
                     RanySet - GetAnnotation($$->right, USED));
                  SetAnnotation($$->right, TARGET, RanySet)
                 : {
                   CVMRMResource* valToStore;
                   CVMRMResource* baseReg;
                   CVMRMResource* indexReg;
                   AluRhs target;
                   valToStore = popResource(con);
                   indexReg = popResource(con);
                   baseReg = popResource(con);
                   ensureInReg(baseReg, RbaseSet);
                   ensureInReg(valToStore, RanySet);
                   ensureInReg(indexReg, RanySet)
                   generateIndexedAssignment(baseReg, indexReg,
valToStore);
                 }
  statement: ASSIGN LOCAL32 reg32 : 1 :
                 : SetAnnotation($$->right, TARGET, RanySet)
   : {
                   CVMRMResource* val;
                   LocalInfo* l;
                   l = extractLocalInfo($$->left);
                   val = popResource(con);
                   ensureInReg(valToStore, RanySet);
                   generateLocalAssignment(l, val);
                 }
reg32: ICONST_32 : 1 : SetAnnotation($$,USED,Rempty) : : {
                   CVMRMResource* val;
                   ConstantValue conval;
                   conval = extractConstValue($$);
                   val  = getResource(GetAnnotation($$,TARGET));
                   generateLoadConstantValue(val, conval);
                   pushResource(con, val);
                 }
reg32: LOCAL32 : 1 : SetAnnotation($$,USED,Rempty) : : {
                   CVMRMResource* val;
                   LocalInfo* l;
                   l  = extractLocalInfo($$);
                   val = getResource(GetAnnotation($$,TARGET));
                   generateLoadLocal(val, l);
                   pushResource(con, val);
                 }
//
// The set of registers used by this tree is the union of the
// sets of its subtrees, since this node has no particular
// requirements. Since the left-hand subtree is evaluated first,
// it would be wise to target it to any that the
// right-hand subtree does not require.
reg32: IADD32 reg32 aluRhs : 1 :
                  SetAnnotation($$, USED,
                     GetAnnotation($$->left,USED)
                     + GetAnnotation($$->right,USED))
                 :
                  SetAnnotation($$->left, TARGET,
                     RanySet-GetAnnotation($$->right,USED));
                  SetAnnotation($$->right, TARGET, RanySet)
                 : {
                   AluRhs rhsVal;
```

```
                        CVMRMResource* lhsVal;
                        CVMRMResource* resultVal;
                        popAluRhs(con, &rhsVal);
                        lhsVal = popResource(con);
                        resultVal =
getResource(GetAnnotation($$,TARGET));
                        ensureInAluRhs(&rhsVal, RanySet);
                        ensureInReg(lhsVal, RanySet);
                        generateALUInstruction(OPCODE_ADD, resultVal,
lhsVal,
                                       &rhsVal);
                        pushResource(con, resultVal);
                      }
  //
  // This operation requires that the value produced by the left
  // subtree be placed in register Rl,
  // and the value produced by the right subtree
  // be placed in register Rr. This rule uses a large set of registers
  // as a side effect. It produces a value in register Rd.
  reg32: IDIV32 reg32 reg32 : 10 :
                       SetAnnotation($$, USED, RdivSet
                          + GetAnnotation($$->left,USED)
                          + GetAnnotation($$->right,USED));
                      : SetAnnotation($$->left,TARGET,Rl);
                       SetAnnotation($$->right,TARGET,Rr)
                      : {
                        CVMRMResource* rhsVal = popResource(con);
                        CVMRMResource* lhsVal = popResource(con);
                        CVMRMResource* resultVal;
                        ensureInReg(lhsVal,Rl);
                        ensureInReg(rhsVal,Rr);
                        flushRegs(RdivSet-Rl-Rr);
                        generateCall("intDiv");
                        resultVal = getResource(Rd);
                        pushResource(resultVal);
                      }
```

```
aluRhs: ICONST_32 : 0
 : SetAnnotation($$, USED, Rempty)
               : {
                 ConstantValue conval = extractConstValue($$);
                 pushAluRhs(con, CONST_RHS, conval, NULL);
               }
aluRhs: reg32 : 0
               : /* chain rule: pass through USED from below*/ ;
               : /*chain rule: pass through TARGET from above */ ;
               : {
                 CVMRMResource* val = popResource(con);
                 pushAluRhs(con, VALUE_RHS, 0, val);
               }
```

5.3.1 Targeting with DAGs

In the presence of DAGs, the simple strategy given above doesn't quite work. Although the synthesized attributes propagate correctly, there is a problem with the inherited attributes. Consider an IDENTITY node with multiple parents. Each parent will hand down to it the target attribute desired for the evaluation of that parent. Because of the first-to-last left-to-right evaluation order, that means that the IDENTITY node will be left with the inherited attributes of the last parent that will be evaluated. But it almost always makes more sense for it to have the attributes of the first one when it is first evaluated. For this reason, inherited attributes can be kept in the stack used by the Action phase to manage recursion. The inherited data are encapsulated in the structure <name>_attributes mentioned in the section on JCS.

Inherited attributes for the current node are accessed in the body of a semantic action using

```
(goal_top-1)->curr_attributes.
```

In the body of an inheritance action the attribute inherited by the current node from the parent currently being evaluated is also accessed using

```
(goal_top-1)->curr_attributes
```

while the value we wish to pass to our children -- left-to-right – are

```
goal_top->attributes[0] ... goal_top->attributes[n-1]
```

Note that this also makes it easier to write default rules for any n-ary rules without knowing the exact topography of them.

5.3.2 Targeting for Avoidance

The above example contained instances of targeting the evaluation of an expression into a specific register or set of registers, and of targeting the evaluation of an expression to avoid a specific set of registers. Consider this example IR tree:

```
                       ASSIGN(8)
                      /    \
       /     \
       /      \
                   INDEX      IDIV32(7)
    /   \      /    \
    /    \     /     \
LOCAL32(1) IADD32(4) LOCAL32(5) LOCAL32(6)
                    /   \
     /    \
                LOCAL32(2)  LOCAL32(3)
```

In order to obey strict left-to-right evaluation semantics, we must evaluate all the components of the ASSIGN's left hand subtree before evaluating its right hand subtree. Conceptually, they will be evaluated in the order indicated by the labels. (The INDEX node is not labeled because it is not the root of a rule submatch.)

Here is the targeting information for each node. In this table we break out the positive targeting information (target set) from the negative targeting information (avoid set).

node	target set	avoid set
1	RbaseSet	RdivSet
2	RanySet	RdivSet
3	RanySet	RdivSet
4	RanySet	RdivSet
5	Rl	Rempty
6	Rr	Rempty

7	RanySet	Rempty
8	n/a	n/a

In the CDC-HI compiler, the inherited register targeting attributes are represented as pairs of register sets: the target set and the avoid set. These provide more information to the register manager, which can weigh the following factors in a register allocation decision:

- Allocation request is strict or non-strict;
- Which register, if any, the resource is currently in;
- The values of the target set and the avoid set;
- Whether any registers are available in the requested set.

5.4 Using and Reference Counting Resources

There are a few rules that must be followed when working with CVMRMResources.

- Whenever an instruction is generated, all value cells referenced by that instruction must be bound to addresses. In general, this means bound (or as we say ’pinned’) to registers. This includes values consumed by an instruction as well as any values to be produced.
- `CVMRMResources` should not be bound any longer than required, as this makes the Register Manager’s job harder. In general, we don’t keep CVMRMResources pinned between code generation rules.
- `CVMRMResources` have reference counts, and these must be maintained as code is generated.
  - A resource gets its initial reference count from the rule that first produces the value. This count is taken from the IR node which is the root of the rule submatch. IDENTITY and CONSTANT nodes can have a reference count greater than 1.
  - Each time a rule consumes a value, the reference count must be decremented by 1. Note that there isn’t necessarily a 1-to-1 correspondence between number of instructions referencing a value and the number of conceptual uses of that value: one example would be a special rule for multiplying an integer variable by a small constant, which could be done by shifting and adding or subtracting. This may result in several instructions referencing the variable, but would still count as only one use. Generally, there is one use of a value for each matched rule in which it appears in the pattern. An obvious exception to this generality is a chain rule: this isn’t actually a use of the value but simply a renaming, so must not be counted.

- When a CVMRMResource's reference count reaches 0, the register manager is free to re-use the registers and temporary memory location occupied by it without saving the value for any further uses.
- So long as the reference count on a CVMRMResource remains greater than 0, the register manager will preserve its value for a future reference.

5.5 Full Example of Inheritance and Resouce Management

It is now possible to complete the above example with all the necessary targeting and resource management code. All references to a node's TARGET attribute have been replaced with the goal_top-relative attribute handling, and all references to a node's USED attribute have been replaced by references to its regsRequired field. An IDENTITY rule has been added to illustrate DAG handling.

This example is still a little simpler than the actual CVM-HI compiler. The example causes constants to be evaluated into a register immediately. In the full CVM-HI compiler, we use CVMRMbindResourceForConstant32 to create a resource for a constant without allocating a register for it at this time. This allows us to delay actual allocation until the call to CVMRMpinRegister.

```
%{
/*
 * Make SimpleExample_attribute a pair of sets.
 * the node annotations we've been using.
 */
typedef struct {
  SET target;
  SET avoid;
} SimpleExample_attribute;
/*
 * Easy way to get to the target of the current rule.
 * Result is a pair of SETs.
 */
#define GET_REGISTER_GOALS \
                 (goal_top-1)->curr_attributes.target, \
                 (goal_top-1)->curr_attributes.avoid
/*
```

```
 * Easy way to pass our own goals to one of our children
 */
#define PASS_GOALS_TO_CHILD(n) \
                 goal_top->attributes[n].target = (goal_top-1)-
>attributes.target;\
                 goal_top->attributes[n].avoid = (goal_top-1)-
>attributes.avoid
/*
 * Many rules have at least one child with relaxed targeting.
 */
#define PASS_ANY_GOALS_TO_CHILD(n) \
                 goal_top->attributes[n].target = RanySet;\
                 goal_top->attributes[n].avoid = Rempty
/*
 * pin and relinquish the reg32 underlying an AluRhs if indeed
 * there is one
 */
static void
pinAluRhs(..., AluRhs *rhsVal, SET targetSet, SET avoidSet){
  if (rhsVal->rhsTag == VALUE_RHS){
                 CVMRMpinResource(..., rhsVal->rhsRegVal, targetSet,
avoidSet);
  }
}
static void
relinquishAluRhs(..., AluRhs *rhsVal){
  if (rhsVal->rhsTag == VALUE_RHS){
                 /* to relinquish a resource is to both unpin it and
to
                  * decrement the reference count.
                  */
                 CVMRMrelinquishResource(..., rhsVal->rhsRegVal);
  }
}
%}
  %dag reg32: IDENTITY32 reg32 : 0
```

```
                 : $$->regsRequired = submatch_roots[0]-
>regsRequired
                 : PASS_GOALS_TO_CHILD(0)
                 : {
                   CVMRMResource* src;
                   if (!CVMJIT_DID_SEMANTIC_ACTION($$)){
                     /* FIRST EVALUATION */
                     /* get subtree's CVMRMresource from stack */
                     src = popResource(con);
                     /* associate this resource with the IDENTITY
node
                      * and give the resource the node's reference
count.
                      */
                     CVMRMoccupyAndUnpinResource(..., src, $$);
                   } else {
                     /* SUBSEQUENT EVALUATIONS */
                     /*
                      * find the resource containing the value.
                      */
                     src = CVMRMfindResource(..., $$);
                   }
                   /* Our parent rule expects to find a
                    * CVMRMResource* on the stack
                    */
                   pushResource(con, src);
                 };
  // target the base register to be in an acceptable base register
  // if possible. Target base register to avoid anything required
  // by the index or the RHS. Target the index register to avoid
  // the requirements of the RHS.
  statement: ASSIGN INDEX reg32 reg32 reg32 : 1 :
                 : goal_top->attributes[0].target = RbaseSet;
                  goal_top->attributes[0].avoid = $$->right-
>regsRequired
                              + $$->left->right->regsRequired;
                  goal_top->attributes[1].target = RanySet;
                  goal_top->attributes[1].avoid = $$->right-
>regsRequired;
                  PASS_ANY_GOALS_TO_CHILD(2)
                 : {
                   CVMRMResource* valToStore;
                   CVMRMResource* baseReg;
                   CVMRMResource* indexReg;
                   valToStore = popResource(con);
                   indexReg = popResource(con);
                   baseReg = popResource(con);
                   CVMRMpinResource(..., baseReg, RbaseSet, Rempty);
                   CVMRMpinResource(..., valToStore, RanySet,
Rempty);
                   CVMRMpinResource(..., indexReg, RanySet, Rempty)
                   generateIndexedAssignment(baseReg, indexReg,
valToStore);
                   /* to relinquish a resource is to both unpin it
and to
                    * decrement the reference count.
                    */
                   CVMRMrelinquishResource(..., baseReg);
                   CVMRMrelinquishResource(..., valToStore);
                   CVMRMrelinquishResource(..., indexReg);
                 }
  statement: ASSIGN LOCAL32 reg32 : 1 :
                 : PASS_ANY_GOALS_TO_CHILD(0)
                 : {
                   CVMRMResource* val;
                   LocalInfo* l;
                   l = extractLocalInfo($$->left);
                   val = popResource(con);
                   CVMRMpinResource(..., valToStore, RanySet,
Rempty);
                   generateLocalAssignment(l, val);
                   CVMRMrelinquishResource(..., valToStore);
                 }
  reg32: ICONST_32 : 1 : $$->regsRequired = Rempty : : {
                   CVMRMResource* val;
                   ConstantValue conval;
                   conval = extractConstValue($$);
                   val  = CVMRMgetResource(..., GET_REGISTER_GOALS,
1);
                   generateLoadConstantValue(val, conval);
                   CVMRMUnpinResource(...,val);
                   pushResource(con, val);
                 }
  reg32: LOCAL32 : 1 : $$->regsRequired = Rempty : : {
                   CVMRMResource* val;
                   LocalInfo* l;
                   l  = extractLocalInfo($$);
                   val = CVMRMgetResource(..., GET_REGISTER_GOALS,
1);
                   generateLoadLocal(val, l);
                   CVMRMUnpinResource(...,val);
                   pushResource(con, val);
                 }
  //
  // The set of registers used by this tree is the union of the
  // sets of its subtrees, since this node has no particular
  // requirements.Since the left-hand subtree is evaluated first,
  // it would be wise to target it to any that the right-hand
  // subtree does not require.
  reg32: IADD32 reg32 aluRhs : 1 :
                  $$->regsRequired = $$->left->regsRequired
                             + $$->right->regsRequired
                 : goal_top->attributes[0].target = RanySet;
    goal_top->attributes[0].avoid = $$->right->regsRequired;
                  PASS_ANY_GOALS_TO_CHILD(1)
                 : {
                   AluRhs rhsVal;
                   CVMRMResource* lhsVal;
                   CVMRMResource* resultVal;
                   popAluRhs(con, &rhsVal);
                   lhsVal = popResource(con);
                   resultVal = CVMRMgetResource(...,
GET_REGISTER_GOALS, 1);
                   pinAluRhs(..., &rhsVal, RanySet, Rempty);
                   CVMRMpinResource(..., lhsVal, RanySet, Rempty);
                   generateALUInstruction(OPCODE_ADD, resultVal,
lhsVal,
                                  &rhsVal);
                   CVMRMunpinResource(...,resultVal);
                   relinquishAluRhs(..., &rhsVal);
                   CVMRMrelinquishResource(..., lhsVal);
                   pushResource(con, resultVal);
                 }
  //
  // This operation requires that the value produced by the left
  // subtree be placed in register Rl,
  // and the value produced by the right subtree
  // be placed in register Rr. This rule uses a large set of registers
  // as a side effect. It produces a value in register Rd.
  //
  reg32: IDIV32 reg32 reg32 : 10 :
                  $$->regsRequired = RdivSet
                     + $$->left->regsRequired + $$->right-
>regsRequired;
                 : goal_top->attributes[0].target = Rl;
                  goal_top->attributes[1].target = Rr;
                 : {
                   CVMRMResource* rhsVal = popResource(con);
                   CVMRMResource* lhsVal = popResource(con);
                   CVMRMResource* resultVal;
                   lhsVal = CVMRMpinResourceSpecific(..., lhsVal,
Rl);
                   rhsVal = CVMRMpinResourceSpecific(..., rhsVal,
Rr);
                   flushRegs(RdivSet-Rl-Rr);
                   generateCall("intDiv");
```

```
                        CVMRMrelinquishResource(..., lhsVal);
                        CVMRMrelinquishResource(..., rhsVal);
                        resultVal = CVMRMgetResourceSpecific(..., Rd, 1);
                        CVMRMunpinResource(..., resultVal);
                        pushResource(resultVal);
                      }
aluRhs: ICONST_32 : 0
 : $$->regsRequired = Rempty
                      : {
                        ConstantValue conval = extractConstValue($$);
                        pushAluRhs(con, CONST_RHS, conval, NULL);
                      }
aluRhs: reg32 : 0
                      : /* chain rule: pass through USED from below */ ;
                      : /*chain rule: pass through TARGET from above */ ;
                      : {
                        CVMRMResource* val = popResource(con);
                        pushAluRhs(con, VALUE_RHS, 0, val);
                      }
```

5.6 Rule Costs and Overriding

Because JCS uses costs in rule selection, it is possible to write a rule that effectively overrides a similar rule of higher cost. The procedure for simple rule overriding is very simple: write a new rule with the same result and pattern body as the rule you wish to replace, but give it a lower cost. It is possible to override all uses of a rule, or to write more specialized rules that use other, cheaper, instructions in some circumstances.

For example, the default rules supplied with the CVM-HI RISC JIT assume that 32-bit integer division is done by calling a helper function. The rule looks looks like this:

(a) `reg32: IDIV32 reg32 reg32 : 90 : ...`

Note the very high cost of 90.

As an example of a simple yet complete rule override, a target instruction set having a divide instruction might choose to add a rule looking like this:

(b) `reg32: IDIV32 reg32 reg32 : 40 : ...`

Because 40 < 90, this rule will completely replace all uses of the above rule.

The CVM-HI RISC JIT also has a rule for integer division by a constant. It does shifting or multiplication by a reciprocal. Its rule looks like this:

(c) `reg32: IDIV32 reg32 ICONST_32 : 40 : ...`

Because of the lower cost, 40, this rule is used for integer division by a constant when only the default rule (a) above is present, since 40 < 90.

But look at the interaction of rule (c) and the example overriding rule (b). To use rule (b), both operands would have to be the result of reg32-producing rules. To turn an ICONST_32 into a reg32 requires executing this rule, which has cost 20:

(d) `reg32: ICONST_32 : 20 : : : : ...`

So the overall cost of dividing a reg32 by a constant using rule (c) would actually be the sum of the costs of rules (c) and (d), which is 60. Since the cost of rule (b), 40, is less than this, it will still be used in preference to using rule (d) followed by rule (c).

CHAPTER 6

Code Cache Manager

Compiled methods are stored in one large allocated piece of memory called the code cache. The code cache is allocated during VM startup and, once allocated, it cannot grow or shrink. The default size is 512k, but this can be modified on the command line using the `-Xjit:codeCacheSize` option. The start of the code cache is pointed to by `CVMglobals.jit.codeCacheStart` and the end is pointed to by `CVMglobals.jit.codeCacheEnd`.

6.1 Code Buffers

When a method is compiled, a code buffer is allocated for it from the code cache, and when a method is decompiled, its code buffer is returned to the code cache. A buffer in the code cache has the following layout, whether allocated or free:

```
CVMUint32 size;

<variable length data, such as compiled code>

CVMUint32 size;
```

The first size field is used to locate the next buffer in the code cache. The second size field is used to locate the start of the previous buffer. Thus the 4 bytes located before any code buffer will be the size field of the previous buffer. This is mainly used when coalescing free blocks so the start of the pervious buffer can be located.

The size fields also can have one of the following bits set to indicate what the code buffer is used for:

```
#define CVMJIT_CBUFSIZE_FREE_BITS     0x1 /* free buffer */

#define CVMJIT_CBUFSIZE_UNCOMMITTED_BITS 0x2 /* compiling into
*/
```

```
#define CVMJIT_CBUFSIZE_CCMCOPY_BITS    0x3 /* copied CCM code
*/
```

If no bits are set then the code buffer contains a compiled method.

6.2 Free Buffers

Free buffers are maintained in a link list pointed to by `CVMglobals.jit.codeCacheFirstFreeBuf`. The following struct is used for the list:

```
struct CVMJITFreeBuf {
    CVMUint32   size;
    CVMJITFreeBuf* next;
    CVMJITFreeBuf* prev;
};
```

Note that the size field is first, as is required by all buffers in the code cache. There is also size field at the end of the free buffer that is not shown in the struct.

When a buffer is freed, it coalesced with the previous and next buffers if they are also free, and is then added to the start of `CVMglobals.jit.codeCacheFirstFreeBuf`.

`CVMglobals.jit.codeCacheLargestFreeBuffer` is used to keep track of the largest free buffer, so it is not necessary to walk the free list if there isn't a buffer of sufficient size, and also to determine if forced decompilation produced a large enough free buffer.

6.3 Allocated Buffers for CDC 1.0.1

Allocated buffers in CDC 1.0.1 have the following layout:

