Overview and History of OpenMCL

- Commercial MCL was first released in 1987; ran on 1MB MacPlus (68K), rich IDE, GUI; fast compilation
- Acquired by Apple in 1989; transferred to Digitool in 1994
- MCL ported to PPC Macintosh in ~1995

History of OpenMCL

- Erann Gat of JPL wanted to develop a small-footprint Lisp for use in PPC embedded systems (rovers/robots, flight systems). After exploring other alternatives, MCL source license was acquired and MCL (sans GUI/IDE) was ported to VxWorks and LinuxPPC (which used the same ABI) in 1998.

OpenMCL at JPL

- Objections to use of Lisp within NASA/JPL were perhaps based more on cultural than technical considerations (see http://flownet.com/gat/jpl-lisp.html).

  The port was discussed at AI conferences and drew some interest, but JPL's license did not allow redistribution. This work was never used in production systems.

OpenMCL at Sandia

- In 2001, the Advanced Information Systems Laboratory at Sandia National Labs, having heard of the JPL port, expressed interest in using it in their research. I was able to persuade Digitool to open-source the work that had been done at JPL. A small consulting company formed by original MCL developers hosts the open-source project at http://openmcl.clozure.com/.

OpenMCL History: 2001-2005

- Ported to OSX/Darwin (PPC).
- Interface translation system (based on GCC frontend.)
- Most of the (quasi-standard) MetaObject Protocol (MOP)
- Native (preemptively-scheduled) threads
- “Demo” Cocoa-based GUI/IDE
- Ported to PPC64

Users

- Currently about 180 mailing list subscribers
- Server logs suggest ~5 downloads a day
- Most users seem to be using OpenMCL under OSX/Darwin as opposed to LinuxPPC (~9:1). Linux version has had problems with NPTL and TLS, to be fixed in 0.14.4.
- A guess would be around 200-300 active users, but very hard to estimate
- -Probably- many more individual users than institutional ones.

OpenMCL History 2005-

- Apple's x86 announcement
- Handling of large objects in GC/EGC, handling of very large address spaces (especially on 64-bit platforms.)
- Better (e.g., "some") lifetime analysis in the compiler (reduce floating-point consing.)
- Metering/performance tools
- Other funded work

OpenMCL technology: GC

- OpenMCL's GC is "precise" - the GC always knows whether a "root' (stack location or machine register) contains a pointer to a lisp object or an immediate value. This enables it to reliably move objects (to improve locality, improve paging behavior, etc.)

OpenMCL GC: precise vs conservative

- Precise GC (as opposed to “conservative GC”) depends on cooperation from the compiler and runtime system and strict adherence to register-usage conventions.
- “compiling to C” generally mandates conservative GC.
- Register usage conventions are easier to enforce on machines that have registers.

OpenMCL GC and threads

- For precise GC, register-usage conventions have to be followed at any time that a GC could occur.
- In general, native threads mean that a GC could occur at any time (because of activity in some other thread.) See previous bullet.
- Current GC must stop other threads.

OpenMCL GC and threads, continued.

- Newly allocated heap memory is zeroed (often by the OS.) Newly allocated stack memory may contain random bits; stack allocation has to be done carefully in some cases.
- Some operations (consing) aren't truly atomic, but we pretend that they are: the runtime system can recognize these cases and ensure that an interrupted thread is in a consistent state when the GC runs.

GC Implementation Details

- Single-space GC, compacts in place
- The (approximate) age of an object can be determined by its relative position in the heap.
- Old objects (generally) don't move; young ones generally do.
- “Ephemeral” GC depends on ability to detect the (rare?) case when old objects are destructively modified to point to new ones

More OpenMCL GC Details

- Current releases use MMU write-protection to implement the “write barrier” (invariant mentioned on previous slide.)
- Next release will use a software write barrier, and the runtime system understands how to ensure atomicity and handle contention.
- Software write-barrier offers much better granularity (8/16 bytes vs 4K bytes)

GC and Threads

- Threads allocate “segments” (~32KB), maintain current pointer in segment and low limit of segment in registers.
- Consing (allocating objects of known and small size) happens inline and takes about 6 instructions (next slide).
- If thread X is interrupted while consing, the interrupt-handling code completes or backs out of the instruction sequence.

