Rosetta-ized MCL ("RMCL") notes.

May 20, 2009

1 Background

Macintosh Common Lisp (MCL) is a PPC application; like many other CL implementations, it depends on OS-level exception-handling facilities (to deal with many lisp-level error conditions, to support fast memory allocation and GC integration, to detect stack overflow) as part of its normal execution model. Apple's Rosetta translator does a very good job of allowing many PPC applications to run on x86 hardware at acceptable speeds; one way in which it achieves good execution speed for translated code interferes with the ability of translated applications to reliably handle machine exceptions that those applications generate. As a result, MCL has been unable to run under Rosetta (it dies as soon as it generates an exception, which occurs very soon after it's launched.)

A Clozure customer contracted with us to port a legacy MCL application to modern (x86-based) Macintosh hardware. Our original approach to this was to try to provide an MCL-like compatibility layer in CCL, but for a variety of reasons (including differences between MCL's and CCL's threading models and FFIs and the degree to which the customer's application depended on MCL internals) we began to have doubts about the viability of that approach. Another idea - changing MCL so that it would run under Rosetta, which we'd previously dismissed as being too time-consuming - began to look more attractive. Eventually, we decided to switch to Plan B (eliminating MCL's use of machine-level exceptions so that it would run under Rosetta). The result seems to have been successful in that it allows our customer (and their customers) to run their legacy MCL application on x86 Macintosh hardware. Since MCL is licenced under the LLGPL, the results of that work (with full sources) are provided here.

Since the goal of the project was to support the needs of a specific application, we made some choices that may seem surprising at first glance. In particular, we chose to use MCL 5.1b1 sources that I (gb) happened to have installed on an old machine, leftover from a time when I was working with Digitool to get MCL running on OSX. This choice was influenced by the facts that our customer's application was known not to run on MCL 5.2 and that the 5.1b1 sources that I had were known to build a working MCL while it was not clear that that was true of the released 5.2 sources. We later merged in some sources

that Digitool had released with 5.1 final; the result is that RMCL is a weird hybrid of 5.1b1 and 5.1, and some things may differ in behavior from anything that Digitool released.

2 Build process

2.1 Kernel issues

MCL (including 5.2) is a CFM[1] application; this is true regardless of whether its FFI exposes functionality defined in CFM libraries and called via CFM conventions or whether its FFI exposes functionality defined in native, Mach-O libraries. On OSX, CFM applications are loaded into memory by a native helper application ("LaunchCFMApp", which ordinarily resides in /System/Library/Frameworks/Carbon.framework/Versions/A/Support/). LaunchCFMApp tries to find external CFM libraries that the CFM application is linked against and resolve references to symbols defined in those libraries; if it's successful, it transfers control to initialization and startup routines in the CFM executable.

MCL refers to a CFM library named "pmcl-kernel", and its initialization and startup routines call functions that're defined in that library. On traditional MacOS, this file was "the kernel" and provided image loading and saving, exception handling, and other services for the MCL application. On OSX, a separate copy of the C kernel sources were compiled into a "native" kernel (compiled and linked as a native Mach-O object file and named "pmcl-OSX-kernel"), and the fact that this native kernel could communicate with OSX more directly made some things more reliable (though popular wisdom was that it could also have the opposite effect); in that scheme, the CFM kernel was redundant if the native kernel could be found and loaded, and the native kernel was (to some degree) optional. In RMCL, the CFM kernel exists solely to load the native kernel and doesn't offer any other functionality, and the native kernel is mandatory. (The CFM kernel also contains some other code and other resources, including the "subprimitives"[2].)

2.1.1 Building the CFM kernel

The little bit of C code in the CFM kernel (and the subprimitives and some of the other resources) still need to be built with Apple's MPW tools, which are available from `http://developer.apple.com/tools/mpw-tools/`. MPW is a "Classic" PPC application and as such will only run on a PPC running Tiger or earlier (on which the Classic environment and its dependencies are installed.) At one point, the C sources to the "CFM" kernel had to be compiled with a certain version of Stan Shebs' port of GCC to MPW. That version of GCC for MPW doesn't seem to be available anymore; fortunately, there doesn't need to

[1] The shared library technology used in Classic MacOS was called the Code Fragment Manager.