```
CVMCompiledMethodDescriptor

<generated code>

CVMUint32 size;
```

The first field of the `CVMCompiledMethodDescriptor` is a size field, so the first field of an allocated buffer is the size field as is required for all code buffers.

CVMJITcbufAllocate() is called to allocate a code buffer for compilation. An estimate for the code buffer size is made based mostly on the size of the IR. CVMJITcbufAllocate() will look for the first free block that meets the estimate requirement (by walking CVMglobals.jit.codeCacheFirstFreeBuf).

Initially there is just one giant free buffer, and CVMJITcbufAllocate() will return this entire buffer (since parallel compilations aren't allowed). When compilation is complete, CVMJITcbufCommit() is called to commit the code buffer. If any extra memory is available after the end of code generated into the buffer, it is split off into a new code buffer that is freed. Thus until any decompilation occurs, there will always be just one free code buffer, and new compiled methods are placed at the start of it.

If there has been decompilation and the code cache is fragmented, then CVMJITcbufAllocate() simply returns the first free buffer from the CVMglobals.jit.codeCacheFirstFreeBuf list that is large enough to meet the estimate. Any extra is split into a new free block when CVMJITcbufCommit() is called. If the estimate is too small, then the entire buffer is freed, and recompilation is done with a larger block.

During code generation, CVMJITCompilationContext.codeBufAddr will point to the start of the code buffer and codeBufEnd will point to the last word that code can be generated into. codeBufEnd is not the end of the code buffer, since CVMJITcbufAllocate() always reserves 4 bytes for the trailing size field, and will also limit codeCacheEnd so generated code does not exceed the maximum allowed by the -Xjit:maxCompiledMethodSize option.

CVMJITCompilationContext.codeEntry points to the first word of generated code, which occurs just after the CVMCompiledMethodDescriptor struct at the start of the code cache.

6.4 Allocated Buffers for CDC 1.1

For CDC 1.1, the format of an allocated buffer has changed quite a bit. The reason for the change was to move into the code buffer many supporting data structures that previously were allocated using malloc(). This served four purposes:

- Reduced fragmentation in the malloc() heap.
- Reduced memory consumption. malloc() headers, footers, and padding no longer waste memory. Also, these data structures are now located using a 16-bit offset off of the cmd rather than a 32-bit pointer.
- Easier to free. Since these allocations are now part of the code buffer, freeing them is done automatically when the code buffer is released.

- Better accounting of memory used. The `-Xjit:codeCacheSize` option is suppose to limit the amount of memory used by compiled methods. However, since some of the data being allocated for compiled methods was located outside of the code cache, total memory consumption for compiled code would actually exceed the specified size. In CDC 1.1, all memory allocated for compiled methods is now located in the code cache.

The four items that have been moved to the code cache are listed below. They are still accessed using the same CVMcmdXXX() macros, except now the macros rely on offsets from the start of the CMD rather than pointers to locate the data structures.

- `pcMapTable` - Table mapping between Java PCs and compiled PCs.
- `inliningInfo` - Table mapping compiled PCs to inlined methods.
- `gcCheckPCs` - Table of all instructions patched for GC check points.
- `stackMaps` - Table of stack maps for the method.

The following is the new layout of an allocated code buffer:

```
                 CVMUint32 size;
                 CVMCompiledMethodDescriptor* cmd;
                 pcMapTable[];  /* accessed using
CVMcmdCompiledPcMapTable() */
                 inliningInfo[]; /* accessed using
CVMcmdInliningInfo() */
                 gcCheckPCs[];  /* accessed using CVMcmdGCCheckPCs()
*/
                 CVMCompiledMethodDescriptor
                 <generated code>
                 stackmaps[];  /* accessed using CVMcmdStackMaps() */
                 CVMUint32 size;
```

Since the cmd is no longer located first in the code buffer, it no longer starts with the codeBufSizeX. Note however that there is still a size field at the start of the code buffer and at the end.

6.5 Decompilation

There are two ways a method can end up being decompiled. The first is when a class is unloaded, all compiled methods belonging to the class are decompiled and their code buffers freed. The second way a method can end up being decompiled is due to forced decompilation when there is a lack of space in the code cache.

There are two situations that result in forced decompilation. The first occurs while compiling a method and a request for a code buffer fails because of lack of space in the code cache. In this case an attempt is made to decompile enough methods to make room for the method. An emphasis is placed on decompiling a sequence of methods that result in freeing up a large enough buffer for the method, while also trying not to free up methods that don't aide in creating the needed buffer.

The second cause of forced decompilation is when during a GC it is discovered that the utilization of the code cache (bytes currently allocated) exceeds the value stored in `CVMglobals.jit.upperCodeCacheThreshold`. This value is settable as a percentage using `-Xjit:upperCodeCacheThresholdPercent` (default is 95%).

Regardless of the trigger for forced decompilation, an attempt is made to decompile enough methods so the utilization of the code cache is brought down to `CVMglobals.jit.lowerCodeCacheThreshold`, which is settable as a percentage using `-Xjit:lowerCodeCacheThresholdPercent` (default is 90%).

During forced decompilation, the `entryCounts` of all methods are "aged" to help locate less frequently called methods. When the `entryCount` reaches 0, then the method is a candidate for decompilation.

Aging the `entryCount` consists of doing a right shift of the value. At one point the `entryCount` was incremented every time the method was called, but this was determined to be too expensive. Instead it is initialized to 8 when the method is first compiled, and reset to 8 every time the method is found in a backtrace during a GC. It is also incremented when called from the interpreter loop, since this is an expensive operation anyway.

Forced decompilation is handled by `CVMJITcodeCacheAgeEntryCountsAndDecompile()`.

```
void
CVMJITcodeCacheAgeEntryCountsAndDecompile(
                CVMExecEnv* ee,
                CVMUint32 bytesRequested);
```

`bytesRequested` is 0 when called from GC. When called during compilation to free up a large enough buffer for the method being compiled, it is set to the estimated size of the required buffer.

6.6 Logical PC vs. Physical PC

During compilation, references to the code buffer can be either logical or physical. Physical refers to the actual physical address. Logical refers to the offset from the start of generated code (`CVMJITCompilationContext.codeEntry`). Usually logical addresses are used until it is time to actually emit an instruction to the code buffer.

The current physical and logical code buffer addresses are stored in the `CVMJITCompilationContext`, and are accessed with the following macros:

```
#define CVMJITcbufGetLogicalPC(con)  ((con)->curLogicalPC)

#define CVMJITcbufGetPhysicalPC(con) ((con)->curPhysicalPC)

#define CVMJITcbufLogicalToPhysical(con, logical) \ (&(con)-
>codeEntry[logical])
```

6.7 Pushing and Popping Fixup PCs

During compilation, sometimes a forward reference is generated to an address that is unknown until a few instructions later. A forward branch around data of unknown length is a common example of this. This is handled in the following matter:

1. Save away current `logicalPC`.
2. Emit the instruction with the forward reference, but use an offset of 0 for the forward reference address since the forward address is not known.
3. Emit code that varies in size and occurs before the forward address.
4. Forward address is now known. Call `CVMJITcbufPushFixup()`, using the address saved in (1). This changes the current `logicalPC` back to the PC of the instruction with the forward reference.
5. Properly emit the instruction with the forward reference.
6. Call `CVMJITcbufPopFixup()` to restore the proper logical PC.

The following example is from `jitgrammarrules.jcs`. The code is in charge of dumping the constant pool and emitting a branch around the constants:

```
                /* save away current logicalPC */
```

```
               CVMInt32 startPC = CVMJITcbufGetLogicalPC(con);
               /* emit dummy branch */
               CVMCPUemitBranch(con, startPC, CVMCPU_COND_AL);
               /* dump constant pool */
               CVMJITdumpRuntimeConstantPool(con, CVM_TRUE);
               /* save away current logicalPC */
  CVMJITcbufPushFixup(con, startPC);
               /* Emit branch around the constant pool dump: */
               CVMCPUemitBranch(con, endPC, CVMCPU_COND_AL);
               /* restore proper logicalPC */
               CVMJITcbufPop(con);
```

Note that this technique is not used for all forward references. For branches between basic blocks and for references to the constant pool, the forward references are automatically patched by `CVMJITfixupAddress()` when the forward address becomes known.

Also note that both of the above calls to `CVMCPUemitBranch()` must emit the same number of instructions. The first call is really just reserving code buffer space for the second call.

6.8 Emitting Code

Normally an instruction is emitted into the code buffer using `CVMJITcbufEmit()`. It will emit one instruction of size `CVMCPU_INSTRUCTION_SIZE` into the code buffer and increment `CVMJITCompilationContext.curLogicalPC` and `curPhysicalPC`. It will also check to make sure the size of the code buffer is not exceeded.

`CVMJITcbufEmit()` can also be used to emit data. However, it can only emit data of size `CVMCPU_INSTRUCTION_SIZE`, so multiple calls to `CVMJITcbufEmit()` are needed if the data is larger than an instruction. `CVMJITcbufEmit()` makes no attempt to align data.

`CVMJITcbufEmit()` assumes a uniform instruction size. If instruction sizes can vary, then `CVMCPU_INSTRUCTION_SIZE` should be set to 1, and either multiple calls to `CVMJITcbufEmit()` can be used to emit multiple bytes, or the `CVMJITcbufEmit1()`, `CVMJITcbufEmit2()`, and `CVMJITcbufEmit4()` macros can be used to emit instructions or data of size 1, 2, or 4 bytes respectively.

6.9 Copying Assembler Code to the Code Cache

Dynamically compiled code makes many calls to various helper functions written in assembler. On most platforms these calls need to be made using a multiple instruction long call. This is because the code cache is in the malloc heap, and is normally too far away from code linked with the CDC-HI Java runtime binary to call it with a single instruction.

In order to locate the assembler functions closer to the dynamically compiled code so they can be called with a single instruction, the assembler functions can optionally be copied into the start of the code cache where the dynamically compiled code resides. This usually results in better performance.

The copying of the assembler functions is handled by shared code. It is enabled by #define `CVM_JIT_COPY_CCMCODE_TO_CODECACHE`. This will cause all code between the symbols `CVMCCMcodeCacheCopyStart` and `CVMCCMcodeCacheCopyEnd` to be copied. Normally `ccmcodecachecopy_cpu.S` is used to properly setup these two symbols.

A table of all functions that are copied must be setup in `CVMJITinitCompilerBackEnd()`, which is usually implemented in `jitinit_cpu.c`. The functions in the table must appear in the same order that they appear in memory.

Note that if you enable `CVM_JIT_COPY_CCMCODE_TO_CODECACHE`, debugger breakpoints set in the assembler code won't function properly. If they are set after running CDC, then they will never be hit since the code cache copy of the functions are the ones actually used. If they are set before running CDC, then the trap instruction inserted by the debugger will get copied into the code cache copy. This will cause the breakpoint to be hit, but the debugger won't know it's a breakpoint and get will get confused.

6.10 Debugging Support

The following code cache manager functions are provided for debugging purposes. Calls to them can be made from a debugger like GDB if supported, or they can be added to VM code as necessary.

```
/*
 * Dump all free and allocated blocks in the code cache and
 * also do a bit of sanity checking.
 */
CVMBool
CVMJITcodeCacheVerify(CVMBool doDump);
/*
 * Find the method that the specified pc is in.
 */
CVMMethodBlock*
CVMJITcodeCacheFindCompiledMethod(CVMUint8* pc, CVMBool doPrint);
```

`CVMJITcodeCacheVerify()` is used to verify the integrity of the code cache, and also to dump its contents if requested.

`CVMJITcodeCacheFindCompiledMethod()` is used to locate the method owning the specified compiled pc. The `CVMMethodBlock*` of the method is returned, and the name of the method is also printed if requested.

6.11 Reference

Primary Source Files:

- `jitcodebuffer.h`
- `jitcodebuffer.c`

`-Xjit` Command Line Options:

- `codeCacheSize` - Size of the code cache
- `upperCodeCacheThreshold` - % full that will trigger decompilation
- `lowerCodeCacheThreshold` - Target % full when decompilation occurs
- `maxCompiledMethodSize` - Maximum size of a compiled method

`CVMJITGlobalState` fields:

- `CVMUint8* codeCacheStart; /* start of allocated code cache */`
- `CVMUint8* codeCacheGeneratedCodeStart;/* first method */`
- `CVMUint8* codeCacheEnd;  /* end of allocated code cache */`
- `CVMJITFreeBuf* codeCacheFirstFreeBuf;  /* list of free buffers */`
- `CVMUint32 codeCacheBytesAllocated;  /* bytes allocated */`
- `CVMUint32 codeCacheLargestFreeBuffer; /* largest free buffer */`

- `CVMUint32 upperCodeCacheThresholdPercent; /* start decompiling */`
- `CVMUint32 lowerCodeCacheThresholdPercent; /* stop decompiling */`
- `CVMUint32 upperCodeCacheThreshold; /* start decompiling */`
- `CVMInt32 lowerCodeCacheThreshold; /* stop decompiling */`

`CVMJITCompilationContext` fields:

- `CVMUint8*   codeBufAddr;  /* The allocated code buffer */`
- `CVMUint8*   codeBufEnd;  /* End of code buffer */`
- `CVMUint8*   codeEntry;   /* start of generated code */`
- `CVMUint16   bufSizeEstimate; /* est. size of code buffer */`
- `CVMInt32   curLogicalPC;`
- `CVMUint8*   curPhysicalPC;`
- `CVMInt32   logicalPCstack[4]; /* for CVMJITcbufPushFixup */`
- `CVMInt32   curDepth;     /* depth of logicalPCstack */`

CHAPTER 7

JIT Runtime Support

7.1 CVMglobals.jit

`CVMglobals.jit` (of type `CVMJITGlobalState`) holds all the global data for JIT runtime support. This includes:

- Compiler tuning parameters, e.g.
  - Compilation trigger parameters: costs and threshold
  - Code cache size
  - Inlining limits
- Compiler global state
- Runtime computed data tables
- Pointer to the code cache

7.2 CPU Cache Flushing

When a method is compiled, the generated code is written into a code buffer allocated from the code cache (see the Code Cache Manager). From the CPU's perspective, emitting generated code is the same as writing data out to memory.

7.2.1 Data Caches

For CPUs that implement a write-back instead of write-through data cache, this means that the generated code have been written to the data cache but may not yet have been written back to main memory.

7.2.2 Instruction Caches

For CPU's that implement an instruction cache, before code can be executed, it must first be loaded into the instruction cache. When the CPU tries to execute code at a certain memory location, it first checks to see if the content of that memory is already loaded into the instruction cache. If so, it will proceed with executing the code out of the instruction cache. If not, the memory content will be loaded into the instruction cache before it gets executed.

7.2.3 Cache Coherency

If the CPU's data cache and instruction caches are distinct, and the CPU implementation does not guarantee coherency between the 2 caches, then the dynamic compiler will have to ensure coherency is achieved for the memory region for a newly compiled method at the end of the compilation process.

If this is not done, unpredictable failures can occur. Here are some scenarios of how these failures can arise:

- The compiled method has been written to the data cache but not flushed to main memory yet. After compilation, the thread attempts to execute the compiled method. It gets the starting address of the method and branches to that address. This causes the CPU to load the memory content at that address from main memory into the instruction cache. But since main memory does not contain the code for the compiled method yet, the content loaded into the instruction cache is garbage.
- The compiled method occupies a code buffer. This method gets executed and loaded into the instruction cache. Later on, this method gets decompiled, and the code buffer is evacuated. A new method gets compiled and ends up occupying the same code buffer in memory. The thread then tries to execute this newly compiled method. The CPU checks the address of the method and finds that it is already in the instruction cache. So, it proceeds executing directly out of the instruction cache even though the instruction cache content is outdated and invalid.

To avoid these cache coherency problems, the dynamic compiler will first need to flush the data cache to make sure that the newly compiled method is written back out to main memory. Next, it needs to flush the instruction cache to ensure that outdated code doesn't wrongly get executed. The dynamic compiler does this by calling a cache flushing function that the target platform is supposed to provide:

```
extern void

CVMJITflushCache(void* begin, void* end);
```

`CVMJITflushCache` need only flush the data and instruction cache lines which correspond to the address range specified by the begin and end arguments. The data cache must always be flushed first, before the instruction cache is flushed. There's no harm in flushing more cache lines than the minimum needed, except in some performance loss due to some perhaps unnecessary cache misses. You might need to do this if the only available target platform API for flushing the caches is to flush the entire cache.

7.2.4 External Caches

If external caches are unified (no distinction between data and instruction caches), then there is no need to flush them. Otherwise, they will need to be flushed as well.

7.3 `CVMCompiledMethodDescriptor`

For every compiled method, there is one `CVMCompiledMethodDescriptor` struct (commonly referred to as the CMD) that is associated with it. The CMD is always located in the code cache immediately before the generated code for the method. See the Code Cache Manager for details about the layout of the code buffer allocated for compiled methods.

7.3.1 Computing the CMD

In CVM, methods are primarily identified using a CVMMethodBlock pointer (commonly referred to as an MB). If a method is compiled, given its MB, the CMD can be computed as follows:

```
cmd = (((CVMCompiledMethodDescriptor *)mb.startPCX) - 1)
```

The `startPCX` in the MB contains the address of the generated code for the compiled method. The CMD is computed by subtracting the size of the `CVMCompiledMethodDescriptor` from this address.

7.4 Entry to Compiled Code

There are 2 entry points to a compiled method:

- **Calling from Outside Compiled Code -** From outside of compiled code (i.e. the interpreter), when the compiled method is called, the caller will setup its frame and just branch to the code address indicated in the `startPCFromInterpretedX` field of the method's CMD.
- **Calling from Compiled Code -** From inside compiled code, the caller just sets up a return address and branches to the address indicated in the `jitInvokerX` field of the MB. For a compiled method, the `jitInvokerX` field will point to the same location as the startPCX field, i.e. the first line of generated code of the compiled method.

  The difference between `startPCX` and `startPCFromInterpretedX` is that `startPCX` includes prologue code to setup the stack frame for this method while `startPCFromInterpretedX` skips the prologue code and starts in the method body.

  When calling from compiled code to compiled code, the callee is responsible for setting up the callee stack frame through the use of prologue code. When calling compiled code from outside code, the caller is responsible for setting up the callee stack frame. Hence, the prologue code should not be executed when the compiled code is called from outside code.

7.5 Assembler Glue

To transfer execution control from the VM to compiled code, the VM calls `CVMJITgoNative`.

```
extern CVMMethodBlock*
CVMJITgoNative(CVMObject* exceptionObject, CVMExecEnv* ee,
                                    CVMCompiledFrame* jfp, void* pc);
```

`CVMJITgoNative` is a piece of assembler glue code that the target platform port must provide. `CVMJITgoNative` is responsible for setting up a native stack frame for use by the compiled code. This native stack frame includes memory reserved for the `CVMCCExecEnv` struct (see `CVMCCExecEnv` below).

Another important piece of assembler glue that is called by VM code is `CVMJITexitNative`.

```
extern void
CVMJITexitNative(CVMCCExecEnv* ccee);
```

`CVMJITexitNative` is used for unwinding native frames back to the interpreter frame which called the compiled code (i.e. the frame just before the one that was pushed by `CVMJITgoNative`). This is used for throwing Java exceptions (see Throwing Exceptions below).

7.6 Helper Functions

The dynamic compiler does not generate inline code for every possible VM operation. For some operations, the dynamic compiler will generating code to call compiled code runtime helper functions instead.

Why use helper functions instead of generating inlined code to do all the work?

- To reduce the amount of work to get a JIT port up and running quickly, a lot of functionality can be implemented initially in terms of calls to helpers. The performance of the compiled code may not be the most optimal but the port will reach functional completeness quicker. Optimizations can subsequently be applied gradually to targeted areas as schedule permits.
- Some functionality is complex and not suitable for being inlined into the compiled code for the following reasons:
  - It may be difficult to generate code for such complex functionality.
  - Such complex functionality may take up a lot of code space, thereby resulting in inefficient usage of compiled code cache space.
  - The functionality may inherently take a long time to execute. Inlining it into the compiled method would not yield any noticeable performance gain anyway.

7.6.1 Default C Helper Functions

CDC HI provides some default C helper functions. These helper functions include:

- Integer division and remainder
- Long (64-bit) arithmetic, logical and shift operators
- Single and double precision floating point operators
- Primitive type converters (e.g. integer to float, float to double, etc.)
- Comparators (long, float, and double)
- Java object allocation
- Java synchronization
- Constant pool resolution
- Static / class initialization
- Throwing Java exceptions
- Checkcast, instanceof, interface MB lookup, array assignability test
- GC rendezvous

Note that some of these helper functions like integer division is provided in case the CPU hardware does not have an instruction set that supports this operator. Again, having these helper functions reduces the amount of work needed to get the dynamic compiler up and running initially.

7.6.2 ASM Helper Functions

Calling C helper functions from compiled code does incur function call overhead at runtime. To optimize for speed (where appropriate), the dynamic compiler may choose to implement some of these helper functions in assembler code.

Hand-crafted assembler code helpers can take advantage of knowledge about the internals of compiled code, such as the reuse of the compiled code native frame and the ccee (see Code Execution & Stack Frames below). This can help reduce or eliminate the cost of the function call overhead.

In the ARM port, an example of this type of assembler helper are the single and double precision floating point operator functions.

One of the main reasons for using helper functions in the first place is because the work done by these helpers is inherently complex and difficult to implement. However, some of this work may have a fast path which is easier to implement and a slow path which is more complex and difficult to implement. To optimize for speed, the dynamic compiler port may want to implement the fast path in assembler and defer to the C helpers for the slow path case. These types of assembler helpers are also commonly referred to as assembler glue functions.

In the ARM port, this type of assembler glue includes:

- `CVMCCMruntimeNewGlue`
- `CVMCCMruntimeNewArrayGlue`
- `CVMCCMruntimeANewArrayGlue`
- `CVMCCMruntimeMultiANewArrayGlue`
- `CVMCCMruntimeCheckCastGlue`
- `CVMCCMruntimeInstanceOfGlue`
- `CVMCCMruntimeCheckArrayAssignableGlue`
- `CVMCCMruntimeLookupInterfaceMBGlue`
- `CVMCCMruntimeMonitorEnterGlue`
- `CVMCCMruntimeMonitorExitGlue`

Another reason to use assembler glue is to to do marshaling of arguments before calling C helpers. This simplifies the dynamic compiler port by alleviating it from having to emit code to do this marshaling.

In the ARM port, examples of this type of glue code includes:

- `CVMCCMruntimeGCRendezvousGlue`
- `CVMCCMruntimeThrowNullPointerExceptionGlue`
- `CVMCCMruntimeThrowArrayIndexOutOfBoundsExceptionGlue`
- `CVMCCMruntimeThrowDivideByZeroGlue`
- `CVMCCMruntimeThrowObjectGlue`

A third reason to use assembler glue is to code rewriting of the compiled code that called it. This is done for code paths that are only expected to be executed once. Once this path has been executed, the caller can be patched so that this helper need not be called again the next time this code path is taken.

In the ARM port, examples of this type of glue code includes:

- `CVMCCMruntimeRunClassInitializerGlue`
- `CVMCCMruntimeResolveGlue`

Another side benefit of using assembler glue code is that it can take care of flushing execution state like the program counter (PC), the Java frame pointer (JFP), and the Java stack pointer (JSP) to the compiled frame on the Java stack before calling to a C helper function. This again alleviates the dynamic compiler from having to generate code to do this work when calling helper functions.

The above lists of assembler glue functions can be used as an indicator of assembler glue that any dynamic compiler port may want to consider implementing.

7.6.3 Disabling Default Helper Functions

If the dynamic compiler port chooses to implement its own assembler (or even C) helper function which can perform better than the default C helpers, it will want to disable the defaults ones so that the target specific implementations can use the same helper names without a linker conflict.

The `src/<cpu>/javavm/include/jit/ccm_cpu.h` file should contain a list of `#define` or `#undef` of symbols that look like `CVMCCM_HAVE_PLATFORM_SPECIFIC_<helper name>`. This list is based on the one specified in the comments in `src/share/javavm/include/porting/jit/ccm.h`.

For example, in the ARM port, we have:

```
#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LMUL

#undef CVMCCM_HAVE_PLATFORM_SPECIFIC_LDIV
```

`#define CVMCCM_HAVE_PLATFORM_SPECIFIC_LMUL` indicates that the target port will provide its own implementation of the helper function for the long multiplication operator.

`#undef CVMCCM_HAVE_PLATFORM_SPECIFIC_LDIV` indicates that the target port will not provide its own implementation of the helper function for long division. Instead, it will rely on the the default C helper function implementation.

So, to disable the default helper functions, just `#define` these symbols in the appropriate `ccm_cpu.h` file.

Another reason why the target port will want to disable these helper functions is that the dynamic compiler may choose to implement the functionality by generating inlined code. For example, if the target CPU provides an integer division instruction, then there is no need to use the C helper for integer division. The C helper should be disabled so as to not link in unnecessary code.

7.7 `CVMCCExecEnv`

The `CVMCCExecEnv` (commonly referred to as the ccee) serves as a scratch area for the use of the compiled code. Compiled code does not allocate automatic variables nor push items onto the native stack. Its usage of the native stack is fixed and is known ahead of time. This allows for stack size checks that ensure that the native stack is overflowed during execution. For its own stack scratch area, compiled code will only use memory allocated within the ccee.

7.8 Code Execution and Stack Frames

Calling from the interpreter to compiled code, a `CVMCompiledFrame` frame is pushed on the Java stack, and a native stack frame is pushed by `CVMJITgoNative`. But once in compiled code, invocation of another compiled method only requires a new `CVMCompiledFrame` to be pushed on the Java stack. The same native stack frame will be reused by the new compiled method. Only one native stack frame (and therefore only one ccee) is needed for compiled code per recursion of the interpreter loop. The execution state of a compiled method is stored in its `CVMCompiledFrame` on the Java stack and not in the native stack frame. The native stack frame only provides some scratch area for temporary use.

If the compiled method invokes a JNI native method which recurses into the interpreter loop, which in turn invokes another compiled method, then a new native stack frame (and ccee) will be pushed for that new compiled method.

The following examples illustrate how the Java and native stacks are used by compiled code:

CODE EXAMPLE 7-1 Example 1 - Calling from an interpreted method to a JNI native method

Before:

```
Java Stack:  Interpreted Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod
```

After:

```
Java Stack:  Interpreted Frame 1 ->
         JNI Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeJNIHelper ->
           CVMjniInvokeNative ->
            JNI Native Method
```

CODE EXAMPLE 7-2 Example 2 - Calling from an interpreted method to a compiled method

Before:

```
Java Stack:  Interpreted Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod
```

After:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative
```

Note that the interpreter needs to go through the CVMJITgoNative glue to invoke the compiled code.