Pseudocode for (SETQ X (CONS Y Z))

- (allocptr <- (- allocptr (- cons-size cons-tag)))
- (trap-if-less-than allocptr allocbase)
- (rplaca allocptr y)
- (rplacd allocptr z)
- (x <- allocptr)
- (allocptr <- clear-tag(allocptr))

Relocating GC implications

- Lisp object addresses can (generally) change at any point in time.
- Hash tables that hash objects by identity (address) may need to rehash when addresses change due to GC activity
- Some hash table code needs to briefly disable GC.

OpenMCL: native threads vs cooperative threads

- If N is the number of processors in a system, a program that uses more than N threads can't (generally) run faster than one that uses N or fewer (though multithreaded programs may be simpler to develop and maintain.)
  I.E, there is -some- context-switch and synchronization overhead.

OpenMCL threads: OS scheduling

- The OS ultimately decides how processor resources are allocated to threads and processes. Switching contexts between two threads in the same process (address space) is generally cheaper than switching between processes (MMU overhead).
- Applications do not (generally) have fine-grained control over OS context switch, and have little/no control over CPU allocation.

OpenMCL: cooperative threads

- It's possible to switch execution contexts (stacks/registers) in user space (this is often called “co-routining”); user-space context switching is often faster than OS context switching.
- A timer interrupt can invoke an application-level scheduler function, creating the illusion of preemptive scheduling (e.g, threads don't have to worry about exceeding time quanta.)

OpenMCL: more cooperative threads

- Cooperatively scheduled threads can't (casually) block indefinitely on a system call, since this generally prevents the application-level scheduler from running.
- The application-level scheduler generally has to poll for external events (I/O completion, synchronization events) in order to determine whether or not a thread is runnable. (This constitutes busy-waiting.)

OpenMCL: cooperative thread issues

- Latency (the time between the occurrence of an external event and the application-level scheduler noticing that occurrence and scheduling a thread that's been waiting on the event) can be very high.
- The trick of multiplexing multiple execution contexts within a single OS-level thread doesn't scale well to MP systems (where multiple OS threads may run concurrently.)

OpenMCL: cooperative threads

- It's relatively simple to inhibit scheduling of user-space threads (WITHOUT-INTERRUPTS/WITHOUT-SCHEDULING), and these idioms are often used to guard critical sections (especially when one is too lazy to use locking primitives.)

OpenMCL: native threads

- OpenMCL implementations have used native (OS-level) threads since 2003.
- User-level API is very similar to cooperative thread API, except:
  - PROCESS-WAIT is deprecated
  - WITHOUT-INTERRUPTS controls the interruptibility of current thread, doesn't affect scheduling
  - Use of locks/semaphores strongly advocated

OpenMCL: locks & semaphores

- Lisp semaphores are a very thin wrapper around OS-level semaphores.
- OS-level locks are often too heavyweight to be usable (on OSX, obtaining a lock involves 2 or 3 system calls, even when there is no contention.)
- Lisp locks use atomic memory access primitives to determine contention, only involve the OS when contention exists (Futexes)

Threads and special variables

- Shallow binding: save old value on a stack, set symbol to new value, pop old value on exit from binding construct. Access and update are unit-cost.
- Can't work when multiple threads are involved (value cell is a shared resource.)
- OpenMCL maintains per-thread local bindings; access/update constant cost (about 8 instructions.)

OpenMCL: user-level benefits of native threads

- N can be greater than 1 (threads can execute concurrently on MP systems)
- Greater interoperability with foreign code; lisp threads can “casually” block waiting for I/O or other external events
- Reduced latency associated with waiting for external events.
- Less polling/busy-waiting, better overall CPU utilization

OpenMCL: possible drawbacks of native threads

- Lack of very fine-grained control over scheduling.
- Threads and synchronization objects are “heavier” (involve the OS.)
- Code originally written to run under traditional scheduling models may require careful review. (Use of WITHOUT-INTERRUPTS is often suspect; shorthand for WITH-MORE-APROPRIATE-LOCKING.)