[2] compiler support routines that are written in assembly language

be much C code in the "CFM" kernel anymore, and the one file that remains ("pmcl/kernel-init.c") can be compiled with the MPW mrc compiler.

To build the CFM kernel under MPW:

1. Launch the MPW Shell
2. Set MPW's working directory to RMCL's "pmcl" subdirectory; this can be done via an entry on MPW's Directory menu.
3. Type CMD-B to invoke the "Build..." menu item on MPW's "Build" menu; in the dialog box which results, type "install" and confirm. This should execute any commands needed to build "pmcl-kernel" from its sources and install a copy of the newly built CFM kernel in the root RMCL directory. The installation step may fail if the file's in use, e.g., if a running copy of RMCL is using the library.

2.1.2 Building the native kernel

The native kernel sources are in pmcl/OSX; they can be built with any version of Apple's Xcode tools that target the PPC; the host system can be a PPC or x86-based Macintosh. To build the native kernel in the OSX shell:

1. cd to the pcml/OSX directory
2. execute the command "make install"

That'll rebuild the "pmcl-OSX-kernel" library from its sources and copy the result to the root RMCL directory.

2.2 Building the application

RMCL itself (the file with the coral icon that one double-clicks on in the Finder) is a tiny CFM application (that jumps into some entrypoints in pmcl-kernel) and a larger heap image, both contained in the same file.

Two very similar versions of the RMCL application are included in the distribution. "RMCL" itself is intended for daily use; "PPCCL" has essentially the same contents, only it starts up in the CCL package and the features which try to guard against accidental redefinition of built-in functions are disabled; PPCCL is intended to be used (primarily) for recompiling the lisp itself, since both of these differences make that a bit easier.

In order to build a new version of RMCL (or PPCCL), it's necessary to compile most of its source code into FASL files, build a special bootstrapping application ("ppc-boot") from other source code (in the "level-0" subdirectory), run that bootstrapping application which will load enough of the FASL files to display a listener and print a prompt, load a file which causes the rest of the FASL files to load, optionally load a file which arranges for *PACKAGE* to be set to the CL-USER package and enables the anti-redefinition mechanisms (for RMCL), and use SAVE-APPLICATION to create a new PPCCL or RMCL. The

astute reader will note that there's no easy way to compile those FASL files or create the bootstrapping application without a (generally compatible) PPCCL application; anyone who thinks that this kind of chicken-and-egg situation is at all unusual is invited to try to build GCC from source on a machine that doesn't have a working C compiler.

2.2.1 Compiling the sources

Launch PPCCL and do:

```
? (compile-ccl t)
```

That'll compile several dozen source files and in some cases load the resulting FASL files into the compilation environment. (Some files, especially those in the level-1 subdirectory, are explicitly not loaded into the compilation environment.) There may be several warnings about functions and methods being defined multiple times in multiple files (these are expected parts of the [R]MCL bootstrapping process) and there are currently a couple of warnings describing calls to undefined functions: one of them has to do with saving lisp libraries and ... well, I don't remember what the other one's about, but it was present in the random 5.1b1 sources that I started with.

This process will take a minute or two (depending on how fast the machine is) and should finish without errors; if you get an error during this stage, figure out why, fix it, and try again.

2.2.2 Building the bootstrapping image

In a PPCCL (you can do this before or after the previous step, or in a fresh session), do:

```
? (xload-level-0 :force)
```

That'll compile a few dozen files in the "level-0" directory, claim that it's "loading" them (it's actually loading them into a special in-memory heap image), and eventually write a small application named "ppc-boot" to the root RMCL directory. There may be some cryptic messages during the emulated load stage that claim that the plist of a symbol at some hex address is already set; these messages generally indicate that some function or variable is defined multiple times in the level-0 sources.

Because of the weird way that the "ppc-boot" application is built, the Finder may become convinced that it's a Classic application (it isn't) or that it's compiled for an unsupported architecture (also not true.) Logging out and logging back in again seems to be one way of getting the Finder to take a closer look at the file and reevaluate things.