CODE EXAMPLE 7-3 Example 3 - Calling from a compiled method to a compiled method

Before:

```
Java Stack:  Compiled Frame 1
Native Stack: CVMJITgoNative
```

After:

```
Java Stack:  Compiled Frame 1 ->
         Compiled Frame 2
Native Stack: CVMJITgoNative
```

Note that calling from a compiled method to a compiled method does not require another native stack frame to be pushed.

CODE EXAMPLE 7-4 Example 4 - Calling from a compiled method to an interpreted method

Before:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative
```

After:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1 ->
           Interpreted Frame 2
Native Stack: CVMgcUnsafeExecuteJavaMethod
```

Note that calling from a compiled method to an interpreted method does not recurse into the interpreter loop. Instead, the compiled code returns to the interpreter loop that called it in the first place. Even though the compiled code native frame (`CVMJITgoNative`) has been popped, the compiled code frame is still on the Java stack. This re-iterates that the execution state of the compiled methods are stored on the Java stack and not on the native stack. The native stack frame only provides a scratch area for the use of the compiled code.

CODE EXAMPLE 7-5 Example 5 - Calling from a compiled method to a JNI native method

Before:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative
```

After:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1 ->
           JNI Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative ->
            CVMinvokeJNIHelper ->
              CVMjniInvokeNative ->
               JNI Native Method
```

Compiled methods are able to call JNI methods without having to return to the interpreter loop.

CODE EXAMPLE 7-6 Example 6 - Calling from compiled to JNI to compiled

Before:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1 ->
           JNI Frame 1
```

```
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative ->
            CVMinvokeJNIHelper ->
              CVMjniInvokeNative ->
               JNI Native Method
```

After:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1 ->
           JNI Frame 1 ->
            Compiled Frame 2
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative ->
            CVMinvokeJNIHelper ->
              CVMjniInvokeNative ->
               JNI Native Method ->
                 CVMgcUnsafeExecuteJavaMethod ->
                  CVMinvokeCompiledHelper ->
                    CVMJITgoNative
```

Note that the JNI method's attempts to invoke any Java methods causes recursion into the interpreter loop. If a compiled method is invoked sub of that, another native stack frame for compiled code (`CVMJITgoNative`) will be pushed on the stack.

CODE EXAMPLE 7-7 Example 7 - Calling from a compiled method to a C or Assembler helper

Before:

```
Java Stack:  Compiled Frame 1
Native Stack: CVMJITgoNative
```

After:

```
Java Stack:  Compiled Frame 1
Native Stack: CVMJITgoNative ->
         C/Assembler helper function
```

CODE EXAMPLE 7-8 Example 8 - Returning from an interpreted method to a compiled method

Before:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1 ->
```

```
          Interpreted Frame 2
Native Stack: CVMgcUnsafeExecuteJavaMethod
```

After:

```
Java Stack:  Interpreted Frame 1 ->
         Compiled Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative
```

In order to return execution to compiled code, the interpreter needs to re-establish the ccee for the use of the compiled code. Hence, it calls `CVMJITgoNative` to push the native stack frame for compiled code which allocates the ccee as well. Note that `CVMJITgoNative` is used in this case to continue execution in the middle of a compiled method (`CVMJITgoNative` is given the return address compiled PC) whereas in previous examples, it is used for starting execution at the start of the compiled method.

7.9 Throwing Exceptions

For compiled code, exceptions can be thrown from within assembler glue code or from C helper functions. After an exception is thrown, execution should transfer to an appropriate catch block rather than resuming at the return location in the compiled code that called the glue code or helper function.

To do this, the glue code or helper functions will call `CVMJITexitNative`. This causes execution to transfer resume at the interpreter frame that called `CVMJITgoNative` as if `CVMJITgoNative` had returned with a return value and VM state that tell the interpreter that an exception has been thrown. Note that this execution transfer pops the compiled code native stack frame and all C helper frames above it off the native stack, but the frames on the Java stack remains unchanged.

For example,

CODE EXAMPLE 7-9

Before throwing exception:

```
Java Stack:  Interpreted Frame 1 ->
         Interpreted Frame 2 ->
           Compiled Frame 1 ->
             Compiled Frame 2
```

```
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative ->
            C/Assembler helper function
```

After throwing exception:

```
Java Stack:  Interpreted Frame 1 ->
         Interpreted Frame 2 ->
           Compiled Frame 1 ->
             Compiled Frame 2
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMgcUnsafeHandleException
```

`CVMgcUnsafeHandleException` is the VM's function for finding the appropriate catch block for the exception and for unwinding the Java stack to the appropriate frame of the method with that catch block.

The `try-catch` ranges for methods are kept in terms of bytecode PC (program counter) values. There is no corresponding ranges for compiled code. For an exception thrown in compiled code, the compiled PC at which the exception is thrown is first mapped into its bytecode PC equivalent. Next, the interpreter will search through the `try-catch` ranges (for the methods whose frames are on the Java stack) for a matching `catch` block.

When it finds a matching `catch` block, it will attempt to resume execution at the start of the catch block. The `catch` block may or may not reside in the method that threw the exception.

If the `catch` block is in an interpreted method, then the interpreter loop will start interpreting the bytecodes at the start of the `catch` block.

CODE EXAMPLE 7-10 Before `CVMgcUnsafeHandleException` unwinds the Java stack:

```
Java Stack:  Interpreted Frame 1 ->
         Interpreted Frame 2 ->
           Compiled Frame 1 ->
            Compiled Frame 2
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMgcUnsafeHandleException
```

CODE EXAMPLE 7-11 After exception is caught in Interpreted Frame 1:

```
Java Stack:  Interpreted Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod
```

If the `catch` block is in a compiled method, then the interpreter will map the bytecode PC of the start of the catch block into its corresponding compiled PC. Next, the interpreter will call `CVMJITgoNative` to transfer execution back to compiled code in a similar way as if returning from interpreter code to compiled code. In this case, `CVMJITgoNative` is given the compiled PC of the start of the catch block.

CODE EXAMPLE 7-12 Before `CVMgcUnsafeHandleException` unwinds the Java stack:

```
Java Stack:  Interpreted Frame 1 ->
         Interpreted Frame 2 ->
           Compiled Frame 1 ->
            Compiled Frame 2
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMgcUnsafeHandleException
```

CODE EXAMPLE 7-13 After exception is caught in Compiled Frame 1:

```
Java Stack:  Interpreted Frame 1 ->
         Interpreted Frame 2 ->
           Compiled Frame 1
Native Stack: CVMgcUnsafeExecuteJavaMethod ->
         CVMinvokeCompiledHelper ->
           CVMJITgoNative
```

7.10 On-Stack Replacement (OSR)

While interpreting a method, the interpreter may discover that the method got compiled (perhaps by another thread or by the current thread at backwards branches). If the method is now compiled, the interpreter will want to run the compiled version instead of continuing with the interpreted version. In order to do this, the interpreter will need to be able to replace the interpreted frame on the Java stack with an equivalent compiled frame first. The mechanism for doing this is called on-stack replacement (OSR) of frames.

As of version 1.0.1, CDC-HI is able to perform cases of OSR that match the following criteria:

- The OSR point occurs at the target of a backwards branch (i.e. the start of a basic block).
- The operand stack at that OSR point is empty.
- The OSR is to replace an interpreted frame with a compiled frame.

The OSR mechanism relies on the fact that under the above criteria there isn't much difference between the interpreted frame and the compiled frame. The only differences are:

- The compiled frame has more locals. But the locals that exist in the interpreted frame are mapped exactly the same as the first set of locals in the compiled frame. Hence, the interpreted frame locals can just be copied into the compiled frame at the start of its locals area.
- The compiled frame has a slightly different structure of the frame record. However, the information needed to fill out the compiled frame's frame record can be inferred from the information in the interpreted frame. The bytecode PC in the interpreted frame will, of course, have to be mapped to its equivalent compiled PC.
- The compiled frame has a temp area. But at the start of basic block, this temp areas is basically uninitialized. This means OSR only need to reserve space for the temp area and not worry about initializing it.
- The compiled frame also has an operand stack but perhaps with a greater maximum depth. The fact that the interpreted frame operand stack is empty at the OSR point also means that the compiled frame operand stack is empty. There is nothing to do except to reserve space for the compiled frame operand stack.

Once the OSR mechanism has rewritten the interpreted frame into an equivalent compiled frame, the interpreter calls `CVMJITgoNative` to transfer execution to the compiled code. In this case the compiled PC for the start of the OSR point will be passed to `CVMJITgoNative`.

Note that OSR's feasibility relies on the properties describe above with regards to the interpreted and compiled frames. If the layout and use of compiled frames change and these properties cannot be preserved, then the current CDC-HI OSR mechanism may not work correctly and will either have to be disable or modified.

CHAPTER 8

JIT Memory Manager

CDC-HI includes a memory manager for use by the JIT that provides services above and beyond what is typically provided by `malloc()`. There are two types of memory the JIT will allocate:

- Permanent, which is in use as long as the method remains compiled.
- Transient, which is only in use while the method is being compiled.

8.1 Permanent Memory Allocation

Permanent memory allocations are treated much like `malloc()` allocations. They are allocated by calling `CVMJITmemNewLongLivedMemory()` and freed when a method is decompiled by calling `CVMJITmemFreeLongLivedMemory()`. These APIs just make use of `calloc()` and `free()`. The advantage of using them is that they also provide the Memory Fence and Statistics Gathering features described below.

8.2 Transient Memory Allocation

In order to simplify the handling of transient memory allocations, transient memory allocations are not tracked and explicitly freed. Instead, 8k chunks of memory are allocated by the Memory Manager using `calloc()`, and transient memory is allocated by the Memory Manager from these chunks. When compilation is complete, all the allocated chunks are freed.

There are a few advantages to this approach:

- Simplicity: individual transient memory allocations are not tracked and don’t have to be freed.

- Speed: allocation is very fast, plus there is little overhead in freeing the chunks.
- Efficiency: since the transient allocations are not tracked, there is no memory overhead for allocation headers and trailers. Also, these allocations only need to be 4-byte aligned, whereas `calloc()` does 8-byte alignment.

Transient memory is allocated by using `CVMJITmemNew()`. When compilation is complete, all transient memory is freed by making a single call to `CVMJITmemFlush()` to free all the allocated chunks.

8.3 Memory Fence

For debugging purposes, the Memory Manager can be compiled with support for putting a memory fence around each allocation. The fence consists of header and trailer words that are checked periodically during compilation and decompilation. These checks are enabled by building with `CVM_DEBUG_ASSERTS=true`.

For transient memory, a check of all transient allocations is made when `CVMJITmemFlush()` is called. However, if `jitmemory.c` is modified to also `#define CVM_DO_INTENSIVE_MEMORY_FENCE_VALIDATION`, then a check is made every time `CVMJITmemNew()` is called.

For permanent memory, a check of all permanently allocated memory is made when `CVMJITmemFreeLongLivedMemory()` is called, but will also be made when `CVMJITmemNewLongLivedMemory()` is called if jitmemory.c is modified to `#define CVM_DO_INTENSIVE_MEMORY_FENCE_VALIDATION`.

If validation of a memory fence fails, an assertion is made and information about the failing allocation is displayed, including the file name and line number where the allocation was made from.

8.4 Statistics Gathering

One of the arguments to `CVMJITmemNew()` is a tag value. One of the following `enum` values should be passed in for the tag argument:

```
                enum CVMJITAllocationTag {
                    JIT_ALLOC_IRGEN_NODE = 0,/* IR node allocation
*/
                    JIT_ALLOC_IRGEN_OTHER,  /* Other front-end
allocation */
```

```
                    JIT_ALLOC_OPTIMIZER,   /* Optimizer allocations
*/
                    JIT_ALLOC_CGEN_REGMAN,  /* working memory for
registers */
                    JIT_ALLOC_CGEN_ALURHS,  /* working memory for
aluRhs */
                    JIT_ALLOC_CGEN_MEMSPEC, /* working memory for
memSpec */
                    JIT_ALLOC_CGEN_FIXUP,  /* working memory for
fixups */
                    JIT_ALLOC_CGEN_OTHER,  /* Other code generator
memory */
                    JIT_ALLOC_COLLECTIONS,  /* sets, growable
arrays, etc. */
                    JIT_ALLOC_DEBUGGING,   /* Allocations for
debugging */
                    JIT_ALLOC_NUM_ALLOCATION_KINDS
                };
```

The purpose of the tag is for statistics gathering and is only used when building with `CVM_JIT_COLLECT_STATS=true` and executing with `-Xjit:stats=minimal,trace=stats`. After compilation of each method, details about the transient memory usage will be traced to the console. Upon VM exit, a summary of memory usage for all compilations will be traced to the console.

8.5 Reference

Primary Source Files:

`jitmemory.h`

`jitmemory.c`

APIs

`CVMJITmemNew()`

`CVMJITmemFlush()`

`CVMJITmemNewLongLivedMemory()`

`ory()`

Build Options:

```
CVM_JIT_COLLECT_STATS=true
```

Command Line Options:

```
-Xjit:stats=minimal,trace=stats
```

CHAPTER **9**

Constant Pool Manager

The Constant Pool Manager is used during dynamic compilation to manage 32 and 64 bit constants that are referenced by generated code. Although many smaller constants can be encoded within an instruction (immediate value for ALU instruction or immediate offset for load/store instruction), large constants usually need to be stored in memory and loaded into a register for use. Most large constants are addresses of data structures that need to be referenced or functions that need to be called, but some are just large numeric constants whose origins are in the Java bytecodes.

The Constant Pool Manager provides a mechanism for accumulating these constants into large pools, which limits the number of duplicates. It also allows constants to be grouped together, which improves processor cache performance. Also, when used in conjunction with the Register Manager, it is possible to track which constants are already in registers so constants don't need to be reloaded every time they are referenced.

Note – The constant pools being discussed in this section should not be confused with the per class Java constant pools, whose origins are the constant pool in the class file, and are referenced using the `CVMcbConstantPool()` macro.

9.1 Loading Constants into Registers

When dealing with large constants, a platform has three choices on how to load the constants into a register.

1. Load from a PC relative constant pool. This approach is only supportable on a small number of platforms which allow PC relative loads, such as ARM. Multiple constant pools may be needed within a method if the offset range of a PC relative load is smaller than the size of the method. The constant pools are private to the method.
2. Load from an offset off of a dedicated constant pool base register. This approach makes use of a dedicated GPR as the base register for the constant pool. The base register (called `CVMCPU_CP_REG`) is setup on method entry and whenever execution returns to the method. Each method has its own constant pool.
3. Use multiple ALU instructions to build the constant. This approach is popular for processors that can build any 32-bit constant with at most two instructions. Constants created in this way do not use the Constant Pool Manager.

The preference between (2) and (3) usually depends on the availability of an extra GPR as the base register, the cost of doing a load, and the cost of doing two ALU instructions to build the constant. Having two ALUs often tips the scales towards (3). The ability to do instruction scheduling (something not currently done) also can affect the choice.

9.2 Adding Constants to the Constant Pool

Constants are added to the constant pool using `CVMJITgetRuntimeConstantReference32()` and `CVMJITgetRuntimeConstantReference64()`. They both return the logical address of the constant in the constant pool. The caller is required to immediately use the offset in a load instruction that will load the constant. No other instruction can be emitted before doing this.

In most cases the constant has not yet been emitted, and an offset of 0 is returned. The Patching Forward Constant Pool References section below describes how this is handled properly.

Note that typically `CVMJITgetRuntimeConstantReference32()` is only called from the platform specific `CVMCPUemitLoadConstant()` function, and all code that desires a constant in a register should call `CVMCPUemitLoadConstant()`. This allows `CVMCPUemitLoadConstant()` to determine how the constant should be loaded, because even if there is a constant pool, it may be possible to load the constant with just one ALU instruction if the constant is relatively small, thereby bypassing the constant pool.

9.3 Dumping the Constant Pool

The Constant Pool can be dumped by calling `CVMJITdumpRuntimeConstantPool()`. When using a dedicated base register, this should not be done until all code for the method has been compiled. When using the PC as a base register, it can be done at any point, assuming code is also generated to branch around the constant pool.

When using PC relative constant pools, periodically `CVMJITcpoolNeedDump()` should be called to check if the distance between the current PC and the furthest unresolved reference to a constant is getting close to the maximum offset allowed by a PC relative load. `CVMJITcpoolNeedDump()` will call `CVMJITcanReachAddress()` for each unresolved forward reference to a constant to make sure that the reference is not in danger of becoming out of range of a PC relative load. `CVMJITcanReachAddress()` must be implemented for each processor port if a PC relative constant pool is being used.

9.4 Patching Forward Constant Pool References

As mentioned above, usually when `CVMJITgetRuntimeConstantReference32()` and `CVMJITgetRuntimeConstantReference64()` are called, the constant is not yet dumped and an offset of 0 is returned. When this happens, the Constant Pool Manager adds the current logical address to the list of address that need to be patched (or “fixed up”) when the constant is eventually emitted by `CVMJITdumpRuntimeConstantPool()`. This is why the caller must immediately generate a reference to the constant. Otherwise the wrong instruction will get patched.

Patching is done by having `CVMJITdumpRuntimeConstantPool()` call `CVMJITfixupAddress()` for each unresolved forward reference to a constant. Each processor port must properly implement `CVMJITfixupAddress()` to handle this patching. Usually this just involves masking the offset bits into the existing instruction.

Note that `CVMJITfixupAddress()` must rewrite the instruction with the same number of instructions as were originally emitted when the unresolved forward reference to the constant was first emitted. If the load of the constant will vary in instruction length based on how far the offset of the constant is, then this may require being pessimistic about how many instructions are needed when first emitting the instruction

9.5 CVMCPU_HAS_CP_REG

The presence of the `CVMCPU_HAS_CP_REG` macro tells the Constant Pool Manager that there will be a dedicated constant pool base register rather than a PC relative base register. The Constant Pool Manager will behave differently based on the setting of the macro.

9.6 Register Manager Usage

The Register Manager is capable of keeping track of resources that map to constants, and more importantly, keeping track of constants that have already been loaded into registers. This is done by calling `CVMRMbindResourceForConstant32()` or `CVMRMgetResourceForConstant32()` when associating a constant with a resource.

`CVMRMbindResourceForConstant32()` will defer the actual loading of the constant until the resource is pinned, while `CVMRMgetResourceForConstant32()` will ensure that resource is already loaded into the register. Both result in the resource containing the constant being registered with the Register Manager, so the next time either is called again for the same constant, the same resource will be returned. This allows reuse of a constant already loaded into a register.

When a resource containing a constant is pinned and the register is not already loaded with the constant, this will result in the Register Manager `reloadConstant32()` function being called, which in turn will call `CVMCPUemitLoadConstant()` to actually load a constant into the register.

9.7 Typical Code Generation

CODE EXAMPLE 9-1 PC Relative (ARM)

```
                 0x40756524  936:  ldr  r8, [pc, #+0]  @ 
cardTableVirtualBase
```

```
                    ...
                    0x4075658c  1040: .word 2532847 @
cardTableVirtualBase
                    :::::Fixed instruction at 936 to reference 1040
```

CODE EXAMPLE 9-2 CP Base Register (PowerPC)

```
                    0x30628848  1056: lwz  r22, 0(rCP)   @
cardTableVirtualBase
                    ...
                    0x306288b8  1168: .word 271622479    @
cardTableVirtualBase
                    :::::Fixed instruction at 1056 to reference 1168
```

CODE EXAMPLE 9-3 Multiple ALU Instructions (Sparc)

