On-demand implementation

The details of this approach are explained in "Lazy Continuations for Java Virtual Machines"

Current implementation

Currently the copyStack and resumeStack functions are implemented using:

- debugging and garbage collection information about the stack frames for copying and
- deoptimization for resuming stack

Problems

This leads to a few problems:

- the gc information is incomplete (oop maps: either oop or not oop)
- the debugging stack walking code is not geared towards performance
- deoptimization is complex and can only create interpreter frames
- there's no way to determine if only a part of the stack needs to be changed to resume a continuation
  (a context argument may be passed to copyStack to create a scoped continuation, but the scoping must be provided explicitly, before any of the scoped code is executed; this is not always convenient)
- a continution will be updated even if it's only used as a nonlocal return

Client compiler implementation

To solve most, if not all, of these issues I propose a continuation implementation where continuation support is built into the client compiler.

The main idea is that continuations can be seen as a list of stack frames up to the start frame of the thread. If multiple continuations are stored as linked lists of stack frames they can be collapsed into a tree structure with the stacks start frame as the ultimate root node.
Combining this with an on-demand approach to creating the continuation frames leads to a very space and time efficient implementation.

(*Issue:* The term "stack frame object" is misleading; e.g., stack objects in EA are pseudo-objects allocated on stack. Shall we avoid using the word "stack" for heap-resident objects? We could speak of method activation frames on stack or heap.)

- The thread structure contains a pointer to the next stack frame object that needs to be filled.
- Each time a method is compiled a special continuation code block is create for each callsite that could possibly be contained in a continuation. This block simply stores the current local variables and stack elements into the stack frame object. If the current stack frame has a null "next" pointer it creates an empty stack object and links the two, otherwise a "stack address" pointer within the stack frame objects can be used to check if a new object needs to be inserted in between. This will be explained later on...

We need some data structures to store the continuations:

```
class Continuation {
  JavaThread thread_affinity;
  StackFrameOop top;
}
class StackFrame {  // or ActivationFrame
  intptr_t sp;
  oop next;
  oop restore_last;
  StackFrameDataOop data;
}
class StackFrameData {
  oop method;
  u2 bci;
  u2 local_count;
  u2 stack_count;
  u1 tags[];
  ... values[];
}
```

JavaThread needs to be enhanced with additional fields:

```
class JavaThread {
  ...
  StackFrameOop _next_stackframe = NULL;
  intptr_t _next_stackframe_sp = +inf;
  intptr_t _next_restore_stackframe_sp = 0;
  ...
}
```

Continuation::copy() looks like this:

```
  this.thread = thread;
  this.top = new StackFrame();
  this.top.sp = sp;
  this.top.next = thread._next_stackframe;  // there might already be a Stackframe object that shouldn't be
lost...
  thread._next_stackframe = this.top;
  thread._next_stackframe_sp = sp;
```

Each Callsite in every method that can be contained in a continuation is enhanced with a check:

```
  <call instruction>
  if thread._next_stackframe_sp < sp jmp continuation_block
  ...
continuation_block:
  thread._next_stackframe.data = new StackFrameData();
  fill thread._next_stackframe.data
  if thread._next_stackframe.next == NULL then
    thread._next_stackframe.next = new StackFrame();
  else
    if thread._next_stackframe.next.sp > sp then
      tmp = new StackFrame();
      tmp.next = thread._next_stackframe.next;
      thread._next_stackframe.next = tmp;
    endif
  endif
  thread._next_stackframe = thread._next_stackframe.next;
  thread._next_stackframe.sp = sp;
  thread._next_stackframe_sp = sp;
  jmp back
```

What this does:

- The one conditional jump decides if there is a continuation to be saved at all and if we're interested in the current stackframe.
- If we have to save the current stackframe: create and fill a StackframeData object
- Now the next Stackframe object is created/determined:
    - If the current StackFrame object has no "next" pointer then it is the root of the stackframe tree and we'll just create a new root
    - If there already is a "next" Stackframe we have to decide if this is already the next Stackframe object to use or if we should insert a new object in between. This decision is based on the "sp" values.
- Now that we know the next StackFrame object we'll update the thread._next_stackframe pointer.

Resuming continuations

Would be implemented by a similar conditional jump at the beginning of the methods by comparing sp with thread._next_restore_stackframe_sp.
The copy-block would at the end decide if it should destroy the current stackframe immediately.
The resume method traverses the continuation starting with the top stackframe using the "next" links. While doing this it sets the restore_last pointers that can be used to find the way back up while resuming. Continuations are bound to a thread, therefore it's no problem that only one resume-path can be stored in the stackframe tree.

Some advantages:

- While traversing, as soon as the resume method finds an emtpy StackFrame.data pointer it knows that it needs to destroy the stack only this far.
- Creating a continuation and resuming another one can be accomplished in one operation

Possible drawbacks

- thread affinity - a continuation needs to be copied if it's going to be resumed in another thread

Further suggestions

- the continution frame pointer in the thread object should be weak - if the continuation dies there's no use in still updating it
- having brought some knowledge of continuations into the client compiler this could be used to allow the compiler further optimizations on continuations.
- The code for extracting local variables into the stack frame object can be customized to the few instructions needed to spill registers and copy stack slots. Or, to simplify compilation, it could also be made a generic call that traverses the debug information (as in the current prototype). Profiling (via invocation counters) could determine when to generate optimized code for a continuation point.

Warning

This page is work-in-progress!