2.2.3 Running the bootstrapping image

Double-click on "ppc-boot" in the Finder (see the last paragraph if that doesn't work.) After a few seconds, MCL's menubar should appear and shortly after

that, a listener window should appear. To get to this stage, several FASL files have to have been loaded successfully, but (for historical reasons whose details I don't remember) several other FASL files still haven't been loaded into the running lisp: some CL functionality isn't yet defined, some editor commands and IDE features may not work yet, errors may or may not be handled gracefully. To load the remaining files, do:

? (require "PPC-INIT-CCL")

If you want to save an "RMCL" (intended for CL development, starts in the CL-USER package, guards against accidental redefinition of built-in functions), do:

? (require "PREPARE-MCL-ENVIRONMENT")

then

? (save-application "RMCL")

If you instead want to save a "PPCCL" (intended for RMCL development, starts in the CCL package, allows redefinition of built-in functions), do:

? (save-application "PPCCL")

2.3 Build-time problems

A lot of things have to work correctly in order for any the bootstrapping program ("ppc-boot") to be able to display a listener window, respond to keystrokes and other events, and generally behave itself; still more things need to work correctly in order for any version of MCL to be able to report errors sanely and enter a break loop. The bootstrapping application starts out as a fairly small subset of a full lisp and becomes a larger subset as files are loaded; code that needs to run early in that loading sequence can't depend on functionality that's defined later in the sequence and may therefore have to be written in a subset of CL. Errors that occur early in the bootstrapping process may cause spectacular failures and can be very difficult to debug.

In MCL, some errors that occurred early in the bootstrapping process (aka "the cold load") triggered exceptions; if the kernel could recognize that the lisp wasn't ready to handle the exception yet, a simple kernel debugger was entered. Since RMCL can't use exceptions, this can't work. The only way that I know of to debug problems that occur during this bootstrapping process is to use GDB, and describing how to do that is way beyond the scope of this document. Given the fragility of the cold load process, the best advice that can be offered is to make changes to core MCL functionality as incrementally as possible (and rebuild as often as possible during development) so that the likely cause of a build failure can be easily isolated.

3 Known differences between RMCL and MCL

Some errors that may have offered restarts in MCL may not offer those restarts in RMCL. The wording of some error messages may be different. (Both of these changes have to do with the fact that traps and illegal instructions are no longer used to signal errors in RMCL.)

MCL's stacks are (in some cases) "segmented" (composed of relatively small chunks which are linked together and which may transparently overflow when a write-protected guard page is written to and the resulting exception is handled.) A stack overflow exception is signaled when the total size of all allocated segments in a stack exceeds a given threshold; it's generally possible to continue from this point with a new, larger threshold in effect. By contrast, RMCL's stacks consist of a single generally much larger chunk, overflow on them is detected in software, and it's not possible to continue a computation with more stack spave after overflow has been detected. (There should be enough stack space remaining after overflow has been signaled to run a backtrace dialog to try to determine the cause of non-terminating recursion, if that's what caused the overflow.)

4 Rosetta issues

We found (only) one outright Rosetta bug during the porting process. (On real PPC hardware, "blr" and "bctr" instructions ignore the low two bits of the link register (lr) and count register (ctr), respectively. MCL exploited this; RMCL clears those bits in software, because Rosetta sometimes failed to ignore them.)

Rosetta likes to use the OS's disk cache to cache translated versions of PPC functions; OSX likes to use otherwise unused physical memory for its disk cache. Having more physical memory than running applications need can therefore improve the performance of Rosetta applications.

In most cases, using exceptions to handle exceptional situations is cheaper than doing that in software; the software-only approach may have to carry more context around in general in order to be able to handle the exceptional case via a function call, while the exception-based approach may be able to reconstruct that context in cases where an exception was actually taken. There are of course ... um .. exceptions to this general rule: there's much more overhead associated with handling write-protect faults on OSX than on other OSes, and MCL made heavy use of write-protect faults (intentionally generated lots of them) as part of its normal execution model. Avoiding intentional write-protect faults in RMCL likely sped some things up; having to pass some extra arguments around or do extra subroutine calls instead of conditional traps likely slowed some things in RMCL down relative to MCL, and it's hard to say anything definitive about whether RMCL or MCL is faster in general when running natively on a PPC. (If there's a difference, it probably depends a lot on the code in question.)

It's hard to know how this affects performance of translated code running under Rosetta, or (aside from the disk cache issue) what things do and do not

affect Rosetta performance.