```
                    0x00ae7bd4  1228:sethi 11081, %l5   @
cardTableVirtualBase
                    0x00ae7bd8  1232:or %l5, #83, %l5
```

Note that in the first two examples above that use a constant pool, the initial offset of the constant is 0, but after the constant is dumped the instruction is "fixed" to have the proper offset. The instruction with the proper offset is not shown above since no code for it is traced.

9.8 Reference

Primary Source Files:

`jitconstantpool.h`

`jitconstantpool.c`

Macros:

`CVMCPU_HAS_CP_REG`

Functions:

`CVMJITgetRuntimeConstantReference32`

`CVMJITgetRuntimeConstantReference64`

`CVMJITdumpRuntimeConstantPool`

`CVMJITcpoolNeedDump`

`CVMJITcanReachAddress`

`CVMJITfixupAddress`

`CVMRMbindResourceForConstant32`

`CVMRMgetResourceForConstant32`

CHAPTER **10**

Register Manager

The Register Manager has responsibility for keeping track of register usage during compilation. It could more appropriately be called the Resource Manager, because what it really does is use a data structure called `CVMRMResource` to keep track of where evaluated expressions are currently stored, both in memory and in registers.

10.1 Evaluated Expressions

Evaluated expressions are IR nodes that have applied some sort of operation on their subnode argument(s), such the adding the lhs and rhs subnodes of an ADD IR node. Some expressions are implicitly evaluated, such as locals and constants, which don't require that an operation be performed other than loading from memory. Evaluated expressions can reside in one of five possible locations:

- In the locals area of the frame
- In the temp (spill) area of the frame
- On the parameter stack of the frame
- In a constant pool
- In a register

The resource manager will track in which of the above five locations a resource is currently being stored in. The first four locations are all in memory. If an evaluated expression is in a register, it can at the same time be stored in any one of the above four memory locations.

10.2 Interactions with the JCS Grammar

The main clients of the Register Manager are the semantic actions of the JCS grammar rules. An understanding of how the rules interact with the Register Manager is best explained using a simple example. A general understanding of JCS grammar rules is necessary for this example. Consider the following expression:

```
 y + 1000;
```

The IR for this expressions will consist of an IADD32 node with a LOCAL32 node on the lhs and a CONSTANT32 node on the rhs. The LOCAL32 node for y is handled by the following rule:

```
                 reg32:LOCAL32 : 10 : : : : {
                     CVMJITRMContext *rc = CVMRM_INT_REGS(con);
                     CVMJITLocal*  l = CVMJITirnodeGetLocal( $$ );
                     CVMBool isRef =
CVMJITirnodeIsReferenceType($$);
                     CVMRMResource* dest = CVMRMloadJavaLocal(
                         rc, GET_REGISTER_GOALS, 1, isRef, l-
>localNo);
                     CVMRMoccupyAndUnpinResource(rc, dest, $$);
                     pushResource(con, dest);
                 };
```

The key here is that a resource is created for the local by calling `CVMRMloadJavaLocal()`, and then the resource is pushed onto the grammar semantic stack for use by the rule that consumes the local (the IADD32 rule in this case).

Next, the constant 1000 is handled by the following rule:

```
                 reg32: ICONST_32 : 20 : : : : {
                     CVMJITRMContext *rc = CVMRM_INT_REGS(con);
                     CVMInt32 constant =
CVMJITirnodeGetConstant32($$)->j.i;
                     CVMRMResource *dest =
                         CVMRMbindResourceForConstant32(rc,
constant);
                     CVMRMoccupyAndUnpinResource(rc, dest,
thisNode);
                     pushResource(con, dest);
```

```
                    };
```

This rule creates a resource for the constant by calling `CVMRMbindResourceForConstant32()`. Just like the LOCAL32 rule above, it also pushes this resource to the grammar semantic stack. Now resources for both operands of the “`y + 1000`“ expression are on the grammar semantic stack.

Next, the add is handled by the following rule:

```
                    reg32: IADD32 reg32 reg32 : 10 : : : : {
                        CVMJITRMContext *rc = CVMRM_INT_REGS(con);
                        CVMRMResource* rhs = popResource(con);
                        CVMRMResource* lhs = popResource(con);
                        CVMRMResource* dest =
                            CVMRMgetResource(rc, GET_REGISTER_GOALS,
1);
                        CVMRMpinResource(rc, lhs, CVMRM_ANY_SET,
CVMRM_EMPTY_SET);
                        CVMRMpinResource(rc, rhs, CVMRM_ANY_SET,
CVMRM_EMPTY_SET);
                        CVMCPUemitBinaryALU(con,
                            CVMCPU_ADD_OPCODE,
                            CVMRMgetRegisterNumber(dest),
                            CVMRMgetRegisterNumber(lhs),
                            CVMRMgetRegisterNumber(rhs));
                        CVMRMrelinquishResource(rc, lhs);
                        CVMRMrelinquishResource(rc, rhs);
                        CVMRMoccupyAndUnpinResource(rc, dest,
thisNode);
                        pushResource(con, dest);
                    };
```

The IADD32 rule does the following:

1. Pops the resources for the two operands off of the grammar semantic stack.
2. Creates a new resource for the result of the add by calling `CVMRMgetResource()`, which returns a resource already bound to a register.
3. Calls `CVMRMpinResource()` for the two operands, which forces them into registers.
4. Calls `CVMCPUemitBinaryALU()` to emit the add instruction.

5. Releases both of the operand resources, allowing them to be deleted if no other references are being made to them.
6. Pushes the result resource onto the grammar semantic stack for use by the rule that consumes the expression, such as an assignment to a local or a method argument.

Although the above rules are similar to those used in jitgrammarrules.jcs for the CDC-HI RISC ports, some liberties have been taken to make this example easier to read. Most notably the rhs node of the IADD32 rule should be an aluRhs node, not a reg32 node. However, making it a reg32 node simplifies the example.

More details on the JCS grammar can be found in a the JCS chapter.

10.3 CVMRMResource

As mentioned above, a Register Manager resource is used to track the state of evaluated expressions, and this is done using the `CVMRMResource` struct:

```
struct CVMRMResource {
                    CVMJITIdentityDecoration dec;
                    CVMUint16  flags;
                    CVMInt8   regno;
                    CVMInt32  rmask;
                    CVMInt16  spillLoc;
                    CVMUint16  localNo; /* If local */
                    CVMInt8   size;   /* size in words of data */
                    CVMInt8  nregs;  /* number of registers occupied
*/
                    CVMInt32  constant; /* If constant */
                    CVMJITIRNode*   expr;
                    struct CVMRMResource* prev;
                    struct CVMRMResource* next;
                };
```

- `dec`: Used by resources that map to IDENTITY nodes.
- `flags`: See Resource Flags section below.
- `regno`: Register number assigned to (allocated for) the resource. -1 if no register assignment has been made.

- `rmask`: Bitmap that represents the registers in use by the resource. Usually `1<<regno, but 1<<(regno+1)` will also be in use for resources that represent 64 bit types when registers are only 32-bit.
- `spillLoc`: Cell number in the temp area where the resource is spilled to. See the Dirty Resources and Spilling section below.
- `localNo`: Local number for resources that map to a local. See Resources for Locals section below.
- `size`: Size in words of the resource. 32-bit and 64-bit resources are supported.
- `nregs`: Number of registers needed for the resource. 32-bit resources always need one register. 64-bit resources will need two registers if the registers only hold 32-bits, and one register if they hold 64-bits.
- `constant`: The constant value if the resources represents a constant. See the Resources for Constants section below.
- `expr`: IR node of evaluated expression for occupied resources. Has no use other than for debugging.
- `prev` and `next`: For keeping resources in a doubly linked list of free or allocated resources.

10.4 Resource Flags

The flags field of the resource indicates the state of the resource. One or more of the following flags can be set:

```
#define CVMRMpinned       (1<<0)
#define CVMRMdirty       (1<<1)
#define CVMRMoccupied      (1<<2)
#define CVMRMjavaStackTopValue (1<<3)
#define CVMRMstackParam     (1<<4)
#define CVMRMphi        (1<<5)
#define CVMRMtrash       (1<<6)
#define CVMRMref        (1<<7)
#define CVMRMclone       (1<<8)
#define CVMRMLocalVar      (1<<9)
#define CVMRMConstant32     (1<<10)
```

- `CVMRMpinned`: The resource has been loaded into a register (specified by `regno`), and this register is pinned, meaning that the Register Manager cannot currently reallocate it to for some other resource.

- `CVMRMdirty`: The resource has been loaded into a register, but currently its value has not been flushed to the backing store. This happens after the evaluation of an intermediate result. See the Dirty Resources and Spilling section for details.
- `CVMRMoccupied`: The resource is occupied by an IR node. This is usually is done by calling `CVMRMoccupyAndUnpinResource()` at the end of each grammar rule that produces a result. The expr field of the resource is set to point to the IR node that occupies the resource. Currently the CVMRMoccupied flag is never checked and the expr field is only used for debugging purposes.
- `CVMRMjavaStackTopValue`: This is used to indicated that the resource corresponds to a method result and is currently on the top of the parameter (Java) stack. More details on parameter handling and `CVMRMjavaStackTopValue` can be found in the Stack Parameter Handling section below and the Stack Manager chapter.
- `CVMRMstackParam`: Although referenced from the source, this flag is no longer needed. See the `CVMSMgetSingle()` and `CVMSMgetDouble()` section of the Stack Manager chapter for more details.
- `CVMRMphi`: Indicates that this resource is for a phi value. See the Phi Handling section below and the Phi Values chapter for more information on phis.
- `CVMRMtrash`: This flag is set when a resource is deleted. A resource should never be used after being deleted, and asserts for this are located throughout the Register Manager source.
- `CVMRMref`: This flag is set whenever the value loaded into a resource is an object reference. This flag is set when the resource is first associated with a value, such as in `CVMRMbindStackTempToResource()`, `CVMRMbindUseToResource()`, `CVMRMloadJavaLocal()`, and `CVMRMoccupyAndUnpinResource()`. It is used so the Register Manager can properly maintain the information needed for generating stack maps. See the JIT Stack Maps chapter for more details.
- `CVMRMclone`: Indicates that the resource is a clone of another resource. It is set by `CVMRMcloneResource()` and `CVMRMcloneResourceSpecific()`. Currently setting this flag serves no purpose.
- `CVMRMLocalVar`: Indicates that the resource is for a local. See the Resources for Locals section below.
- `CVMRMConstant32`: Indicates that the resource is for a constant. See the Resources for Constants section below.

10.5 Register Contexts (Register Banks)

The examples above all pass a `CVMJITRMContext*` argument (usually referred to as rc), to the Register Manager APIs. `CVMJITRMContext` represents a register bank. For most CDC-HI ports there are two such banks: one for GPRs and one for FPRs. For processors that do not having an FPU, only the GPR bank is created and used.

Register Contexts are initialized by calling `RMContextInit()` with a set of arguments describing the register bank. By default the Register Manager `CVMRMinit()` function automatically initializes the GPR and FPR (if used) register banks.

The GPR register context is accessed by using the `CVMRM_INT_REGS(con)` macro and the FPR register context is accessed by using the `CVMRM_FP_REGS(con)` macro.

It is possible to add more register banks if a port wishes to. One possible reason for this is to take advantage of multimedia or DSP registers and instructions available on some modern processors. `CVMRMinit()` could be modified to handle the initialization of any additional register banks.

10.6 Register Sets

When a register context is initialized by calling `RMContextInit()`, the registers are categorized using the following register sets, all of which have corresponding fields in the `CVMJITRMContext`:

- `phiRegSet`: The set of registers available for passing phi values in registers. More details on phis can be found in the Phi Values section below.
- `safeSet`: The set of registers that are non-volatile across C function calls.
- `unsafeSet`: The set of registers that are volatile across C function calls.
- `busySet`: The set of registers known to be busy (such as SP). The Register Manager cannot allocate registers from this set, even if they are included in one of the other sets.
- `anySet`: The set of all registers.

For CDC-HI RISC ports, the proper values for all of these registers sets are determined by RISC porting layer macros provided by the port, such as `CVMCPU_BUSY_SET`, `CVMCPU_NON_VOLATILE_SET`, and `CVMCPU_VOLATILE_SET`.

These register sets are used to help determine preferences for register allocations. They are used in conjunction with preferences specified by the Register Manager client. This is described in more detail in the Register Allocation and Targeting section below.

10.7 Allocating Resources

There are a number of APIs that allocate a resource, or in some cases return an existing resource.

- `CVMRMgetResource()`: Allocates a resource and assigns a register for the resource.
- `CVMRMgetResourceStrict()`: Like `CVMRMgetResource()`, but has stricter requirements for register assignments.
- `CVMRMgetResourceSpecific()`: Like `CVMRMgetResource()`, but requires a specific register.
- `CVMRMcloneResource()`: Clones the resource passed to it and assigns a register to the new resource.
- `CVMRMcloneResourceSpecific()`: Like `CVMRMcloneResource()`, but requires a specific register.
- `CVMRMbindStackTempToResource()`: Used by the Stack Manager to create a resources that represents a method result stored on the top of the Java stack. No register is assigned. See the Stack Parameter Handling for more details.
- `CVMRMbindResourceForConstant32()`: If a resource for the constant exits, then returns that resource. Otherwise a new resource is created and returned. No register is assigned. See the Resources for Constants section below for more details.
- `CVMRMgetResourceForConstant32()`: Like `CVMRMbindResourceForConstant32()`, but also assigns a register for the resource and forces the constant into the register.
- `CVMRMloadJavaLocal()`: Allocates a resource for a local. See the Resources for Locals section below for more details.

Register assignments and the concepts of strict and specific register assignments are explained in more detail in the Register Allocation and Targeting section below.

10.8 Reference Counts and Deleting Resources

The Register Manager maintains a reference count (referred to as a refcount) for each resource. A newly created resource has a refcount of 1. The only way a resource can have a refcount greater than 1 is if the resource is for an IDENTITY node. IDENTITY nodes are explained in more detail in JIT Front End and JCS chapters.

Calling `CVMRMrelinquishResource()` decrements the refcount, and the Register Manager will destroy the resource if the refcount goes to zero. However, resources for locals and constants are never destroyed since they may have subsequent references in the block.

10.9 Register Allocation and Targeting

Some of the APIs that allocate a resource will return a resource already bound to a register. Also, any API that does pinning will also bind the resource to a register. These APIs will always take a target argument, which is a register set of the preferred registers to allocate from. Most also take an avoid argument, which is a set of registers to avoid if the registers in target are not available.

Usually the target and avoid set are only suggestions. However, the strict APIs will require that the allocated register be in the target set and not in the avoid set. The specific APIs are similar to the strict APIs, except they are passed a target register number rather than register set, and there is no avoid set because the requested register number has to be allocated for the resource.

When a register has been allocated for a resource, the `regNo` field of the resource will contain the register number rather than a -1, and also the `rc->occupiedRegisters` bitmap is updated to include the newly allocated register.

If a resource has an assigned register and the resource is not pinned (see Resource Pinning section below), it is possible that the Register Manager will take away the register assigned to the resource. This will happen during register allocation if all registers that meet the targeting requirements of the resource have already been allocated to other resources. In this case one of the other resources will have to give up its register assignment. When a register is taken away from a resource, its `regNo` field is set back to -1.

10.10 ResourcePinning

Pinning is the process of loading a resource into a register. If the regNo field of the resource is -1, this means no register has been allocated for the resource yet, so the first thing done is to allocate a register based on the target and avoid register sets specified when pinning. After this, the APIs that handles the pinning calls `reloadRegister()` to force the loading of the resource from its backing store. The type of resource will determine where it is loaded from. See the Dirty Resources and Spilling, Resources for Constants, Resources for Locals, and Phi Handling sections for more details on resource types their backing store.

If the resource being pinned already has a register assigned to it, then the resource is assumed to already be loaded into that register. In this case no action is needed, assuming that the register meets the targeting requirements specified when pinning. If not, a new register is allocated that does meet the targeting requirements, and the old register is moved to the new register and then released.

Pinning always results in setting the `CVMRMpinned` flag. This prevents `regNo` from being reallocated for use by another resource. For this reason, resources are usually only pinned for a very short period of time. Generally resources are only left pinned within the semantic action of a grammar rule, but not across rules. When a resource is unpinned, it will still occupy the register, but the register is also made available for reallocation for another resource.

- `CVMRMpinResource()`: Forces the resource to be loaded into a register.
- `CVMRMpinResourceStrict()`: Like `CVMRMpinResource()`, but forces the resource into a register in the target set.
- `CVMRMpinResourceSpecific()`: Like `CVMRMpinResource()`, but forces the resource into the specified register.
- `CVMRMunpinResource()`: Unpins the resource. This clears the `CVMRMpinned` flag. The resource will still be considered loaded into `regNo`, but `regNo` will made available for reallocation.
- `CVMRMoccupyAndUnpinResource()`: Like `CVMRMunpinResource()`, but also occupies the resource with the specified IR node. Occupying a resource no longer serves any purpose in CDC-HI, except in some Register Manager debugging code.

10.11 Dirty Resources and Spilling

Dirty resources are those that contain a value in a register, and the backing store for the resource is out of date with respect to the value in the register. For example, in the IADD32 rule above, after computing the result of the add into the dest resource, the resource is considered dirty and will have its `CVMRMdirty` flag set. If the register with the result were to be taken for use by another resource, then the result would be lost forever.

The process of updating a dirty resource's backing store with the contents of the resource's register is called spilling. A slot in the spill area (also called the temp area) is set aside for resources that need to be spilled. Whenever a dirty resource's register has to be vacated for any reason (such as for use by another resource), its contents are first stored to its slot in the spill area. The slot number is stored in the resources spillLoc field.

As mentioned above, some spills happen implicitly as registers are reallocated for other uses. However, some are also explicit. For example, when making a method call, all registers allocated by the Register Manager are reused by the callee and are not restored before returning. For this reason, when making a method call all dirty resources need to be spilled first.

There are two APIs for handling this forced spilling of registers:

- `CVMRMminorSpill()`: Used when calling C helper functions that won't cause a GC. All dirty resources in volatile registers (as defined by C calling conventions) are spilled to the temp area. In addition to this, all resources in volatile registers are vacated from their registers (the registers are freed).
- `CVMRMmajorSpill()`: Used when calling anything that might result in a GC, including method calls and C helpers such as allocators. Resources containing object references are always spilled if dirty and also vacated. Also, resources not in the register set specified by the safe argument are spilled if dirty and also vacated. Usually the safe argument is either `CVMRM_EMPTY_SET` for things like method calls that trash all registers, or `CVMRM_SAFE_SET` for calls to C helper functions that preserve non-volatile registers.

10.12 Resources for Constants

The register manager has the ability to track resources used for constants, so for any given constant there is only one resource created for it, even though the constant might be referenced by multiple ICONST_32 IR nodes. This allows a constant already loaded into a register to be reused without having to reload register.

As already described in the Allocating Resources section, `CVMRMbindResourceForConstant32()` and `CVMRMgetResourceForConstant32()` are used for allocating resources associated with constants.

A resource associated with a constant will have its `CVMRMConstant32` flag set, and will have its constant field set to the value of the constant. For this reason, a resource for a constant is never considered to be dirty because code to reload the constant can be generated at any point.

When a constant needs to be loaded into a register, `reloadConstant()` is called. It will in turn call `CVMCPUemitLoadConstant()`, which may rely on the Constant Pool Manager to track where in memory the constant is stored. See the Constant Pool Manager chapter contains more details on this.

10.13 Resources for Locals

In the CDC-HI RISC ports, resources allocated for locals are always up to date and never have their `CVMRMdirty` flag set. This is because locals are write through, and is the result of assignment rules calling `CVMRMstoreJavaLocal()`, which does the following:

1. Stores the specified pinned resource to the backing store for the local.
2. Sets the `CVMRMLocalVar` flag on the resource.
3. Sets the `localNo` field of the resource to the local number.
4. Adds the resource to `rc->local2res[]`, which maps local numbers to resources

When a local is referenced in a grammar rule, such as in the LOCAL32 rule shown in the example above, `CVMRMloadJavaLocal()` is called. It will first check if a resource for the local already exists in `rc->local2res[]`, in which case it can be reused, saving a load if the local is already in a register. Otherwise a new resource is created for the local, and like with `CVMRMstoreJavaLocal()`, the `CVMRMLocalVar` flag is set, the `localNo` field is set, and the resource is added to `rc->local2res[]`.

10.14 Block Handling

Generally speaking, the Register Manager state is not carried across basic blocks. The one exception to this are Phis, which are described in the Phi Handling section below.

- `CVMRMbeginBlock()`: Called on block entry to initialize a few register set related fields and to reset a few data structures, such as `rc->local2res[]`. It also handles the binding of USED nodes to resources, which is described in more detail in the Phi Handling section.
- `CVMRMendBlock()`: Called on block exit. The main purpose of `CVMRMendBlock()` is to free all resources.

10.15 Phi Handling

Phis are values that are left on the semantic stack across blocks. They are discussed in detail in the Phi Values chapter. The following Register Manager APIs exist to support phi handling:

- `CVMRMspillPhis()`: Only used in the rare case where a backwards branch target has incoming phis. If any phis are for object references or are stored in volatile registers, they need to be spilled in order to avoid register corruption if a GC is requested. See the GC Checks in Compiled Code chapter for more details on GC handling at backwards branch targets.
- `CVMRMreloadPhis()`: Reloads the phis spilled by `CVMRMspillPhis()` after the GC is complete.
- `CVMRMbindAllUsedNodes()`: Called on block entry. For each incoming phi, binds the USED IR node for the phi to a newly created resource. The resource will either map the phi to a register or to a cell in the temp area, depending on how it was passed.
- `CVMRMstoreDefinedValue()`: Stores an outgoing phi into its cell in the temp area if the phi is not being passed in a register.
- `CVMRMloadOrReleasePhis()`: The first call to `CVMRMloadOrReleasePhis()` loads all phi resources being passed in registers into their proper registers and leaves the resources pinned so the registers will not be reallocated during handling of a conditional branch instructions. The second call unpins the phi resources after the branch is emitted.

10.16 Method Result Handling

When a resource that maps to a method result is pinned, the Register Manager pops the result off of the Java stack and into a register. This is handled in `reloadRegister()` by calling `CVMSMpopSingle()` or `CVMSMpopDouble()`.

This is only done for resources created by calling `CVMRMbindStackTempToResource()`, which is called by the Stack Manager `CVMSMinvocation()` function, which is called by INVOKE rules in the grammar.

The following two rules demonstrate result handling in the grammar:

