C++ interpreter features

Currently, the C++ interpreter only supports the basic functionality required to implement a compliant Java VM. High end Java implementations with HotSpot use the template interpreter. The template interpreter is a very performant interpreter generated in assembly during startup of the VM. The advantage of the less performant C++ interpreter is that it requires much less assembly to be ported to a new platform, simplifying the porting.

As the VM the PPC port is derived from is targeted to long running server applications, it utilizes the optimizing C2 compiler, assuming most code is compiled at some point making interpreter performance irrelevant.To simplify porting it uses the C interpreter. Still, to get maximal performance, it supports many optimizations implemented in HotSpot, as compressed oops and biased locking. Therefore we extended the C interpreter to support these.

Compressed Oops

The compressed oops optimization reduces Java heap requirements by compressing object references.

We fixed interpretation of the aaload bytecode, where decompressing the loaded array reference was missing.

http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/304e51aad3e3

G1 garbage collection

To support G1 garbage collection, we call obj_at_put() in the aastore bytecode, instead of doing the store and barrier operations locally. This takes care of the different GC requirements.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/684e6ea70669

Biased Locking

Biased locking is an optimization that omits locks if a Java object is only used in one thread. As lock operations on PPC are rather expensive, this is a desirable optimization in the PPC port.

...

Memory Ordering

We follow two different strategies with releasing of store operations. On IA64 we use releasing store operations for all stores into the Java Heap. This guarantees that everybody using a reference sees a fully initialized object. This performs better than executing memory barriers.

On PPC, no releasing store operation is available. Thus we issue StoreStore barriers where necessarry. This is after each bytecode allocating new Java Objects. Further, initializing final fields must be released. Therefore we issue a StoreStore barrier after the return from calls.

http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/4e123d71de46

JVMTI early return

Support for JVMTI early return was missing from the C++ interpreter.

We added support for this feature, which is dual to the support of pop frame: after return from a method a pending jvmti operation has to be checked. In addition, one has to push the desired return value on the stack before returning.

Setting the interpreter return message had to be adapted, so that it is not overwritten if still needed. The pop frame message is cleared when frame manager calls the interpreter loop again with the message popping_frame.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/5db6a140bf94

Eventually have a look at the #ifndef ZERO in prims/jvmtiManageCapabilities.cpp, it might be removed.

Fixes required to support the C2 compiler

On stack replacement

HotSpot can compile a method that is stuck in a hot loop, and switch to the compiled method while the method is interpreted. This technique is called on stack replacement (OSR).

We fixed OSR support in the C interpreter, where a frame was not pushed properly.
http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/0739ece58742

To properly do OSR, the interpreter profiling change is needed, too. See below.

Interpreter Profiling

The C2 compiler relies on profiling information collected by the template interpreter. To run C2 with the C++ interpreter, it has to be extended to collect corresponding profiling data.

We implemented the profiling calls in a separate file bytecodeInterpreterProfiling.hpp. It's incorporated in the interpreter via Macros, so that the functionality can be disabled by a simple preprocessor flag. For VMs using the interpreter without the C2 compiler it's essential to remove the profiling code, as it costs considerable interpreter runtime. Currently we guard this by #ifdef PPC64, it might be better to use COMPILER2.

The profiling information collected is the same as that collected by the template interpreter.

For OSR, the interpreter counts the iterations in the innermost loop in a local variable mdo_last_branch_taken_count.

Non-TLAB allocations are forced to go to the runtime, where the profiling is performed.

We extended note_trap to pass in more information, e.g. the current bytecode index bci. We encapsulated the different exceptions in dedicated note_trap variants. To call note_trap, we added another argument to macro VM_JAVA_ERROR.

http://hg.openjdk.java.net/ppc-aix-port/jdk8/hotspot/rev/de16f32e9563