Threads and GC Implementation in Clozure CL

R. Matthew Emerson
Clozure Associates
rme@clozure.com

1. Introduction

Clozure CL (formerly OpenMCL) provides native threads and a precise, compacting, generational GC on all the platforms it supports.

Native threads are scheduled by the operating system, and are therefore subject to pre-emption at any instruction boundary. Because other threads can allocate memory, this means that a GC can happen at any instruction boundary.

We present some implementation techniques used in the 32-bit x86 port of Clozure CL to deal with this constraint.

2. Register Management

A precise GC believes that it always knows whether a register or stack location contains a Lisp object or just raw bits. We have no reliable way to tell the difference by simply looking at a value, so we adopt a convention: we partition the processor's registers into (at least) two sets: "immediates" (which always contain raw bits) and "nodes" (which always contain Lisp objects). This convention must not be violated, not even for a single instruction. (The GC is guaranteed to run at the most inopportune times.)

There are a few special operations, like consing, that are not atomic. For example, during the consing instruction sequence, there are several states of partial object initialization. There is special runtime support that recognizes when a thread has been interrupted in the middle of one of these pseudo-atomic operations, and performs any necessary PC-lusering.

On the PowerPC (32 registers) and x86-64 (16 registers), a static partitioning is feasible. On the 32-bit x86, we have only 8 registers, so we augment this static partitioning with a dynamic scheme: we associate a bit in thread-private memory with each register. If the bit is set, the GC treats the corresponding register as a node; if the bit is clear, the register is treated as an immediate.

As an additional option, we use the direction flag (DF) in the EFLAGS register to indicate the status of the EDX register. Clozure CL does not employ the x86 string instructions, so this flag is otherwise unused. If DF is set, EDX is treated as an immediate register, and a node otherwise (1). The thought is that manipulating this bit might be faster than accessing memory. Measurements have not yet been made to confirm or deny this notion.

This dynamic scheme seems to work reasonably well. Most operations can use the default partitioning. When the partitioning must be changed (typically to get another immediate register), the cost of doing so seems to be affordable in a larger context: the 32-bit x86 Lisp compiles itself in about the same amount of time as the x86-64 Lisp requires.

3. Stack Management

Like the registers, the Lisp stack also has to be in a GC-consistent state at all times. For instance, we are unable to say something like `(subl ($ 48) (% esp))` to reserve space on the stack for, *e.g.*, outgoing parameters. We must use the (atomic) push and pop instructions.

Another problem is the x86 CALL instruction: it automatically pushes a potentially arbitrarily-tagged return address onto the stack. Our solution to this is to give return addresses their own tag, and arrange for the CALL instructions to be aligned such that the return address will be tagged appropriately (possibly padding with no-op instructions before emitting the CALL).

In addition to keeping the GC from getting confused, these tagged return addresses act as a kind of indirect reference to their containing function.

4. Last Words

These issues are really only of concern to someone working on the compiler or the runtime of Clozure CL. The Lisp programmer does not need to take any special care to maintain GC safety in the presence of pre-emptive threads.

References

[1] Fjeld, Frode Vatvedt. Usenet article with Message-ID <2hveh7y3ip.fsf@vserver.cs.uit.no> in comp.lang.lisp, March 2007.