```
                 invoke32_result: INVOKE32 parameters reg32 : 40 : :
: : {
                     CVMRMResource* dest = invokeMethod(con, $$);
                     pushResource(con, dest);
                 };
                 reg32:invoke32_result: 20 : : : : {
                     /* force into a register */
                     CVMJITRMContext *rc = CVMRM_INT_REGS(con);
                     CVMRMResource* operand = popResource(con);
                     CVMRMpinResource(rc, operand,
GET_REGISTER_GOALS);
                     CVMRMunpinResource(rc, operand);
                     pushResource(con, operand);
                 };
```

The first rule is responsible for emitting the invocation. This is handled by `invokeMethod()`, which will make the call to `CVMSMinvocation()`, which creates the resource that is eventually returned by `invokeMethod()`. This resource is pushed onto the grammar semantic stack, and is later consumed by the second rule above when the method result is needed in a register. The second rule pops the resource that was pushed by the first rule and then pins it, which pops the result off the Java stack and into a register. As mentioned above, this is handled in `reloadRegister()` by calling `CVMSMpopSingle()`.

See the Stack Manager chapter for more details on the interaction of the Register Manager and the Stack Manager.

10.17 Porting Effort

For ports done within the JIT RISC porting layer, the only porting effort needed is providing values for the macros used by the Register Manager that are also part of the RISC porting layer. These are specified in the `jitrisc_cpu.h` header file.

For ports done outside of the JIT RISC porting layer, functionality similar to the Register Manager is probably necessary, although a port is free to do this as it sees fit. Unfortunately the Register Manager is not completely abstracted from shared JIT code.

The first abstraction problem is that the `CVMJITCompilationContext` struct embeds the *CVMJITRMCommonContext* and `CVMJITRMContext` structs, so this implies the presence of some sort of Register Manager. To make matters worse, the header file that describes these structs, `jitrmcontext.h`, is located in shared code. This header file should be moved to `src/portlibs/jit/risc/include/export` so a port is free to define `CVMJITRMCommonContext` and `CVMJITRMContext` any way it wishes.

The second abstraction problem is that the following fields in `con->RMcommonContext` are all accessed from `jitstackmap.c`, which is shared code:

- `maxSpillNumber`
- `spillRefSet`
- `spillBusySet`

These fields should probably all be moved to `CVMJITCompilationContext`.

10.18 Reference

Primary Source Files:

- `jitregman.c`
- `jitregman.h`
- `jitrmcontext.h`
- `jitgrammarrules.jcs`

Data Structures:

- `CVMJITRMCommonContext`
- `CVMJITRMContext`

CHAPTER 11

Stack Manager

11.1 Overview

The Stack Manager provides compile time management of method parameters pushed on to the Java expression stack. It is part of the shared RISC implementation, and is used by the backend in conjunction with the Register Manager to track values on the Java stack that are to be used as parameters.

The following fields of `CVMJITCompilationContext` are used by the Stack Manager:

```
                    CVMInt32SMdepth;
                    CVMJITSetSMstackRefSet;
#ifndef CVMCPU_HAS_POSTINCREMENT_STORE
                    CVMInt32SMjspOffset;
#endif
```

When the backend needs to push a parameter onto the Java stack, it calls `CVMSMpushSingle()` or `CVMSMpushDouble()`. These calls are normally triggered by PARAMETER32 and PARAMETER64 IR nodes. These functions will update the current depth of the stack (`SMdepth`), make sure the resource representing the parameter is loaded into a register by calling `CVMRMpinResource()`, and then call an emitter to store the parameter to the Java stack.

When emitting code for a method call, first `CVMSMpopParameters()` is called. It simply adjusts `SMdepth` by the number of parameters. Next, `CVMSMinvocation()` is called. It creates a resource to represent the method result, binds the resource to the current Java top of stack, and sets the resource's `CVMRMjavaStackTopValue` flag. This is known as a deferred pop and is done to optimize for method results that are to be used as parameters, which is described in more detail below.

11.2 Handling Method Results

Method results are consumed by the grammar using one of the following two JCS rules:

```
                 // Purpose: Stores a 32 return value into a
register.
                 reg32:invoke32_result: 20 : : : : {
                     /* force into a register */
                     CVMRMResource* operand = popResource(con);
                     CVMRMpinResource(CVMRM_INT_REGS(con), operand,
GET_REGISTER_GOALS);
                     CVMRMunpinResource(CVMRM_INT_REGS(con),
operand);
                     pushResource(con, operand);
                 };
                 param32: invoke32_result : 0 : : : : {
                     /* Free! Already on Stack */
                     CVMRMResource *operand = popResource(con);
                     CVMRMrelinquishResource(CVMRM_INT_REGS(con),
operand);
                 };
```

If a method result is not to be used as a parameter, then the first rule is used. The call `CVMRMpinResource()` will result in the `invoke32_result` value being popped from the Java stack and loaded it into a register. This is handled in `jitregman.c` by the following code in `reloadRegister()`:

```
                 } else if (CVMRMisJavaStackTopValue(rp)) {
                     /* this is a deferred pop operation. */
                     if (rp->size == 1){
                         CVMSMpopSingle(con, rp);
                     } else {
                         CVMSMpopDouble(con, rp);
                     }
                     CVMRMclearJavaStackTopValue(rp); /* not on top
of the stack */
                     rp->flags |= CVMRMdirty;
  } else ...
```

Note that the value is marked as "dirty," indicating that it is in a register, but is no longer associated with an up-to-date memory location.

If a method result is to be used as a parameter to another method call, rather than immediately popping the method result into a register, it is simply left on the stack. This is handled by the "`param32: invoke32_result`" rule above. Since this rule has a cost of 0 and the combination of the "`param32: reg32`" and "`reg32: invoke32_result`" rules has a cost of 20, the "`param32: invoke32_result`" rule is chosen if the parameter is a method result. This way the method result is not popped by `CVMSMpopSingle()` and then pushed again by `CVMSMpushSingle()`. Since the result of the invocation is already on the top of the stack, there is no need to push it on the stack again to be used as a parameter.

11.3 CVMSMadjustJSP()

Normally the Stack Manager relies on the platform specific emitters to handle the adjustment of the stack pointer (JSP register) as parameters are pushed to the Java stack. This works well if the platform supports post increment store instructions that can automatically update JSP as part of the store instruction. If this is supported, then the platform will #define `CVMCPU_HAS_POSTINCREMENT_STORE`, and the JSP register is automatically updated as parameters are pushed.

If post increment stores are not supported, then normally the platform would be left with doing a manual adjustment of JSP every time an argument is pushed. Instead of this, the Stack Manager defers any JSP adjustment until necessary. In `con->SMjspOffset` it tracks the number of words pushed since the last adjustment. When an adjustment is required (which is determined the client of the Stack Manager), `CVMSMadjustJSP()` is called to emit code to adjust the JSP registers, and `con->SMjspOffset` is reset to `0`.

11.4 Stack Maps

When `CVMSMpushSingle()` is used to push an object reference onto the Java stack, then `con->SMstackRefSet` is updated to reflect this. Periodically `SMstackRefSet` is grabbed by the Stack Manager `CVMJITcaptureStackmap()` function. This is done so a stackmap can be generated that accurately reflects object references on the Java stack.

SMstackRefSet is also updated when CVMSMinvocation() is called to reflect the refness of the returned result being pushed to the Java stack. Note that only the parts of SMstackRefSet covered by SMdepth are ever used, so there is no need to update the refness of parameters popped by CVMSMpopParameters().

11.5 Handling Non-parameter Stack Values

Only parameters are pushed to the Java stack in compiled methods. Other expressions that would normally end up on the Java stack when interpreted are either maintained in registers or stored to the temp area (spill area) of the compiled Java frame. For example:

```
x = (val1 * val2) + foo(arg1, arg2);
```

First val*val2 is evaluated, and for the CDC-HI RISC ports, it would end up in a register. Next arg1 is loaded into a register. Since it is a parameter, this is handled by CVMSMpushSingle() when it calls CVMRMpinResource(), unless arg1 happens to already be in a register. CVMSMpushSingle() will then emit code to store arg1 to the Java stack. The same is done for arg2. Next the invocation as handled. Since the register containing val1*val2 will be lost during the invocation, the Register Manager will spill the register to the temp area of the compiled frame. After the invocation, val1*val2 is reloaded from the temp area into a register, and can be added to the result of the invocation to provide the value to assign to x. At no point was val1*val2 tracked by the Stack Manager since val1*val2 is considered a temporary value and not a parameter.

11.6 CVMSMgetSingle() and CVMSMgetDouble()

Although referenced in the source code, CVMSMgetSingle() and CVMSMgetDouble() are no longer ever called at runtime and will be removed from CDC 1.1. This is also true of all other code related to CVMSMgetSingle() and CVMSMgetDouble(), including code in reloadRegister() that calls them, CVMRMconvertJavaStackTopValue2StackParam() (plus the call to it), rp->stackLoc, and the CVMRMstackParam flag.

11.7 Porting Effort

The Stack Manager is shared RISC code, so no porting is needed when the processor port is done within the shared RISC porting layer. Otherwise the Stack Manager will need to be replaced with an implementation that still properly maintains `SMstackRefSet`. This is required so the `CVMJITcaptureStackmap()` in `jitstackmap.c` functions properly.

11.8 Reference

Primary Source Files:

- `jitstackman.h`
- `jitstackman.c`

CHAPTER **12**

Phi Values

12.1 Overview

Sometimes the Java stack is not empty when exiting or entering a basic block. The most common example is a selection expression. For example:

```
int x = b ? val1 : val2;
```

```
is semantically equivalent to:
```

```
if (b) {
    push(val1);
} else {
    push(val2);
}
x = pop();
```

The Java bytecodes will look as follows:

```
L1  iload b
    ifeq L2
    iload val1
    goto L3
L2  iload val2
L3  istore x
```

L0, L1, and L2 each start a new basic block. On exit from block L1 (the `goto` bytecode), `val1` is on the Java stack. On exit from L2 (fall through to L3), `val2` is on the Java stack. Block L3 pops the value off the stack and stores it into the local x.

Values left on the Java stack on block exit are referred to as Phi values. Phi values require special handling by both the front end and backend of the JIT. Upon exit from a block, whether the result of a branch (possibly conditional) or fall through, code must be generated to make sure all phi values are flushed to known locations in the compiled frame so the target block can load them when referenced. Implementations can also instead choose to move the phi values well-known registers that carry phi values across blocks. These are known as "register phis" and will be discussed in more detail below.

Java semantics requires that if there are any phis on the stack when branching to another block, all branches to the same block must have the same number and type of phi values, and the target block must be expecting these phi values. Thus in our example above, upon seeing the "`goto L3`" with one phi value on the stack, we can conclude that L3 will expect one phi value, and any other blocks that flow to L3 will also have one phi value. Otherwise the method would have failed bytecode verification. This fact is important in the front end because it needs to know the number and type of all phi values entering a block in order to properly parse the block. Since IR parsing is done in flow order, when the front end begins to parse a block we are guaranteed to have already parsed at least one branch or fall through to the block, thus the number of incoming phi values is always known before a block is parsed.

12.2 Passing Phi Values Between Blocks

Phi values are passed between blocks either in phi registers or in a part of the stack frame referred to as the "spill area", which is used to spill values that cannot be maintained in a register. The "spill area" is also referred to as the "temp area", and is located just above the Java compiled frame (`CVMCompiledFrame` struct) and below the Java expression stack.

The current CDC-HI ports use both phi registers and the spill area depending on the number and type of the phi values. A port could choose just to use the spill area. Choosing to use just phi registers has its limitations because this limits the number of phis that are supported, requiring failure to compile some methods. The front end provides hints for the storage of phi values, but ultimately the decision is up to the backend.

Note that the source block and target blocks must agree on where each phi value is stored. The source block cannot just choose to store the phi value in any available spill location or register. It must be in the location agreed upon with the target block. Usually the stack depth of the phi value is used to determine how it is passed, but the backend is free to make this decision itself.

12.3 DEFINE Nodes and USED Nodes

A DEFINE node must be created for each phi expression when a block exit is detected (branch or fall through). The DEFINE node is a root node that is responsible for forcing the evaluation of the phi expression, and also forcing it either into the proper frame location (referred to as “spilling”) or the proper phi register. Where the phi value is stored is done in agreement with where the target block expects to find the phi value.

DEFINE nodes are represented by the `CVMJITDefineOp`:

```
                 typedef struct {
                     CVMJITIRNode* operand; /* stack item for this
define node */
                     CVMJITIRNode* usedNode; /* USED node for this
define node */
                     CVMRMResource* resource; /* resource into which
it is stored */
                 } CVMJITDefineOp;
```

- `operand` is the IR expression tree that will evaluate into the phi value.
- `usedNode` is the corresponding USED node in the `target` block (see below).
- `resource` is for use by the backend to track which Register Manager resource the DEFINE operand node is represented by.

A USED node must be created for each phi value that flows into a block. USED nodes are responsible for loading a spilled phi into a register for use, or in the case of register phis, binding the USED node to a resource that specifies the register the phi is already loaded into.

USED nodes are represented by the `CVMJITUsedOp`:

```
                 typedef struct {
                     CVMInt16 spillLocation;  /* location this value
spills to */
                     CVMUint8 registerSpillOk; /* true if ok to spill
to a register */
                     CVMRMResource* resource; /* resource into which
it is stored */
                 } CVMJITUsedOp;
```

- `spillLocation` is the offset in words of the phi value from the bottom of the stack. It is computed by the front end when the IR is created, and is used by the backend to determine where in the compiled frame the phi value should be loaded from (or which register to load it into). It is called `spillLocation` because any DEFINE node referring to the USED node also uses it to determine where in the frame the value should be spilled to so it can then be loaded by the corresponding USED node.

- `registerSpillOk` is set true if it is OK to maintain the phi value in a register across blocks rather than spill it to the compiled frame. It is always set true for the current CDC-HI ports, except that if `-Xjit:registerPhis=false` is specified on the command line. Note that `registerSpillOk` is only advisory. It does require that that phi value be passed between blocks in a register. The backend gets to make this choice.
- `resource` is for use by the backend to track which Register Manager resource the USED node is represented by.

USED nodes are maintained in an array attached to each block with incoming phis. The following fields in `CVMJITIRBlock` are used to track incoming phis:

```
                 CVMJITIRNode** phiArray; /* array of USED nodes */
                 CVMInt16   phiCount; /* number of phiArray items */
                 CVMUint16   phiSize; /* size in words of all phi
items */
```

12.4 CVMJITirblockPhiMerge()

The first task in handling phi values is to setup the relevant data structures in the IR. The front end handles this every time it detects a branch or fall through out of a block. `CVMJITirblockAtBranch()` will be called in this case, and it will handle phi values by calling `CVMJITirblockPhiMerge()`, which is in charge of creating all data structures related to passing phi values between blocks.

If this is the first time a branch to the target block has been encountered, `CVMJITIRBlock.phiArray` is created for the target block and is populated with USED nodes that are also created. `CVMJITIRBlock.phiCount` is set to the number of incoming phis and `CVMJITIRBlock.phiSize` is set to the total size in words of all phi values.

In the source block, a DEFINE node is created for each phi value and a root node is created for each DEFINE node to force evaluation of the DEFINE node.

In order to support passing phis in registers, a LOAD_PHIS node is created, and it points to an array containing pointers to all the DEFINE nodes. LOAD_PHIS is used by the backend to make sure that after all expressions involved in the passing of phi values (including expressions used in conditional branches) have been evaluated, all phis are loaded into the proper registers. The registers remain pinned until after the branch is emitted to ensure that arguments used in conditional branches are not loaded into registers used by the phi values.

Next a node to emit the branch is created. After this a RELEASE_PHIS node is created to give the backend a chance to unpin all the phis that are currently in registers.

12.5 `CVMJITirblockAtLabelEntry()`

`CVMJITirblockAtLabelEntry()` is called by the front end to do some prep work for the block just before the block is parsed. One of the tasks is to prime the simulated Java semantic stack with all incoming phi values. This is done by pushing each USED node in the block's `phiArray` on to the simulated Java semantic stack. As the bytecodes are parsed, these phi values are popped when referenced by bytecodes, and made part of the IR tree.

12.6 Unsupported Phi Constructs

If there are any phis on the Java stack when a ret instruction is encountered, compilation of the method is `refused.ret` is used to return from a finally block, and javac never produces phis in this case. Refusing compilation of such methods simplifies phi handling. A CDC-HI implementation is free to reject methods with Phis for other reasons if it wishes.

12.7 Virtual Method Inlining

Inlining of virtual methods also results in phi values being passed between blocks. Inlining of virtual methods is handled by inlining a best guess method. Since this guess might be wrong, first a runtime check needs to be made to make sure the guess is correct and do a virtual method call if it is wrong.

Checking if the guess is correct results in creating a new basic blocks that are not apparent when just observing the Java byte codes. A virtual method call usually looks something like the following:

```
push arguments
```

```
invoke virtual method
```

With the introduction of inlining a guess method, the semantics of an inlined virtual method call will look something like the following:

```
push arguments
if (guess method == vtable method) {
    <inlined method>
} else {
    invoke virtual method
}
```

The single basic block in the original example has expanded into three, and the `else` clause is entered with phi values on the stack. Note that for the `if` clause, the stack arguments are not considered phi values since a new basic block does not start after the conditional branch to the else clause. Since virtual inlining like this is so common, most phi values are due to virtual inlining and selection expressions.

12.8 Register Phis

As mentioned above, CDC-HI will attempt to keep most phis in registers. The RISC porting layer defines some macros for letting shared RISC code determine which phis to pass in registers. The key macro is `CVMCPU_PHI_REG_SET`, which is the set of registers to be used for passing phi values between blocks. Below is an example from the ARM port.

```
 #define ARM_PHI_REG_1(1U << CVMARM_v3)
 #define ARM_PHI_REG_2(1U << CVMARM_v4)
 #define ARM_PHI_REG_3(1U << CVMARM_v5)
 #define ARM_PHI_REG_4(1U << CVMARM_v6)
 #define ARM_PHI_REG_5(1U << CVMARM_v7)
 #define ARM_PHI_REG_6(1U << CVMARM_v8)

 #define CVMCPU_PHI_REG_SET (\
                 ARM_PHI_REG_1 | ARM_PHI_REG_2 |\
                 ARM_PHI_REG_3 | ARM_PHI_REG_4 |\
                 ARM_PHI_REG_5 | ARM_PHI_REG_6\
)
```

12.9 Phi Handling in the Backend

In the backend for the shared RISC ports, the Register Manager is responsible for most of the phi handling.

`CVMRMstoreDefinedValue()` is called by the grammar when a DEFINE node is encountered to make sure the evaluated phi expression is moved or loaded into the proper register (if being passed in a register) or stored to the proper spill location.

`CVMRMbindAllUsedNodes()` is called at the start of each basic block to bind incoming phi values to resources that either represent a register (if the phi was passed in a register) or a spill location. It does this by creating a resource for each USED node in the block's phiArray and binding the resource to either a spill location or a register, depending on how the phi value is suppose to be passed to the block.

Phi handling is simpler if phi values are always passed in the spill area, but is less optimal than passing in phi registers. Passing phis in registers adds some complications, some of which is forced into the front end to make sure DEFINE nodes and conditional branch expressions are all evaluated before attempting to load phi values into registers. The reader should review the source for `CVMJITirblockPhiMerge()` and in `jitregman.c` for more details.

12.10 Reference

Primary Source Files:

`jitir.h`

`jitir.c`

`jitirblock.h`

`jitirblock.c`

`jitregman.c`

`-Xjit` Command Line Options:

`xregisterPhis`: allow or disallow passing phis in registers

CHAPTER **13**

Trap-based NullPointerExceptions

13.1 Overview

The implementations of many Java bytecodes, such as `opc_invokevirtual`, need to first check if an object reference is null, and throw a `NullPointerException` if it is. Dynamically compiled methods must also do the equivalent of a null object check. This leads to slower performance and an increase in generated code. The following is an example of PowerPC code emitted for a null object check on an array object. In this example the check is needed before accessing the `arrayLength` field of the object.

```
cmpi  r25, 0 @ NULL check
beql-  CVMCCMruntimeThrowNullPointerExceptionGlue
lwz   r23, 8(r25)   @ arraylength
```

The CDC-HI dynamic compiler allows for a more lazy approach to check for null object references. This approach is referred to as trap-based `NullPointerExceptions`. It eliminates doing the above explicit check for a null object reference, followed by a conditional branch to throw the exception. Instead the dereference of the null object reference is allowed to cause a crash. This results in a SIGSEGV on most POSIX platforms. A signal handler must be installed to catch this signal, confirm that it occurred in compiled code, and cause execution to resume in code that will throw a `NullPointerException`.

CDC-HI ports usually implement the SIGSEGV signal handler in `jit_arch.c`. The `handleSegv()` function catches the signal, changes the link register to the instruction after the crash occurred, and changes the pc to point to the

CVMCCMruntimeThrowNullPointerExceptionGlue() routine. This exactly mimics the compiled code calling CVMCCMruntimeThrowNullPointerExceptionGlue() itself from the point of the crash, which is the desired behavior when using a null object references. The implementation of the signal handler will vary for every processor and every OS. An example from the Linux/PowerPC port is provided below.