OpenMCL Compiler

- OpenMCL's compiler tries to generate “good” code quickly. It's often successful, but there are situations where it would need to think harder than it does (lifetime analysis, mostly) in order to generate “good” code.
- In general, floating-point objects have to be heap-allocated. It's worth trying to avoid the general case.

OpenMCL compiler: lifetimes of floats

```
(defun fsum1 (n)
  (let* ((sum 0.0d0))
    (dotimes (i n sum)
      (incf sum i))))
;;; (fsum1 1000) allocates 1000 double-floats
(defun fsum2 (n)
  (let* ((sum 0))
     (dotimes (i n (coerce sum 'double-float))
       (incf sum i))))
;;; (fsum2 1000) allocates 1 double-float
```

OpenMCL compiler: float-consing

- The examples on the preceding page are contrived, but illustrate the general case that “not all floating-point results need to be represented as lisp objects of type DOUBLE-FLOAT”. Keeping such results in FP registers or unboxed stack locations would save significant consing; when the number of unnecessary FP-consing operations gets into the millions or billions, the negative impact is significant.

OpenMCL compiler: other performance issues

- As a sweeping generalization, other performance issues that may be encountered are less likely to be endemic (and more likely to be a case of “no one's ever complained”.)
- It's hard to make (or trust) sweeping generalizations about the performance of any large system.

Other opportunities for optimization

- Leaf functions aren't recognized: about 7 instructions to build a stack frame, about 4 to tear it down. Lisp programs are said to spend a high percentage of their execution time near the leaves of the call tree.
- Many functions that aren't (purely) leaf functions have “leaf” execution paths through a toplevel IF/COND/CASE etc.

Compiler summary

- The fact that the OpenMCL compiler compiles quickly is an attractive feature to many users
- It can afford to be a little slower
- Floating-point issues don't seem to impact ACL2 much (is this true ?) but other issues may.
- It can get better and still be pretty quick

Environment & Debugging

- My excuse: “this was supposed to go in embedded systems!”. Debugging environment is spartan and what's there is often poorly documented. (Terse online help via :?)
- Everything (practically everything) is compiled; there's no useful STEP macro.
- Some other environments (SLIME, the Cocoa IDE) provide friendlier debugging facilities.