```
static void handleSegv(int sig, siginfo_t* info, struct ucontext*
ucp)
{
                 CVMUint8* pc = (CVMUint8 *)ucp->uc_mcontext.regs-
>nip;
                 if (CVMJITcodeCacheInCompiledMethod(pc)) {
                     /* Coming from compiled code. */
                     /* Branch and link to throw null pointer
exception glue */
                     ucp->uc_mcontext.regs->link = ucp-
>uc_mcontext.regs->nip + 4;
                     ucp->uc_mcontext.regs->nip =
                         (unsigned
long)CVMCCMruntimeThrowNullPointerExceptionGlue;
                 }
}
```

The implementation of trap-based NullPointerExceptions is a performance optimization that is entirely optional. A CDC-HI port can choose to implement it to increase performance. To disable it, the port must #undef CVMJIT_TRAP_BASED_NULL_CHECKS. CDC-HI 1.0.1 linux ports will also require changes to handleSegv() to avoid treating crashes as NullPointerExceptions. This problem is fixed in CDC-HI 1.1. Once the port is working, the port can choose to enable trap-based NullPointerExceptions and implement the signal handler for it.

One downside of trap-based NullPointerExceptions is that if there is a crash in compiled code due to a bug in the dynamic compiler, the result is a NullPointerException rather than a crash, which can make it difficult to track down. The best way to handle this is to first do a lot of testing with trap-based NullPointerExceptions disbabled, and also disable this feature whenever investigating bugs.

13.2 Reference

Primary Source Files:

`jit_arch.h`

`jit_arch.c`

Macros:

`CVMJIT_TRAP_BASED_NULL_CHECKS`

APIs:

`CVMCCMruntimeThrowNullPointerExceptionGlue()`

CHAPTER **14**

GC Checks in Compiled Code

When a thread needs to do a GC, usually because the heap is full, it first sets the `CVMglobals.cstate[CVM_GC_SAFE].request` flag. It then blocks until all threads that are currently GC-unsafe have checked this flag, and as a consequence have become GC-safe and called `CVMD_gcRendezvous()` to notify the blocking thread. Once all threads become GC-safe, the thread requesting the GC can start doing the GC.

Polling for GC requests must be done on every backwards branch. This is true of both interpreted code and compiled code. This is necessary to ensure that a thread does not sit in a loop, preventing other threads that need to GC from making any progress. GC safety within the Virtual Machine is covered in more detail in first few pages of the Creating a Garbage Collector section of the *CDC Porting Guide*.

In CDC-HI RISC ports, the code responsible for emitting the polling instructions is done by `CVMJITcheckGC()` in `jitgrammarrules.jcs`. It is called by the Register Manager `CVMRMbeginBlock()` function at the start of each block that is a backwards branch target.

14.1 Explicit GC Checks

Polling for a GC request in compiled code can be done with an explicit check of the `CVMglobals.cstate[CVM_GC_SAFE].request` flag before emitting code for a block that is a backwards branch target. If it is set, then the thread must call `CVMCCMruntimeGCRendezvous()`, a C helper function in ccm_runtime.c. Usually rather than generating code to handle calling `CVMCCMruntimeGCRendezvous()`, simpler code is generated to call the platform specific assembler function `CVMCCMruntimeGCRendezvousGlue()`, which handles the C calling convention issues. This simplifies the generated code.

Below is an ARM example of a generated GC rendezvous check:

```
                 @ Do GC Check:
                 ldr   v8, =CVMglobals
                 ldr   v8, [v8, #+24] @
CVMglobals.cstate[CVM_GC_SAFE].request;
                 cmp   v8, #0     @ If GC is requested,
                 blne  PC=(-8552)   @ call
CVMCCMruntimeGCRendezvousGlue
```

Some ports may choose to keep `CVMglobals` in a register, which reduces the overhead of the GC request check.

Note – There is no built in support for generating the above inlined GC check in CDC-HI 1.0.1, although it could be easily added. Instead only patch-based GC checks (described below) are supported. Explicit GC checks are supported in CDC-HI 1.1 if neither `CVMJIT_PATCH_BASED_GC_CHECKS` or `CVMJIT_TRAP_BASED_GC_CHECKS` are defined.

14.2 Patch-based GC Checks

Doing an explicit GC check at the start of every loop iteration as described above can produce a lot of overhead. In many loop intensive benchmarks, this adds as much as 20% overhead. For CDC-HI a mechanism called Patch-based GC Checks is implemented. Rather than generating explicit inline checks, instead when a GC request is made the first instruction of the backward branch target is patched over with an explicit call to `CVMCCMruntimeGCRendezvousGlue()`. This allows for a zero overhead GC check.

The patching mechanism is quite complex, and is made even more complex on processors that can't handle the call to `CVMCCMruntimeGCRendezvousGlue()` with one instruction, including all processors with delay slots. This is because only one instruction can be atomically patched at a time, and after patching in the call instruction, you don't want the instruction in the delay slot to be executed. The other complicating factor for patch-based GC checks is that data structures need to be maintained to keep track of which instructions need to get patched and what they should be patched.

Usually the generation of code at the patch site is done as follows. First the explicit call to `CVMCCMruntimeGCRendezvousGlue()` is generated. Next the code buffer is rewound back to the start of the call instruction by using `CVMJITcbufRewind()`.

Next this call instruction is stored away in a location that can be accessed when a thread requests a GC. After this code generation resumes, and the call instruction is overwritten by whatever code is first emitted for the block.

The following example is from PowerPC. The example shows the code generation for a block that is a backwards branch target, and the first thing the block does is call the method `Foo.bar()`. Note how after the call to `CVMCCMruntimeGCRendezvous()` at offset 88 is generated, the code buffer is rewound back to 88, and the first instruction of the method invocation code is generated there.

```
                          @ Patchable GC Check point
                  88:     bl   PC=(-5168)   @
CVMCCMruntimeGCRendezvousGlue
                          @ Invoke a method w/ a 32bit return type
                  88:     lwz   r3, 0(rCP)   @ mb Foo.bar()LFoo;
                  92:     lwz   r0, 0(r3)    @ call method through mb
                  96:     mtlr  r0
                  100:blrl
```

At runtime when a GC is requested, the entire code cache is walked, and all the calls to `CVMCCMruntimeGCRendezvousGlue()` are patched back in. When an executing compiled method hits this branch, it causes the thread to become GC-safe and do a GC rendezvous. After the GC is complete, the branches are “unpatched” back to the original instructions.

In CDC 1.0.1, the patching and unpatching are handled at GC time by `CVMJITcsPatchRendezvousCalls()` and `CVMJITcsUnpatchRendezvousCalls()` in jit_risc.c. In CDC 1.1, the patching and unpatching is handled at GC time by `CVMJITenableRendezvousCalls()` and `CVMJITdisableRendezvousCalls()`, also in `jit_risc.c`. Platforms not implemented using the RISC porting layer will need to provide their own versions of these functions.

Before doing the patching, if GC detects that all threads are already GC safe, for efficiency reasons, it does not bother doing any of the patching. This is possible because GC safe threads are already prevented from becoming GC-unsafe, so there’s no way that the patched instructions could ever be executed anyway.

14.3 Patch-based GC Checks with Delay Slots

As mention above, it is difficult to deal with patch-based GC checks if the call to `CVMCCMruntimeGCRendezvousGlue()` takes more than one instruction. This section describes how this problem is solved when there is a delay slot, although in general it applies whenever it take more than one instruction.

If there is a delay slot, it is necessary to emit a nop instruction after the call to `CVMCCMruntimeGCRendezvousGlue()` in order to avoid execution of an unwanted instruction in the delay slot. The introduction of the nop means that two instructions need to get patched, and this isn't possible to do in an atomic fashion. Because of this, it is not possible to rewind the code buffer back to the call instruction and then overwrite both the call instruction and the nop with the first instructions of the block.

The solution is to explicitly overwrite the call instruction with a nop, leaving two nop instructions at the start of the block, and the code buffer is left pointing after these two nop instructions rather than overwriting the call instruction. Although this nop solution works, it now means that we always execute two nop instructions at the start of every backwards branch target. Although faster then doing an explicit GC check every time, it is no longer zero cost.

The solution to the overhead of the nop instructions is to make branches to backwards branch targets branch to after the nop instructions. When a GC is requested, these branches are patched to branch to the start of the backwards branch target where the two nop instructions are. Since the first nop is patched to branch to `CVMCCMruntimeGCRendezvousGlue()`, the desired GC check is made. The generation of this backwards branch code is done in `branchToBlock()` in `jitgrammarrules.jcs`.

14.4 Trap-based GC Checks

As mentioned above, patch-based GC checks can be difficult to manage, especially if it takes multiple instructions to call `CVMCCMruntimeGCRendezvousGlue()`. Another downside is that the patching results in writing into the code cache, which means it is not possible to have a read only code cache, a feature that CDC 1.1 will take advantage of for better memory efficiency.

CDC 1.1 introduces trap-based GC checks. They are enabled with `#define CVMJIT_TRAP_BASED_GC_CHECKS`. When trap-based GC checks are enabled, each backwards branch target will have one instruction emitted at the start of the block. This instruction is normally benign, but can be made to trap when a GC is requested. The following example is from the ARM port

```
 ldr    rGC, [rGC, #+0] @ gc trap instruction
```

In this case, rGC is a dedicated register that is preloaded with the address of a page in memory that is normally readable. All threads will load rGC with the same value when entering compiled code. The first word in this page points back to itself, causing the above instruction to reload rGC with itself, thus it is benign.

The value loaded into *rGC* is `CVMglobals.gcTrapAddr`. `CVMglobals.gcTrapAddr` is setup by shared code in `globals.c`, but the loading of `rGC` needs to be setup by platform specific code, usually in `CVMJITgoNative()`.

When a GC is requested, the page of memory pointed to be `CVMglobals.gcTrapAddr` is protected to prevent read access. On Linux this is done with the `mprotect()` function. The platform specific `CVMJITenableRendezvousCalls()` function is responsible for doing this, and is usually implemented in `jit_md.c`. When GC is complete, `CVMJITdisableRendezvousCalls()` is called to allow the platform to make the page readable again.

Protecting the page pointed to by `CVMglobals.gcTrapAddr` will cause the above instruction that loads from rGC to trap. The handling of this trap is done in the manner similar to handling trap-based `NullPointerExceptions`, except execution is redirected to `CVMCCMruntimeGCRendezvousGlue()` instead of `CVMCCMruntimeThrowNullPointerExceptionGlue()`. This is usually done by the `handleSegv()` function in `jit_arch.c`. See the section on trap-based `NullPointerExceptions` for details.

The result the trap handling makes the execution of the trap causing instruction behave just like a call to `CVMCCMruntimeGCRendezvousGlue()` from compiled code.

14.5 Reference

APIs:

- `CVMJITcheckGC`
- `CVMJITcsPatchRendezvousCalls` (CDC-HI 1.0.1)
- `CVMJITcsUnpatchRendezvousCalls` (CDC-HI 1.0.1)
- `CVMJITenableRendezvousCalls` (CDC-HI 1.1)
- `CVMJITdisableRendezvousCalls` (CDC-HI 1.1)

Macros (CDC-HI 1.1 only):

- `CVMJIT_PATCH_BASED_GC_CHECKS`
- `CVMJIT_TRAP_BASED_GC_CHECKS`
- `CVMJITGlobalState` fields: (1.1 only)
  - `void**`
  - `gcTrapAddr;`

CHAPTER **15**

JIT Stack Maps

Like interpreted methods, stack maps are required for compiled methods so GC knows how to properly scan object references in compiled frames. Unlike interpreted methods, which have stack maps lazily produced for them on demand at GC time, compiled methods have stack maps produced during compilation.

A stack map needs to be produced for every instruction in the compiled method that could possibly become GC-safe. GC safety within the Virtual Machine is covered in more detail in the first few pages of the Creating a Garbage Collector section of the *CDC Porting Guide*.

15.1 Stack Map Components

The are three components of a compiled stack frame that need to be included in (covered by) the stack map:

- Locals
- Temp area (spill area)
- Java parameters on the Java stack

During compilation, the refness of locals is maintained in `con->localRefSet` by the Register Manager `CVMRMstoreJavaLocal()` function. It calls `CVMJITlocalrefsSetRef()` if the local is an object reference or `CVMJITlocalrefsSetValue()` otherwise. This keeps `con->localRefSet` up to date as locals are stored.

The refness of values in the temp area is maintained in `con->RMcommonContext.spillRefSet` by the Register Manager as temp values are loaded and stored to the temp area. Storing occurs when a resource is spilled to the temp area. Loading occurs when a resource that is not currently loaded into a register is pinned. See the Register Manager section for more details on loading and spilling resources.

The refness of Java parameters is maintained by the Stack Manager in `con->SMstackRefSet` as parameters are pushed using `CVMSMpushSingle()` and `CVMSMpushDouble()`, popped using `CVMSMpopParameters()` and also as method results are left on the parameter stack by calling `CVMSMinvocation()`.

Note that only Java parameters are ever found on the evaluation stack, thus it is often referred to as the parameter stack. Any other temporary evaluations are stored to the temp area rather than pushed to the evaluation stack. The handling of Java parameters is covered in more detail in the Stack Manager section.

15.2 GC Points in Compiled Code

As noted above, stack maps are needed at every instruction for which a garbage collection can occur. These are referred to as “GC points” and can occur at any of the following instructions in compiled code:

- Backward branch target
- Exception handler entry point
- Method call
- Object allocation
- Monitor enter/exit
- Calls to C helpers that may cause a GC or become GC-safe

At every GC point, a stack map must be captured and stored with the compiled method so it can be consulted when a GC occurs. If GC finds a compiled method in the backtrace of a thread, it will expect to be able to find a stack map for the current PC of the thread.

GC points are easily observed when tracing generated code. Just look for the “Captured a stack map here” messages in the trace output. Tracing generated code is enabled by building with `CVM_TRACE_JIT=true` and running with `-Xjit:trace=codegen`.

15.3 Capturing Stack Maps

Stack maps are captured (produced) during compilation by calling `CVMJITcaptureStackmap()`, which creates a stack map for the current logical PC. The stack map consists of a bitmap that indicates where object references are stored in each of the three stack map components listed above.

When compilation of the method is complete, CVMJITwriteStackmaps() is called. It copies all the stack maps captured by CVMJITcaptureStackmap(), and moves them to permanent memory that is allocated for the method to hold the stack maps. CVMJITwriteStackmaps() stores the stack maps in con->stackmaps, which is later moved to CVMcmdStackMaps(cmd).

15.4 Accessing Stack Maps

A method's stack maps are accessed by GC at runtime by using CVMcmdStackMaps(CVMmbCmd(mb)), which returns a pointer to the following data structure:

```
                 struct CVMCompiledStackMaps {
                     CVMUint32       noGCPoints;
                     CVMCompiledStackMapEntry smEntries[1];
                 };
```

noGCPoints indicates the size of the smEntries array, whose entries are the following struct:

```
                 struct CVMCompiledStackMapEntry {
                     CVMUint16  pc;    /* offset from start of
method's code */
                     CVMUint8   totalSize; /* total # of bits to
examine */
                     CVMUint8   paramSize; /* outgoing parameters */
                     CVMUint16  state;   /* state bits or offset to
them. */
                 };
```

- pc is the PC offset from the start of the method for this stack map. GC uses it to find the proper stack map for a given PC.
- totalSize is the number of entries in the stack map.
- paramSize is the number of parameters in the stack map for stack maps produced for method invocations. Special handling is needed for invocation stack maps based on whether the invocation is in progress (being made from the topmost frame) or the frame for the invocation has already been pushed. In the later case, the parameters have already been made part of the locals area in the callee's frame, so they are not scanned as part of the caller's frame.

- `state` is a bitmap indicating the refness of each field in the frame, unless `totalSize` is `0xff`, in which case state is used as an offset from the end of `smEnties[]` to the start of large stack map entry, specified by the following data structure:

```
struct CVMCompiledStackMapLargeEntry{
    CVMUint16  totalSize;
    CVMUint16  paramSize;
    CVMUint16  state[1];
};
```

The fields are similar to those in `CVMCompiledStackMapEntry`, except state is now an array of bitmaps indicating the refness of each field.

15.5 `CVMcompiledFrameScanner()`

`CVMcompiledFrameScanner()` is called by GC at runtime to scan compiled frames. It makes use of the compiled frame stack maps described above to properly scan the compiled frame passed to it.

15.6 Porting Effort

When porting within the RISC JIT porting layer, no stack map related porting is needed.

If not porting within the RISC JIT porting layer, the following all have to be ported:

- Proper calls to `CVMJITcaptureStackmap()` and `CVMJITwriteStackmaps()` must be made.
- The refness of temps and locals must be maintained in a way similar to what the Register Manager does (the Register Manager is part of shared RISC code).
- The refness of stack parameters must be maintained in a way similar to what the Stack Manager does (the Stack Manager is part of shared RISC code).

If any changes are made to the layout of a compiled frame, then changes may also be needed in `CVMcompiledFrameScanner()` to account for the changed layout.

Getting stack maps wrong has very serious consequences. Object references can go unscanned, resulting in random crashes. If testing is done with asserts enabled (`CVM_DEBUG_ASSERTS=true`), a missing stack map should result in an assert in

`CVMcompiledFrameScanner()` if the PC with the missing stack map is current when a GC is requested. It usually takes many days of stress testing to bring about such an assert.

15.7 Reference

Primary Source Files:

- `jitstackmap.h`
- `jitstackmap.c`

Data Structures:

- `CVMCompiledStackMaps`
- `CVMCompiledStackMapEntry`
- `CVMCompiledStackMapLargeEntry`

Macros:

- `CVMmbCmd(mb)`
- `CVMcmdStackMaps(cmd)`

APIs:

- `CVMJITcaptureStackmap()`
- `CVMJITwriteStackmaps()`
- `CVMcompiledFrameScanner()`

CHAPTER **16**

JIT Intrinsic Methods

Intrinsic methods (commonly referred to as intrinsics) are Java methods whose semantics are known to the VM implementation. Hence, the VM can choose to implement the execution of these methods in a special way to optimize for speed. As of CDC HI 1.0.1, the dynamic compiler implements a framework for defining and implementing intrinsic methods.

16.1 How Intrinsics Work

When compiling a method, if the dynamic compiler encounters a method invocation operation, it checks to see if the method is one that it knows to be an intrinsic method. This is done in the compiler front-end during the process of generating the intermediate representation (IR) tree of the method being compiled. If the invocation target method is a known intrinsic method, the compiler front-end will emit an IR node tree that uses special intrinsic nodes as opposed to normal invoke nodes.

The resultant IR tree will look like an invocation tree except the node types are different. Normal invocations will use `INVOKE`, `PARAMETER`, and `NULL_PARAMETER` nodes. The `INVOKE` node indicates the method to invoke. The `PARAMETER` nodes form a list of parameters to be passed to the method being invoked, and the `NULL_PARAMETER` node terminates the parameter list.

The intrinsic equivalent will use `INTRINSIC`, `IARG`, and `NULL_IARG` nodes respectively. The meaning of these nodes are analogous to their counterparts.

For example, here's a piece of Java code:

```
public static int doMin(int i, int j) {
  return Math.min(i, j);
}
```

The bytecodes for this method will look like:

```
<0> (0xb6c910): iload_0
<1> (0xb6c911): iload_1
<2> (0xb6c912): invokestatic_quick #2 <java.lang.Math.min(II)I>
<5> (0xb6c915): ireturn
```

Here's an excerpt of the IR tree generated for this code if `Math.min` is not an intrinsic method:

```
<(ID: 9) RETURN_VALUE (int)
 <(ID: 8) INVOKE (int)
   <(ID: 7) PARAMETER (int)
    <(ID: 2) LOCAL (int)  0>
    <(ID: 6) PARAMETER (int)
      <(ID: 3) LOCAL (int)  1>
      <(ID: 5) NULL_PARAMETER (NONE)
   <(ID: 4) CONST_MB (NONE) (java.lang.Math.min(II)I)
```

Here's an excerpt of the IR tree generated for this code if `Math.min` is an intrinsic method:

```
<(ID: 9) RETURN_VALUE (int)
  <(ID: 8) INTRINSIC (int)
    <(ID: 7) IARG (int)
     <(ID: 2) LOCAL (int)  0>
     <(ID: 6) IARG (int)
       <(ID: 3) LOCAL (int)  1>
       <(ID: 5) NULL_IARG (NONE)
    <(ID: 4) CONST_MB (NONE) (java.lang.Math.min(II)I)
```

Having the intrinsic invocation IR tree used different IR node types allows the compiler back-end to generate the code for the intrinsic method differently than for normal method invocations.

16.2 The Intrinsics Framework

For CDC-HI 1.0.1, all intrinsic methods are defined in an intrinsics config list (`CVMJITIntrinsicConfigList`). The list is made up of `CVMJITIntrinsicConfig` records which specifies details about each intrinsic method. The record will indicate details like:

- Method identifiers (class, name, and signature)

- Calling convention to use for this intrinsic
- Attributes of the intrinsic method e.g.
  - Is it a static method?
  - Does it require register spills before calling it?
  - Will it offer a GC point, i.e. need a stackmap at the invocation point?
- Pointer to a native function implementing the intrinsic method, or a pointer to an intrinsic emitter function table for emitting inlined code.

The file `jitintrinsics.h` has a comment section that describes the meaning of all the values that can be used to initialize a `CVMJITIntrinsicConfig` record.

16.2.1 Chaining the Intrinsics Config List

The intrinsics config list can be implemented as chained of multiple lists. For example, on the ARM port, it is implemented as a chain of three lists:

In `src/arm/javavm/include/jit/jit_cpu.h, CVMJITintrinsicsList is #define`'d to be `CVMJITarmIntrinsicsList`.

The intrinsics framework considers `CVMJITintrinsicsList` to be the head of the list. This makes the ARM port's `CVMJITarmIntrinsicsList` the head of the list.

In `src/arm/javavm/runtime/jit/ccmintrinsics_cpu.c`, `CVMJITarmIntrinsicsList` is defined as follows:

```
CVMJIT_INTRINSIC_CONFIG_BEGIN(CVMJITarmIntrinsicsList)
...
CVMJIT_INTRINSIC_CONFIG_END(CVMJITarmIntrinsicsList,
              &CVMJITriscIntrinsicsList)
```

Note that `CVMJITarmIntrinsicsList` specifies `CVMJITriscIntrinsicsList` as a parent list.

In `src/portlibs/jit/risc/ccmintrinsics_risc.c`, `CVMJITriscIntrinsicsList` is defined as follows:

```
CVMJIT_INTRINSIC_CONFIG_BEGIN(CVMJITriscIntrinsicsList)
...
CVMJIT_INTRINSIC_CONFIG_END(CVMJITriscIntrinsicsList,
             &CVMJITriscParentIntrinsicsList)
```

`CVMJITriscIntrinsicsList` specifies `CVMJITriscParentIntrinsicsList` as its parent list. `CVMJITriscParentIntrinsicsList` is defined in `src/portlibs/jit/risc/include/export/jit_risc.h` to be `CVMJITdefaultIntrinsicsList`.

Finally, `CVMJITdefaultIntrinsicsList` is defined in `src/share/javavm/runtime/ccmintrinsics.c` as follows:

```
CVMJIT_INTRINSIC_CONFIG_BEGIN(CVMJITdefaultIntrinsicsList)
...
CVMJIT_INTRINSIC_CONFIG_END(CVMJITdefaultIntrinsicsList, NULL)
```

`CVMJITdefaultIntrinsicsList` specifies NULL as its parent. This means the chain ends here.

Conceptually, the above chain of configuration list means that the ARM list inherits from the RISC list which in turn inherits from the default list. Just like with typical object-oriented inheritance, if we have duplicate definitions of config records in this list, the child list's record would override the parent list's record, i.e., the duplicate record in the parent list will be ignored. This inheritance mechanism means that the target port can reuse some of the record already defined in the default list, and only add and/or override records for intrinsic methods that it wants to customize in a target specific way.

16.2.2 Compiler Front-End Support

The compiler front-end uses this intrinsics config list to identify if a given target method is intrinsic or not. If it is, then the invocation IR tree is generated using intrinsics IR node types instead of normal invocation IR node types.

16.2.3 Compiler Back-End Support

When the compiler back-end encounters the intrinsics IR nodes, it looks up the `CVMJITIntrinsicConfig` record for the method. Based on the information in the record, the compiler back-end will emit code for the intrinsics invocation in different ways. Note that the information in the record are just hints for the compiler back-end.

In order to supports intrinsics, the compiler back-end port will have to implement some means of emitting code for these intrinsics. The compiler back-end will also have to be very careful to be compliant with all the nuances indicated by the intrinsics attributes indicated in the `CVMJITIntrinsicConfig` record. Failure to handle these nuances properly may result in failures that are very difficult to detect and debug. And example of this would be the attribute that indicates that certain class of registers need to be spilled before the intrinsics method is invoked. If the proper register spills are not done, then the values in the registers may not be preserved appropriately during the intrinsics invocation, and unpredictable failures can occur during execution.

16.2.4 Intrinsics Code Generation

In CDC HI 1.0.1, intrinsics code generation can be done using one of three calling conventions:

- Custom compiler back-end defined code emission (`CVMJITINTRINSIC_OPERATOR_ARGS`).
- Native function call with native arguments and return values (`CVMJITINTRINSIC_C_ARGS`).
- Native function call with arguments and return values on the Java stack (`CVMJITINTRINSIC_JAVA_ARGS`).

16.2.5 CVMJITINTRINSIC_OPERATOR_ARGS

This option basically allows the compiler back-end to define its own custom calling convention. This is useful in cases where the intrinsic method can be implemented using special machine instructions, or an optimized sequence of instructions to be inlined into the caller method. Arguments and return values are usually passed in machine registers.

An example of this is the `Math.min` method which can be implemented efficiently as a few instructions inlined into its caller. The following shows the `CVMJITIntrinsicConfig` record for `Math.min`:

```
 {
  "java/lang/Math", "min", "(II)I",
  CVMJITINTRINSIC_IS_STATIC |
  CVMJITINTRINSIC_OPERATOR_ARGS |
  CVMJITINTRINSIC_SPILLS_NOT_NEEDED |
  CVMJITINTRINSIC_STACKMAP_NOT_NEEDED |
  CVMJITINTRINSIC_NO_CP_DUMP,
  CVMJITIRNODE_NULL_FLAGS,
  (void *)&CVMJITRISCintrinsicIMinEmitter,
},
```

`CVMJITRISCintrinsicIMinEmitter` is a table of pointers to functions that the compiler back end can call to:

- Setup the arguments for this intrinsic,
- Emit the code to do the work of the intrinsic
- Retrieve the return value of the intrinsic

16.2.6 CVMJITINTRINSIC_C_ARGS

This option indicates that the intrinsic method will be implemented as a native C or assembly function. The calling convention to be used will be the C calling convention defined by the underlying target platform. For example, arguments and return values would be passed in registers or on the native stack.

An example of this is the `Math.cos` method. The intrinsic is implemented as a direct call to the `CVMfdlibmCos` function. The `CVMJITIntrinsicConfig` record looks like this:

```
{
  "java/lang/Math", "cos", "(D)D",
  CVMJITINTRINSIC_IS_STATIC |
  CVMJITINTRINSIC_C_ARGS |
  CVMJITINTRINSIC_NEED_MINOR_SPILL |
  CVMJITINTRINSIC_STACKMAP_NOT_NEEDED |
  CVMJITINTRINSIC_CP_DUMP_OK,
  CVMJITIRNODE_NULL_FLAGS,
  (void*)CVMfdlibmCos,
},
```

`CVMfdlibmCos` is the native function that will do the work of the intrinsic. Its prototype looks like this:

```
double CVMfdlibmCos(double x)
{
  ...
}
```

16.2.7 CVMJITINTRINSIC_JAVA_ARGS

This option indicates that the intrinsic method will be implemented as a native C or assembly function, but arguments and return values will be passed on the Java stack.

An example of this is the ARM implementation of `Object.hashCode` as an intrinsic. The `CVMJITIntrinsicConfig` record looks like this:

```
{
  "java/lang/Object", "hashCode", "()I",
  CVMJITINTRINSIC_IS_NOT_STATIC |
  CVMJITINTRINSIC_JAVA_ARGS |
  CVMJITINTRINSIC_NEED_MAJOR_SPILL |
  CVMJITINTRINSIC_NEED_STACKMAP |
  CVMJITINTRINSIC_CP_DUMP_OK |
  CVMJITINTRINSIC_NEED_TO_KILL_CACHED_REFS |
  CVMJITINTRINSIC_FLUSH_JAVA_STACK_FRAME,
  CVMJITIRNODE_THROWS_EXCEPTIONS,
  CVMCCMARMintrinsic_java_lang_Object_hashCodeGlue,
},
```

The native function has the prototype: `void(*)(void)`. If any additional arguments are needed, then the `CVMJITIntrinsicConfig` record's function pointer should point to assembly glue logic which sets up the additional arguments before invoking the actual native function to do the needed work. The glue logic will also be responsible for marshalling any return values back onto the Java stack. The compiler back-end is only responsible for emitting code to setting up the arguments on the Java stack before invoking the intrinsic method, and retrieving the return value from the Java stack afterwards if necessary.

In the above example, `CVMCCMARMintrinsic_java_lang_Object_hashCodeGlue` is a piece of glue code that marshals the arguments from the Java stack onto the native stack as well as set up any additional arguments needed, calls a C function to do the work of the intrinsic, and lastly marshals the return value of the function back onto the Java stack.

16.3 The Value of Intrinsics

Being able to support intrinsics basically allows the dynamic compiler to do targeted optimizations on methods whose semantics are known ahead of time. This yields the following benefits:

- Certain *intrinsic methods can be implemented using special machine instructions* where appropriate. The complexity of the compiler is reduced as it does not need to support these special instructions or instruction sequences in the general case.
- The VM can get the benefit of more *advanced and time consuming or hand-crafted optimization techniques on intrinsic methods* without having to implement the general optimization technique in the dynamic compiler, and without incurring compilation time.
- The VM can implement native methods as well as Java methods as intrinsics. This means that *intrinsic native methods can be inlined into the caller*. Normally, native methods cannot be inlined.

16.4 Disabling Intrinsics

Because support for the intrinsics framework can be difficult to implement, a porting engineer may not want to attempt it for an initial port. Alternatively, if it has been determined that intrinsics would not yield any benefit for certain target platforms, then a port of the dynamic compiler may not want to support intrinsics.

To disable support for intrinsics, just `#undef CVMJIT_INTRINSICS` in the `src/<cpu>/javavm/include/jit/jit_cpu.h` file.

16.5 Reference

Primary Source Files:

- `jitintrinsics.h`
- `ccmintrinsics.c`

Data Structures:

- `CVMJITIntrinsicConfigList`
- `CVMJITIntrinsicConfig`
- `CVMJITIntrinsicEmitterVtbl`

Macros:

- `CVMJIT_INTRINSIC_CONFIG_BEGIN(listname)`
- `CVMJIT_INTRINSIC_CONFIG_END(listname, parentlistname)`

CHAPTER **17**

JIT Debugging Support

17.1 Tracing

CDC-HI provides a tracing facility to aid in the debugging of compiled methods. When tracing is enabled, certain informative text is sent to the console. To add support for tracing, build with `CVM_TRACE_JIT=true`, which is the default for `CVM_DEBUG=true` builds.

To enable tracing at runtime, use the -`Xjit:trace=<option>` command line option. The available tracing options are described in the CDC Runtime Guide. A few are listed here:

- `status`: Prints a line of status each time a method is compiled or decompiled.
- `bctoir`: Prints the Java bytecodes being compiled, along with the IR they are converted into.
- `codegen`: Prints the generated code in a format similar to an assembler listing.

More than one tracing option can be specified at a time. For example:

```
-Xjit:trace=status+bctoir+codegen
```

If `-Xjit:trace=codegen` is specified with a build done with `CVM_JIT_DEBUG=true`, then the JCS rules used to generate code are interspersed with the generated code. This is known as "rule tracing" and is a very powerful tool for learning how IR trees are actually parsed. An example of this rule tracing follows:

```
                    @ Doing node  49 codegen rule [148] param32:
reg32
                312:st      %o0,[%i5 + 0]
                    @ Doing node  50 codegen rule [ 26] reg32:
LOCAL32
```

```
                 316:ld     [%i4 + -12], %l1@ Java local cell # 1
                    @ Doing node  50 codegen rule [148] param32:
reg32
                 320:st      %l1,[%i5 + 4]
                    @ Doing node  54 codegen rule [ 35] reg32:
ICONST_32
                    @ Doing node  54 codegen rule [148] param32:
reg32
                 324:sethi  1036288, %i1@ const 1061158912
                 328:st      %i1,[%i5 + 8]
```

17.2 Controlling Which Methods Are Compiled

CDC-HI provides a filtering mechanism for controlling which methods get compiled. Although designed to limit compilation to a specified list of classes or methods, it can also be used to allow compilation in all but the specified list. To make use of this filtering mechanism, CDC-HI must be build with `CVM_JIT_DEBUG=true`.

The following code can be found in `jitdebug.c`:

```
#undef USE_COMPILATION_LIST_FILTER
#ifdef USE_COMPILATION_LIST_FILTER
CVMJIT_DEBUG_METHOD_LIST_BEGIN(methodsToCompile)
                 {<method name or class name},
                 {<method name or class name},
                 ...
CVMJIT_DEBUG_METHOD_LIST_END(methodsToCompile)
#endif /* USE_COMPILATION_LIST_FILTER */
```

To limit compilation to methods and classes specified in the list, first change the `#undef USE_COMPILATION_LIST_FILTER` to `#define USE_COMPILATION_LIST_FILTER`. Next add all the desired class and methods to the list. When specifying methods, the full signature of the method must be given. Examples are provided in the default list in `jitdebug.c`. These can be deleted.

To change the `methodsToCompile` list to a “do not compile” list, the following code needs to be modified:

```
                 CVMBool
                 CVMJITdebugMethodIsToBeCompiled(CVMExecEnv *ee,
CVMMethodBlock *mb)
                 {
                 #ifdef USE_COMPILATION_LIST_FILTER
                   if (CVMJITdebugMethodIsInMethodList(ee, mb,
&methodsToCompile))                               {
                        return CVM_TRUE;
                     }
                     return CVM_FALSE;
                 #else
                     return CVM_TRUE;
                 #endif
                 }
```

Change the first "`return CVM_TRUE;`" to "`return CVM_FALSE;`" and the "`return CVM_FALSE;`" to "`return CVM_TRUE;`"

17.3 Controlling Which Methods Are Traced

`jitdebug.c` also contains a filtering mechanism to control which methods get traced when using `-Xjit:trace=bctoir` or `-Xjit:trace=codegen`. It works the same as the `methodsToCompile` list described above, except the list to edit is `methodsToTraceCompilation` and the macro to enable is `USE_TRACING_LIST_FILTER`.

17.4 `CVMJITcodeCacheFindCompiledMethod()`

The function `CVMJITcodeCacheFindCompiledMethod()` can be used to map a native PC to a compiled method. It is especially useful when called from gdb to determined which compiled method crashed. `CVMJITcodeCacheFindCompiledMethod()` is described in the Debugging Support section of the Code Cache Manager chapter.

17.5 GDB Support

Information on debugging CDC-HI using GDB can be found in the Appendix A: Debugging with gdb in the *CDC Porting Guide*. The information provided in the appendix can also be useful when using other debuggers.

17.6 Trap-based `NullPointerExceptions`

Disabling trap-based `NullPointerExceptions` may be necessary when debugging. See the Trap-based `NullPointerExceptions` chapter for details.

17.7 Reference

Build Options:

- `CVM_DEBUG=true`
- `CVM_TRACE_JIT=true`
- `CVM_JIT_DEBUG=true`

Runtime Options:

- `-Xjit:trace=status`
- `-Xjit:trace=bctoir`
- `-Xjit:trace=codegen`

APIs:

- `CVMJITcodeCacheFindCompiledMethod()`
- `handleSegv()`

CHAPTER 18

Profiling Dynamically Compiled Code

The CDC-HI JIT has built-in support for profiling dynamically compiled code (referred to as "JIT profiling"). It can be used to determine the percentage of time spent in each compiled method, along with the time spent in assembler helper functions that have been copied to the code cache (see the Code Cache Manager chapter for details on copying assembler code to the code cache).

18.1 Building Support for JIT Profiling

To include support for JIT profiling, build with `CVM_JIT_PROFILE=true`. This will include profiling support in the CDC-HI binary, but unless enabled at runtime it has no performance impact.

18.2 Enabling JIT Profiling at Runtime

To enable JIT profiling at runtime, run with `-Xjit:profile=<filename>`. On Linux, this will result in the use of the `profil()` API to profile the code cache. `profil()` uses timer interrupts to profile the PC. 100 samples are taken per second. It causes about a 1% performance hit.

18.3 Profiling Output Format

When the VM exits, the profile will by dumped into the specified filed. The following is an example of the output file:

```
                 totalSampleCount = 336 = 37.07% of program execution
time


                 column #1: estimated savings if method is inlined
                 column #2: % of program time spent in method
                 column #3: % of code cache time spent in method


                  <1>  <2>  <3>
                 -----------------------
                     0.77% 2.08% CVMCCMruntimeMonitorEnterGlue
                     1.43% 3.87% CVMCCMruntimeMonitorExitGlue
                     0.88% 2.38% CVMCCMruntimeNewGlue
                     5.41% 14.58%
CVMCCMinvokeNonstaticSyncMethodHelper
                     0.66% 1.79% CVMCCMreturnFromMethod
                     4.30% 11.61% CVMCCMreturnFromSyncMethod
                     9.05% 24.40% java.lang.StringBuffer.append()
                  3.72% 3.20% 8.63% java.lang.String.<init>()
                     11.25% 30.36% Strings.execute()
                  0.31% 0.11% 0.30% java.lang.String.toString()
```

Note that there is also a top 10 list printed at the end, but it has been omitted for this example.

The first line of output contains total number of samples taken (`totalSampleCount`). In this case there were 336, which would represent 3.36 seconds of execution time. The first line also includes the percent of total execution time that was spent in the profiled area (the code cache). In this case 37.07%. From this you can deduce that the total execution time was 9.06 seconds (3.36 / .3707). The time not spent in the code cache is spent in the VM and OS libraries.

For each method or helper function in the code cache that had at least one sample taken, a line of output is provided containing the following:

- The first column is the estimated savings if the method had been inlined. This column can be ignored since the user has little control over methods that don't get inlined.
- The second column is the percentage of total program time that was spent in the method or helper function. This is the most meaningful value provided.
- The third column is the amount of time spent in each method or helper function as a percentage of the amount time spent executing in the code cache.
- The fourth column is the name of the method or helper function that was profiled.

18.4 Instruction Level Profiling

If `-Xjit:profileInstructions=true` is also specified, the sample count for each pc offset will also be included in the profiling output. This allows profiling to be done at the instruction level. However, due to instruction latencies, sample counts for any given instruction usually more accurately reflect the time spent in an earlier instruction rather than the one specified. The instruction responsible for doing a function return usually results in a large sample count at the return address rather than at the instruction doing the return. This is especially true on ARM when ldm is used to restore registers and load the PC with the return address.

18.5 Application Exit

If the application calls `System.exit()`, then normally VM shutdown code is skipped and the profile is not dumped. To ensure that the profile is dumped in this case, `-XsafeExit` should be specified on the command line.

If the application is forced to exit, such as when forcing termination of a process using kill, then this will also prevent the profiling data from being dumped on exit. The only workaround for this in CDC-HI 1.0.1 is to modify the VM to call `CVMJITcodeCacheDumpProfileData()` at some predetermined time. In CDC-HI 1.1, the new `sun.misc.CVM.dumpCompilerProfileData()` API can be used by the application to trigger the dump of the profiling data from Java at any time.

18.6 Decompilation

Profiling data is only accurate if no decompilation takes place. This is because profiling data is gathered simply by keeping track of the number of samples taken at any given offset into the code cache. The mapping of instructions to methods does not take place until the profiling data is dumped. Because of this, decompilation will cause inaccurate results.

Decompilation will take place when a class is unloaded or the code cache has filled. For most applications, class unloading does not occur unless the application has a custom `ClassLoader`, so decompilation due to class unloading is usually not an issue. To prevent decompilation due to running out of space in the code cache, a sufficiently large code cache needs to be provided. To verify that no decompilation is taking place, CDC-HI can be build with `CVM_TRACE_JIT=true` and then run with `-Xjit:trace=status.` A message will appear each time a method is decompiled.

18.7 Porting Effort

If `profil()` is supported on the platform, then very little porting effort is necessary. All that is needed is to `#include "portlibs/posix/posix_jit_profil.h"` from `jit_arch.h`.

If `profil`() is not supported, then the functionality found in `portlibs/posix/posix_jit_profil.h` needs to be provided by the port. However, a port can choose not to support JIT profiling and simply never build with `CVM_JIT_PROFILE=true`.

18.8 Reference

Primary Source Files:

`jitcodebuffer.h`

`jitcodebuffer.c`

`posix_jit_profil.h`

APIs:

`CVMJITcodeCacheDumpProfileData()`

`profil()`

Build Options:

`CVM_JIT_PROFILE=true`

Runtime Options

`-Xjit:profile=<filename>`

`-Xjit:profileInstructions=true`

`-XsafeExit`

CHAPTER **19**

Assembler Listings for Dynamically Compiled Code

CDC-HI provides two facilities for generating assembler listings for generated code. The first is part of the standard CDC JIT tracing support. JIT tracing support is added at build time using `CVM_TRACE_JIT=true` (defaults to true for `CVM_DEBUG=true` builds). Tracing of generated code can then be enabled at runtime using the `-Xjit:trace=codegen`. The second facility is found in `jitcomments.h` and is used to make the tracing of generated code more flexible.

19.1 Example Code Generation Assembler Listing

The following is an example assembler listing for dynamically compiled code that can be produced using the tracing facitlities provided with CDC-HI:

```
@ Do putfield:
0x40756518 924: ldr r9, [JFP, #+28] @ Java temp cell # 1
0x4075651c 928: add r11, r9, #8   @ fieldAddr = obj + fieldOffset;
0x40756520 932: str r0, [r11, #+0] @ putfield(fieldAddr, valueObj);
0x40756524 936: ldr r8, [pc, #+0]  @ cardTableVirtualBase
0x40756528 940: mov r7, #0 @ zero
0x4075652c 944: strb r7, [r8, +r11, LSR #9] @ mark card table
```

The first column of each line contains the address of the instruction and the second column contains the offset from the start of the compiled method. The remainder of the line consists of the generated instruction and an applicable comment if provided.

19.2 CVMtraceJITCodegen() and CVMtraceJITCodegenExec()

Traces are conditionally generated at runtime (based on the build and runtime options mentioned above) using the macros `CVMtraceJITCodegen()` and `CVMtraceJITCodegenExec()`. For example:

```
CVMtraceJITCodegenExec({
printPC(con);
CVMconsolePrintf("          sub JSP, JFP, #%d", offset);
});
CVMtraceJITCodegen(("call method"));
```

`CVMconsolePrintf()` is essentially the same as `fprintf()` to `stderr`, except that it supports some additional arguments types, such as `CVMClassBlock*` and `CVMMethodBlock*`. See `CVMformatStringVaList()` in utils.c for details. It is the standard mechanism for sending debugging messages to the console from within the CDC-HI virtual machine.

`CVMtraceJITCodegen()` uses `CVMconsolePrintf()` to trace its arguments, but only if `-Xjit:trace=codegen` was specified on the command line. Likewise, `CVMtraceJITCodegenExec()` only executes its argument if `-Xjit:trace=codegen` was specified on the command line, allowing for more complex conditional traces. Both of these macros expand to nothing if `CVM_TRACE_JIT=false`.

Note – Usually `CVMJITprintCodegenComment()` is used instead of `CVMtraceJITCodegen()`.

19.3 CVMJITprintCodegenComment()

`CVMJITprintCodegenComment()` is just a helper function that is used in placed of `CVMtraceJITCodegen()` to make comment printing more consistent. It just adds some tab spacing to the start of the comment, plus the `@` comment character, and includes a newline at the end. It is used for comments that will take up an entire line.

```
CVMJITprintCodegenComment(("Method prologue"));
```

Like `CVMtraceJITCodegen()`, it only traces if `-Xjit:trace=codegen` was specified on the command line, and compiles into nothing if `CVM_TRACE_JIT=false`.

19.4 `CVMJITaddCodegenComment()` and `CVMJITdumpCodegenComments()`

One downside of using `CVMtraceJITCodegen()` is that frequently you want to add a comment to the generated code trace, but don't have the proper context. For example,

```
CVMCPUemitMemoryReferenceImmediate(con,
CVMCPU_STR32_OPCODE,
CVMCPU_ARG1_REG,
VMCPU_JFP_REG,
mbOffset);
```

produces the following trace:

```
0x40756198  28: str r0, [JFP, #+12]
```

`CVMCPUemitMemoryReferenceImmediate()` is responsible for emitting the above trace. It has no idea what the purpose of the code is, so it cannot provide a useful comment. However, the caller of `CVMCPUemitMemoryReferenceImmediate()` probably knows the reason for emitting the code, and would like to see a comment indicating the reason. To do this, `CVMJITaddCodegenComment()` is used:

```
CVMJITaddCodegenComment((con, "Store MB into frame"));
CVMCPUemitMemoryReferenceImmediate(con,
CVMCPU_STR32_OPCODE,
CVMCPU_ARG1_REG,
CVMCPU_JFP_REG,
mbOffset);
```

which produces:

```
0x40756198  28: str r0, [JFP, #+12] @ Store MB into frame
```

It is up to `CVMCPUemitMemoryReferenceImmediate()` to make sure the added comment gets traced. To do this it calls `CVMJITdumpCodegenComments()`, which will print out all comments added by `CVMJITaddCodegenComment()`, and also clear them from the queue.

19.5 `CVMJITpushCodegenComment()` and `CVMJITpopCodegenComment()`

Sometimes it is necessary to generate some code with a comment similar to the `CVMCPUemitMemoryReferenceImmediate()` example above, but you know that whoever called you already added a comment, and you don't want to immediately use this comment. In this case the comment can be saved away using `CVMJITpopCodegenComment()` and later restored for use with `CVMJITpushCodegenComment()`. This is common in code emitters that may need to emit more than one instruction. The coding sequence is as follows:

```
CVMCodegenComment *comment;
/* save away current pending comment */
CVMJITpopCodegenComment(con, comment);
/* add your own comment */
CVMJITaddCodegenComment((con, "my comment"));
<Call some API that will emit code and dump your comment>
/* restore original comment */
CVMJITpushCodegenComment(con, comment);
<call another API that will emit code and dump restored comment>
```

19.6 `CVMJITsetSymbolName()` and `CVMJITgetSymbolName()`

`CVMJITsetSymbolName()` plays a role similar to `CVMJITaddCodegenComment()`. It is used to set the name of a constant pool symbol. The symbol will be retrieved by `CVMJITgetSymbolName()` and used in a comment when the constant pool entry is dumped.

`CVMJITgetSymbolName()` is usually used just before calling `CVMRMgetResourceForConstant32()` (which will retrieve and store away the symbols), or before calling a function known to call `CVMRMgetResourceForConstant32()` or `CVMRMbindResourceForConstant32()`, and the caller is providing the constant.

CODE EXAMPLE 19-1 Example 1

```
CVMJITsetSymbolName((con, "cardTableVirtualBase"));
```

```
cardtableReg = CVMRMgetResourceForConstant32(
CVMRM_INT_REGS(con),
CVMRM_ANY_SET, CVMRM_EMPTY_SET,
(CVMInt32)CVMglobals.gc.cardTableVirtualBase);
0x4075658c 1040:.word 2531911 @ cardTableVirtualBase
```

CODE EXAMPLE 19-2 Example 2

```
CVMJITsetSymbolName((con, "mb %C.%M", CVMmbClassBlock(mb), mb));
dest = CVMRMbindResourceForConstant32(
CVMRM_INT_REGS(con), (CVMInt32)mb);
0x407565a0 1060: word 4362584 @ mb java.lang.String.<init>([CII)V
```

19.7 Reference

Primary Source Files:

`jitcomments.h`

`jitcomments.c`

`jitutils.h - for CVMJITtraceXXX() macros`

Command Line Options:

`-Xjit:trace=codegen`: enable tracing of code generation

Build Options:

`CVM_TRACE_JIT=true`

CHAPTER 20

Code Generation Examples

Here are a few code generation examples, including examples of accessing object fields, accessing array elements, method invocation and simple arithmetic. The assembly code in the examples are generated on ARM platform.

20.1 Example 1

This is a simple example of field accessing and arithmetic operation.

CODE EXAMPLE 20-1 Java code

```
class c {
    int _foo;
    public int f(int k) {
      return _foo + k;
    }
  }
```

CODE EXAMPLE 20-2 Bytecodes

```
  aload_0   // this
  getfield 2 // _foo
  iload_1   // k
  iadd    // _foo + k
  ireturn
```

CODE EXAMPLE 20-3 IR

```
  <(ID: 9) RETURN_VALUE (int)
   <(ID: 8) ADD (int)
     <(ID: 6) FETCH (int)
```

```
      <(ID: 5) FIELD_REF (int)
        <(ID: 2) LOCAL (reference)  0>
        <(ID: 4) RESOLVE_REFERENCE (NONE)
         <(ID: 3) CONST_GETFIELD_FB_UNRESOLVED (NONE)
                           (cpIndex 2)
     <(ID: 7) LOCAL (int)  1>
```

CODE EXAMPLE 20-4 Generated code

Following is a sequence of generated code corresponding to the above IR tree. The first column is the physical address of the generated instructions. The second column is the logical address. Comments start with @.