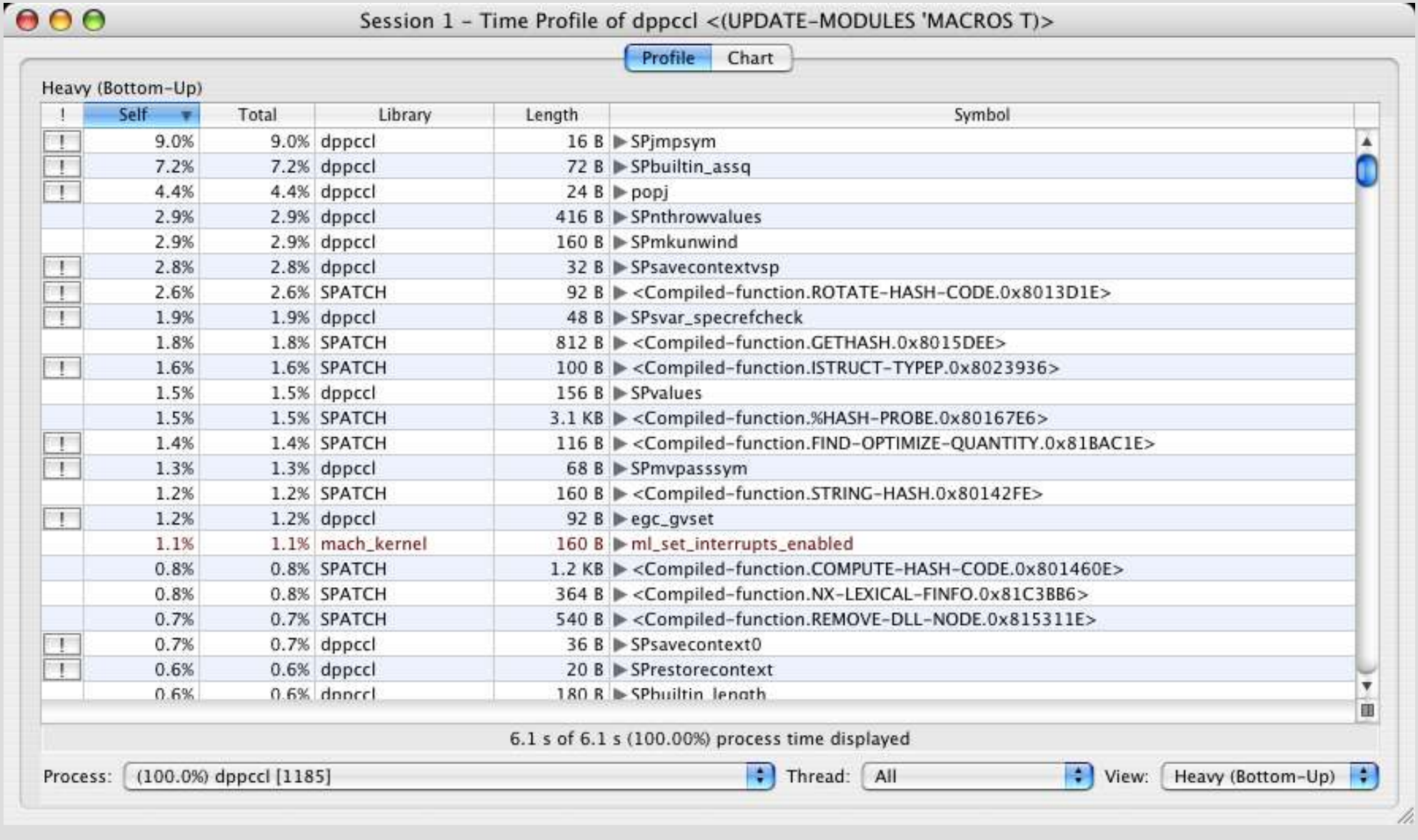
Performance tools

- Metering based on 100Hz clock interrupts often yields unsatisfactory results: the interrupts aren't guaranteed to be delivered reliably, they may be delivered to an arbitrary (and possibly uninteresting) thread, and the resolution is at least a little low.
- Apple's CHUD performance tools use OS kernel extensions to provide high-resolution timer interrupts without affecting scheduler

Performance Tools

- User-level CHUD tools (e.g., Shark) don't understand how to identify compiled Lisp functions, but (with some limitations) can be persuaded to do so.
- I may need help (or time) to make them more usable with ACL2; I've been able to identify some things (heavy use of symbol plist operators, frequent calls to EQUAL) that may not have been apparent otherwise.

Shark profiling (when it works)



Build process

- Show of hands: has anyone in audience built OpenMCL ?
- Lisp implemented as a kernel (written in C and PPC assembler) and a heap image. GC, exception handling code in C; C code only runs when lisp code can't.
- Build process is circular: need a heap image to build a heap image.
- Recompiling all lisp sources and rebuilding an image takes a few minutes.

Development process

- Two CVS trees, “main” and “bleeding-edge”.
- Bleeding-edge tree often contains experimental/new features, may not build with released image. (Usually, new images are available in /testing directory on openmcl.clozure.com.)
- Releases have not been as regular as one might like; about 10 months between 0.14.2 and 0.14.3, maybe 5 months for 0.14.4

Development process, continued.

- Bugs are often fixed in development tree, fixes don't always make it into patch releases. (Often, there are technical reasons for this, involving ABI changes, but not always.)
- Need to try to commit to regular release schedule (2 months ? 3 months ?)
- Releases take time to put together.

Documentation and Resources

- Online documentation is probably fairly accurate, but not extensive.
- Mailing list is medium-low volume, usually very good signal-to-noise ratio.
- Clozure (the consulting company) is starting to offer commercial support options (details TBD.)
- Clozure has a contract to provide support for ACL2 (not sure how broad; ask Warren.)

Development Model

- It -is- an open-source project; other people do contribute, and hopefully more people will be encouraged to do so.
- “internals” document on website is about 5 years old (predates native threads); maybe 60% accurate.
- Clozure is seeking funding for the Intel port and other work.