```
         @ Doing node 2 codegen rule [26] reg32: LOCAL32
0x40756cc0 64: ldr   r11, [JFP, #-8] @ Java local cell # 0
         @ Doing node 4 codegen rule [184] reg32: RESOLVE_REF
         @ Resolving an instance field:
0x40756cc4 68: ldr   r11, [pc, #-8] @ load cachedConstant
0x40756cc8 72: mov   r2, #2 @ ARG3 = cpIndex
         @ call CVMCCMruntimeResolveGetfieldFieldOffset
0x40756ccc 76: bl   PC=(-10704)
>>>>>>>>>Push Code Buffer to PC = 68 (0x40756cc4) >>>>>>>>
0x40756cc4 68: ldr   r11, [pc, #+4] @ load cachedConstant
<<<<<<<<<Pop Code Buffer to PC = 80 (0x40756cd0) <<<<<<<<<
0x40756cd0 80: .word  -1   @ cachedConstant
         @ Captured a stackmap here.
0x40756cd4 84: ldr   r11, [pc, #-12] @ load cachedConstant
         @ Doing node 4 codegen rule [15] memSpec: reg32
         @ Doing node 6 codegen rule
         @  [105] reg32: FETCH32 FIELDREF32 reg32 memSpec
         @ Do getfield:
0x40756cd8 88: ldr   r9, [JFP, #-8] @ Java local cell # 0
         @ value{I|F} = getfield(obj, fieldIdx);
0x40756cdc 92: ldr   r10, [r9, +r11]
         @ Doing node 7 codegen rule [26] reg32: LOCAL32
0x40756ce0 96: ldr   r11, [JFP, #-4] @ Java local cell # 1
         @ Doing node 7 codegen rule [13] aluRhs: reg32
         @ Doing node 8 codegen rule
         @  [42] reg32: IADD32 reg32 aluRhs
```

```
0x40756ce4 100: add   r8, r10, r11 LSL #0
         @ Doing node 9 codegen rule [177] root: IRETURN reg32
0x40756ce8 104: str   r8, [JFP, #-8] @ Java local cell # 0
0x40756cec 108: sub   JSP, JFP, #4
         @ PREV (r6) = frame.prevX for return helper
         @ to use
0x40756cf0 112: ldr   r6, [JFP, #+0]
         @ goto CVMCCMreturnFromMethod
0x40756cf4 116: bl   PC=(-7824)
```

In this example, the ldr instruction at logical address 64 is doing IR node 2 by loading local #0 (this object ref) from the Java frame. The instructions at address 68 ~ 84 are generated corresponding to node 4, `RESOLVE_REFERENCE`. A helper function, `CVMCCMruntimeResolveGetfieldFieldOffset` is called to resolve the field reference. The mov instruction at address 72 sets up the third argument, `cpIndex`, before calling the helper function. The rest of the arguments are set up by assembler glue code. The result of the resolved field is stored at address 80 by the helper function. After the helper returns, the instruction at address 84 loads the result into the target register. The instructions at 88 ~ 92 are for node 6, `FETCH`. The first ldr instruction loads the object reference (local #0) from the Java frame, the second ldr loads the field value from the object using the resolved field offset. The instruction at 96 loads local #1 from the Java frame, which is corresponding to node 7. The add instruction at address 100 is generated for node 8. The rest of the instructions (104 ~ 116) are for `RETURN_VALUE`, node 9. First the result of the add is stored into the Java frame as local #0. Then the sub instruction adjusts the JSP (the top of the Java stack). The assembler code `CVMCCMreturnFromMethod` is called to do the return.

20.2 Example 2

CODE EXAMPLE 20-5 Java code

```
    public static int arith(int x, int y, int z) {
        return x * y + z;
    }
```

CODE EXAMPLE 20-6 Bytecodes

```
<0> iload_0
<1> iload_1
<2> imul
<3> iload_2
```

```
<4> iadd
<5> ireturn
```

Where local 0, 1, and 2 correspond to x, y and z respectively.

CODE EXAMPLE 20-7 IR tree

```
  <(ID: 7) RETURN_VALUE (int)
   <(ID: 6) ADD (int)
     <(ID: 4) MUL (int)
      <(ID: 2) LOCAL (int)  0>
      <(ID: 3) LOCAL (int)  1>
     <(ID: 5) LOCAL (int)  2>
```

CODE EXAMPLE 20-8 Generated code

```
     @ Doing node  2 codegen rule [ 26] reg32: LOCAL32
64:             ldr r11, [JFP, #-12]@ Java local cell # 0
     @ Doing node  3 codegen rule [ 26] reg32: LOCAL32
68:             ldr r10, [JFP, #-8]@ Java local cell # 1
     @ Doing node  4 codegen rule [ 47] reg32: IMUL32 reg32 reg32
72:             mul r9, r11, r10
     @ Doing node  5 codegen rule [ 26] reg32: LOCAL32
76:             ldr r8, [JFP, #-4]@ Java local cell # 2
     @ Doing node  5 codegen rule [ 13] aluRhs: reg32
     @ Doing node  6 codegen rule [ 42] reg32: IADD32 reg32 aluRhs
80:             add r7, r9, r8 LSL #0
     @ Doing node  7 codegen rule [177] root: IRETURN reg32
84:             str r7, [JFP, #-12]@ Java local cell # 0
88:             sub JSP, JFP, #8
     @ PREV (r6) = frame.prevX for return helper to use
92:             ldr r6, [JFP, #+0]@ goto CVMCCMreturnFromMethod
96:             bl  PC=(-6132)
```

20.3 Example 3

This is a field accessing example with explicit `NULL` check.

CODE EXAMPLE 20-9 Java code

```
  class C {
    Object f;
    [...]
  }
  class C2 {
    public static Object accessField(C x)
    {
      return x.f;
    }
  }
```

CODE EXAMPLE 20-10 Byte-codes

```
<0> aload_0
<1> getfield <C.x>
<4> areturn
                 IR
  <(ID: 8) RETURN_VALUE (reference)
   <(ID: 7) FETCH (reference)
     <(ID: 6) FIELD_REF (reference)
      <(ID: 5) NULL_CHECK (reference)
        <(ID: 2) LOCAL (reference)  0>
      <(ID: 4) RESOLVE_REFERENCE (NONE)
        <(ID: 3) CONST_GETFIELD_FB_UNRESOLVED (NONE)
                           (cpIndex 2)
```

CODE EXAMPLE 20-11 Generated code

```
     @ Doing node  2 codegen rule [ 26] reg32: LOCAL32
64:              ldr r11, [JFP, #-8]@ Java local cell # 0
     @ Doing node  5 codegen rule [139] reg32: NULLCHECK reg32
68:              cmp r11, #0 @ NULL check
     @ CVMCCMruntimeThrowNullPointerExceptionGlue
72:              bleqPC=(-9688)
     @ Doing node  4 codegen rule [184] reg32: RESOLVE_REF
     @ Resolving an instance field:
76:              ldr r11, [pc, #+4]@ load cachedConstant
80:              mov r2, #2  @ ARG3 = cpIndex
     @ call CVMCCMruntimeResolveGetfieldFieldOffset
84:              bl  PC=(-9036)
88:              .word-1 @ cachedConstant
```

```
     @ Captured a stackmap here.
92:             ldr r11, [pc, #-12]@ load cachedConstant
     @ Doing node  4 codegen rule [ 15] memSpec: reg32
     @ Doing node  7 codegen rule
     @  [104] reg32: FETCH32 FIELDREFOBJ reg32 memSpec
     @ Do getfield:
96:             ldr r9, [JFP, #-8]@ Java local cell # 0
     @ valueObj = getfield(obj, fieldIdx);
100:            ldr r10, [r9, +r11]
     @ Doing node  8 codegen rule [177] root: IRETURN reg32
104:            str r10, [JFP, #-8]@ Java local cell # 0
108:            sub JSP, JFP, #4
     @ PREV (r6) = frame.prevX for return helper to use
112:            ldr r6, [JFP, #+0]@ goto CVMCCMreturnFromMethod
116:            bl  PC=(-6156)
```

In this example, the ldr instruction at address 64 is generated corresponding to node 2 in the IR tree. It loads the object reference from local #0 into a register. The next instruction performs the `NULLCHECK` (node 5) on the loaded object reference by doing a comparison. If the object is `NULL`, a `NullPointerException` is thrown by calling `CVMCCMruntimeThrowNullPointerExceptionGlue` (the bleq instruction at address 72).

20.4 Example 4

This is a simple example of array element accessing with explicit `NULL` check.

CODE EXAMPLE 20-12 Java code

```
  public static Object accessArray(Object[] arr, int idx)
  {
    return arr[idx];
  }
```

CODE EXAMPLE 20-13 Bytecodes

```
<0> aload_0
<1> iload_1
<2> aaload
<3> areturn
```

CODE EXAMPLE 20-14 IR

```
  <(ID: 10) RETURN_VALUE (reference)
   <(ID: 9) FETCH (reference)
     <(ID: 7) INDEX (int)
      <(ID: 2) IDENTITY (reference) (ref count: 2)
        <(ID: 8) LOCAL (reference)  0>
      <(ID: 6) BOUNDS_CHECK (NONE)
        <(ID: 3) LOCAL (int)  1>
        <(ID: 5) ARRAY_LENGTH (int)
         <(ID: 4) NULL_CHECK (reference)
           <(ID: 2) IDENTITY (reference) (ref count: 2)
```

CODE EXAMPLE 20-15 Generated code

```
     @ Doing node  8 codegen rule [ 26] reg32: LOCAL32
64:              ldr r11, [JFP, #-8]@ Java local cell # 0
     @ Doing node  2 codegen rule [ 38] reg32: IDENT32 reg32
     @ Doing node  3 codegen rule [ 26] reg32: LOCAL32
68:              ldr r10, [JFP, #-4]@ Java local cell # 1
     @ Doing node  2 codegen rule [ 38] reg32: IDENT32 reg32
     @ Doing node  4 codegen rule [139] reg32: NULLCHECK reg32
72:              cmp r11, #0 @ NULL check
76:              bleqPC=(-9988)@ NULL check
     @ Doing node  5 codegen rule [ 62] reg32: ALENGTH reg32
80:              ldr r9, [r11, #+8]@ arraylength
     @ Doing node  6 codegen rule
     @  [169] reg32: BOUNDS_CHECK reg32 reg32
84:              cmp r9, r10 LSL #0
     @ CVMCCMruntimeThrowArrayIndexOutOfBoundsExceptionGlue
88:              bllsPC=(-9928)
     @ Doing node  6 codegen rule [ 97] arraySubscript: reg32
     @ Doing node  9 codegen rule
     @  [101] reg32: FETCH32 INDEX reg32 arraySubscript
     @ Do load(arrayObj, index) (elem type=L):
92:              add r8, r11, r10 LSL #2
96:              ldr r9, [r8, #+12]
     @ Doing node  10 codegen rule [177] root: IRETURN reg32
100:             str r9, [JFP, #-8]@ Java local cell # 0
```

```
104:             sub JSP, JFP, #4
     @ PREV (r6) = frame.prevX for return helper to use
108:             ldr r6, [JFP, #+0]@ goto CVMCCMreturnFromMethod
112:             bl  PC=(-6456)
```

The instructions at address 72 and 76 are generated to do the `NULL_CHECK` (node 4) for the array. If the `NULL` check fails, a `NullPointerException` is thrown (the bleq instruction at address 76). The ldr instruction at address 80 loads the array length into a register. The next two instructions are generated corresponding to the `BOUNDS_CHECK` (node 6) node. Bounds check is done by comparing the index and the array length. If bounds check fails, the blls calls `CVMCCMruntimeThrowArrayIndexOutOfBoundsExceptionGlue` to throw an `ArrayIndexOutOfBoundsException`. The instructions at 92 and 96 load the indexed array element if both the `NULL` check and array bounds check pass.

20.5 Example 5

This is an example of method invocation with parameters and return value.

CODE EXAMPLE 20-16 Java code

```
  class C {
    public static int f(int arg1, int arg2, int arg3)
    {
      return arg1 + arg2 + arg3;
    }
    public static int g(int v1, int v2, int v3) {
      int result = f(v1, v2, v3);
      return result * result;
    }
  }
```

CODE EXAMPLE 20-17 Bytecodes

```
For g():
<0> iload_0
<1> iload_1
<2> iload_2
<3> invokestatic #3  // f(III)
<6> istore_3
<7> iload_3
```

```
<8> iload_3
<9> imul
<10> ireturn
```

CODE EXAMPLE 20-18 IR

```
  <(ID: 12) TEMP (NONE)
   <(ID: 11) IDENTITY (int) (ref count: 3)
     <(ID: 14) INVOKE (int)
      <(ID: 10) PARAMETER (int)
        <(ID: 2) LOCAL (int)  0>
        <(ID: 9) PARAMETER (int)
         <(ID: 3) LOCAL (int)  1>
         <(ID: 8) PARAMETER (int)
           <(ID: 4) LOCAL (int)  2>
           <(ID: 7) NULL_PARAMETER (NONE)
      <(ID: 6) RESOLVE_REFERENCE (NONE)
        <(ID: 5) CONST_STATIC_MB_UNRESOLVED (NONE) (cpIndex 3)
  <(ID: 15) RETURN_VALUE (int)
   <(ID: 13) MUL (int)
     <(ID: 11) IDENTITY (int) (ref count: 3)
     <(ID: 11) IDENTITY (int) (ref count: 3)
```

CODE EXAMPLE 20-19 Generated code

```
     @ Doing node  2 codegen rule [ 26] reg32: LOCAL32
64:              ldr r11, [JFP, #-16]@ Java local cell # 0
     @ Doing node  2 codegen rule [148] param32: reg32
68:              str r11, [JSP], #+4
     ...
     @ Doing node  6 codegen rule [184] reg32: RESOLVE_REF
     @ Resolving a static method:
88:              ldr r0, [pc, #+4]@ load cachedConstant
92:              mov r2, #3  @ ARG3 = cpIndex
     @ call CVMCCMruntimeResolveStaticMethodBlockAndClinit
96:              bl  PC=(-9616)
100:             .word-1 @ cachedConstant
     @ Captured a stackmap here.
104:             ldr r0, [pc, #-12]@ load cachedConstant
     @ Doing node  14 codegen rule
     @  [125] invoke32_result: INVOKE32 parameters reg32
     @ Invoke a method w/ a 32bit return type
108:             mov lr, pc LSL #0@ call method through mb
112:             ldr pc, [r0, #+0]
     @ Captured a stackmap here.
     @ Doing node  14 codegen rule [142] reg32: invoke32_result
116:             ldr r11, [JSP, #-4]!
     @ Doing node  11 codegen rule [ 38] reg32: IDENT32 reg32
     @ Doing node  12 codegen rule [151] effect: FOR_TEMP reg32
     @ Doing node  12 codegen rule [  1] root: effect
     @ Doing node  11 codegen rule [ 38] reg32: IDENT32 reg32
     @ Doing node  11 codegen rule [ 38] reg32: IDENT32 reg32
     @ Doing node 13 codegen rule [ 47] reg32: IMUL32 reg32 reg32
120:             mul r10, r11, r11
     @ Doing node  15 codegen rule [177] root: IRETURN reg32
124:             str r10, [JFP, #-16]@ Java local cell # 0
128:             sub JSP, JFP, #12
     @ PREV (r6) = frame.prevX for return helper to use
132:             ldr r6, [JFP, #+0]@ goto CVMCCMreturnFromMethod
136:             bl  PC=(-6712)
```

In this example, the ldr instruction at logical address 64 loads local #0 from the Java frame into a register. The value is pushed onto the Java stack by the str instruction after the ldr. The str instruction also adjusts the top of the stack.

The instructions at address 88 ~ 104 correspond to node #6, `RESOLVE_REFERENCE`. The `CVMCCMruntimeResolveStaticMethodBlockAndClinit` helper function is called to resolve the method block. The resolved MB is stored at address 100 by the helper function.

The instructions at address 108 and 112 invoke the resolved method, corresponding the node 14. The method invoker is the first word of the method block. Depending the type of the method, the invoker can be `CVMCCMletInterpreterDoInvoke` (for the Java method), `CVMCCMinvokeJNIMethod` (for the JNI method), and `CVMCCMinvokeCNIMethod` (for the CNI method). If a Java method is compiled, then the invoker is changed to be the start PC (in codecache) of the compiled method. When the method returns, the result is loaded from the Java stack by the ldr instruction at 116.

20.6 Example 6

These are examples of method prologue for compiled method. When a compiled method invokes another compiled method, the callee's method prologue is used. Interpreted to compiled invocation does not go through the callee's method prologue. The prologue assumes the caller has set up the current MB (*methodblock*) as the first argument (`ARG 1`) properly before the invocation.

The synchronized non-static method's prologue is similar to the one for synchronized static method, the only difference is that they call different assembler helper functions (`CVMCCMinvokeNonstaticSyncMethodHelper` for synchronized non-static method, and `CVMCCMinvokeStaticSyncMethodHelper` for synchronized static method). The prologue for static non-synchronized method basically is the same as the one for non-static no-synchronized method.

CODE EXAMPLE 20-20 Non-static non-synchronized method

```
                           @ Method prologue
                       @ Set R1 = JSP + (capacity - argsSize) * 4
0:                     add r1, JSP, #36
                           @ Stack limit check
4:                     ldr r3, [sp, #+4]@ ccee->stackChunkEnd
8:                     str lr, [JFP, #+16]@ Store LR into frame
12:                    cmp r3, r1 LSL #0
16:                    bls PC=(-8276)@ letInterpreterDoInvoke
                           @ Set up frame for method
                       @ Store curr JFP into new frame
20:                    str JFP, [JSP, #+0]
                       @ JFP = JSP + (maxLocals - argsSize) * 4
24:                    add JFP, JSP, #0
28:                    str r0, [JFP, #+12]@ Store MB into frame
                           @ Interpreted -> compiled entry point
......                     @ Debugging only code
60:                    add JSP, JFP, #24@ spill adjust goes here
......                     @ Compiled code for the Java method
>>>>>>>>>Push Code Buffer to PC = 0 (0x40756d9c) >>>>>>>>
                           @ Capacity is 11 word(s)
0:                     add r1, JSP, #36
<<<<<<<<<Pop Code Buffer to PC = 84 (0x40756df0) <<<<<<<<<
```

```
>>>>>>>>>Push Code Buffer to PC = 60 (0x40756dd8) >>>>>>>>
                    @ spillSize is 0 word(s), add to JFP+24
60:              add JSP, JFP, #24
<<<<<<<<<Pop Code Buffer to PC = 84 (0x40756df0) <<<<<<<<<
```

This is an example of the method prologue for non-static non-synchronized method. First it checks if there is enough space for the new frame (the instructions at 0 ~ 16) in the current stack chunk. If the new frame would exceed the end of the chunk, the invocation will be handled by the interpreter (the bls instruction calls `letInterpreterDoInvoke` when there is not enough space). The str instruction at address 8 saves the return address into the current frame.

The instruction at address 20 ~ 28 set up the new frame when there is enough space. The instruction at 20 saves the current JFP into the new frame (frame->prevX). The instruction at address 24 computes the new JFP. And the instruction at address 28 stores the MB (current method block) into the frame (frame->mb). The instruction at address 60 adjust the JSP with the spill size. That is also the entry point when invoking from the interpreter. The compiled code for the Java method starts after the spill adjust.

The maximum temp words required by the compiled method is not determined when the prologue is being generated. So the instructions at address 0 and 60 are regenerated at the end of the method compilation.

CODE EXAMPLE 20-21 Static synchronized method

```
                 @ Method prologue
                 @ Set R1 = JSP + (capacity - argsSize) * 4
0:               add r1, JSP, #40
                 @ Stack limit check
4:               ldr r3, [sp, #+4]@ ccee->stackChunkEnd
8:               str lr, [JFP, #+16]@ Store LR into frame
12:              cmp r3, r1 LSL #0
16:              bls PC=(-6448)@ letInterpreterDoInvoke
                    @ Set up frame for synchronized method
                 @ NEW_JFP = JSP + (maxLocals - argsSize) * 4
20:              add r7, JSP, #4
                 @ call CVMCCMinvokeStaticSyncMethodHelper
24:              bl  PC=(-6736)
                    @ Interpreted -> compiled entry point
28:              add JSP, JFP, #24@ spill adjust goes here
......              @ Compiled code for the Java method
>>>>>>>>>Push Code Buffer to PC = 0 (0x40756678) >>>>>>>>
```

```
                     @ Capacity is 11 word(s)
0:               add r1, JSP, #40
<<<<<<<<<Pop Code Buffer to PC = 52 (0x407566ac) <<<<<<<<<
>>>>>>>>>Push Code Buffer to PC = 28 (0x40756694) >>>>>>>>
                     @ spillSize is 0 word(s), add to JFP+24
28:              add JSP, JFP, #24
<<<<<<<<<Pop Code Buffer to PC = 52 (0x407566ac) <<<<<<<<<
```

Above is an example of a static synchronized method prologue. It calls assembler helper, `CVMCCMinvokeStaticSyncMethodHelper` to do the synchronization work (the bl instruction at address 24).