Porting CCL to the ARMv8: An initial scouting expedition

Ayla Kurdak
September 5, 2022 (18:51)

Overview

Over the course of this summer, we studied the implementation details of the Clozure Common Lisp (CCL) compiler and runtime system First we explored the CCL sources, then we compiled key Lisp examples, and analyzed the disassembled code emitted from the x86-64 compiler ("x86" for short). After getting a lay of the land, we built simulators for the x86, 32-bit ARMv7, and 64-bit ARMv8. These simulators were used to execute the result of running two examples. The first was a function that calculates integer a^b. This example demonstrates basic fixnum arithmetic, recursion, and calling a subprimitive. The second was a function that creates a list of two elements. This example demonstrates inline code for doing memory allocation. Building the x86 and ARMv7 simulators gave us a directioned approach to revealing details of the existing Lisp implementations, ensured accuracy in our understanding of the LAP code (in Lisp Assembly Program syntax), and provided a stepping stone to the ARMv8 simulator and LAP code translation. By hand-writing equivalent ARMv8 code, we made first steps towards porting CCL to the ARMv8. As a final step, we translated our LAP code to GNU 'as' assembly language and tested it on ARMv8 hardware.

A note on the x86 sources

Two files were particularly useful in our exploration, and we referred to them frequently as we built the simulators. These are the files **x86-constants64.s** and **x86-spentry64.s**.

The *constants* file defines tags, subtypes, object structures, register aliases, and the fields of the TCR (Thread Context Record). The *spentry* file defines the subprimitives.

Subprimitives

Subprimitves are used to perform operations more complicated than those that typically open-coded. For example, *addition* is open coded in the simple case where the operands are fixnums and in which no overflow occurs, but it calls out to a subprim otherwise. The subprims themselves call out to "built-ins" in the most general case. While the subprims are written in assembly, these built-ins are Lisp functions that can handle things like arithmetic on non-fixnums. The subprim jump table is defined statically in **x86-spjump64.s** for the x86 and at the end of the TCR for ARMv7. We relied on the x86 and ARMv7 spentry definition of *builtin-times* to write the ARMv8 version called by the *my-expt* example function.

Motivation for the simulators

After doing line-by-line annotations of the x86 LAP code, we decided that having a simulator that could automatically read in the disassembly and perform the computation would not only be a valuable tool for checking the correctness of handwritten ARMv8 LAP code, but by writing it we would gain a more intricate understanding of the existing x86 LAP code. Since our priority was clarity and readability over machine-level accuracy, we took a symbolic programming approach that used the Lisp macro-expander as a LAP-to-Lisp translator, which made following and debugging the simulator's actions simpler. For example, we could directly insert Lisp code for debugging.

Once the x86 simulator worked, we dove straight into an ARMv8 simulator, but quickly ran into roadblocks. Since the two simulators would share many of the same functions but also needed architecture-specific pieces, we needed to create separate Lisp packages to keep the architectures separate. Furthermore, while we had disassembler-generated LAP code to use for the x86, for the ARMv8 we were inventing new code. Being new to ARM assembly language coding style, we decided to write an ARMv7 simulator as a stepping stone. Though ARMv7 and ARMv8 have differences in their assembly language and architecture (ARMv7 is of course 32-bit and features *conditional execution* of many instructions), writing this simulator gave us a better sense of how to proceed with the ARMv8.

The ARMv8 simulator and LAP code

The ARMv8 simulator proposes some core design choices and envisions what the LAP code will look like. It is closely aligned with the x86, using the same tagging scheme and keeping the format of objects in memory similar. There are subtle differences in function calls, notably the way it sets up a new function. In x86, the function register **fn** is set using the program counter, while in ARMv8, we have a new function register called **nfn**. The presence of a link register on the ARM and the use of the *value stack* are other notable differences. The LAP code was written by referencing both the x86 and ARMv7.

Design decisions in the ARMv8 and differences from x86

Given limited resources and manpower, it is important that porting to ARMv8 be as simple as possible. The goal is to make as few changes from the x86 as possible, thus allowing us to adapt rather than rewrite the runtime system and especially the garbage collector. Though this may mean we do not take full advantage of ARM specific properties (e.g. that the top 8 bits of addresses are ignored), we believe this loss is outweighed by the workload benefits. We made our design decisions with this principle in mind.

Low tags v High Tags

There has been some discussion of switching to a "high tags" system on the ARMv8, since the high 8 bits of pointers would be ignored by the hardware, and there would be no need to shift

fixnums before and after certain types of arithmetic. However, all other CCLs have used low tags, and introducing a fundamental data representation change for the ARMv8 port will complicate future work and in particular would require a *major* overhaul of the garbage collector. Therefore, we have built our simulator with the assumption and *recommendation* that the ARMv8 CCL will continue to use a low tagging scheme that matches the x86.

Function calls

Below is an example of x86 and ARMv8 callers. These are snippets from the function call-list2, and show how the caller prepares for and makes the call.

x86:

```
;; call-list2
    (movl ($ 8) (% arg-y.l))            ; 1
    (movl ($ 16) (% arg-z.l))           ; 2
    (movl ($ 16) (% nargs))             ; 2
    (movq (@ 'LIST2 (% fn)) (% temp0))  ; temp0 = list2 symbol structure address
    (lisp-call (@ 10 (% temp0)))        ; function call to list2 function cell
```

ARMv8:

```
;; call-list2
    (mov (% arg-y.w) ($ 8))   ; 1
    (mov (% arg-z.w) ($ 16))  ; 2
    (mov (% nargs.w) ($ 16))  ; 2
    (ldr (% fname) (@ (% fn) 'list2))  ; fname = list2 symbol structure address
                                       ; (loaded from fn's constants vector)
    (ldr (% nfn) (@ (% fname) ($ 10))) ; nfn = pointer to function object
    (ldr (% nfn) (@ (% nfn)  ($ fn-tag-offset)) ; nfn = code vector pointer
    (blr (% nfn))              ; function call (branch to register and set lr)
```

Overall, the two approaches are similar. They both set up the arguments, link through the callee's **symbol-function** (the symbol is found in the current function's *constants vector*), then branch to the callee's *code vector* via the symbol's *function object*. The first notable difference is the presence of the **nfn** register on the ARM which points to the new function and will also be used by the callee to set **fn**. Also of note is the way the call itself happens. On the x86, the call is performed by the *lapmacro* **lisp-call** which uses the x86 **call** instruction. The call instruction saves the return address on the stack. On the ARM, the call is slightly different, performed by the instruction **blr** (branch with link to register), which saves the return address in **lr** (the *link register*). For the x86, a return instruction branches to the address on the top of the stack, while the ARM return instruction branches to the address in **lr**. Also note that Lisp tail-calls are performed by **jmpq** on the x86 and by **br** on the ARM, which do not save the return address.

Below is a snippet from the corresponding callee LAP code.

x86:

```
;; list2
    (recover-fn-from-rip)     ; lapmacro that sets fn
    (cmpl ($ 16) (% nargs))   ; check nargs
    (jne L137)                ;
    (pushq (% rbp))           ; save frame pointer
    (movq (% rsp) (% rbp))    ; new function frame
    (pushq (% arg-y))         ; push args to new frame
    (pushq (% arg-z))         ;
```

ARMv8:

```
;; list2
    (cmp (% nargs) ($ 16))       ; check nargs
    (beq L137)
    (uuo-error-wrong-nargs)

    L137
    (push1 (% fp))               ; save frame pointer
    (mov (% fp) (% sp))          ; new function frame
    (push1 (% vsp))              ; save current vsp
    (push2 (% fn) (% lr))        ; save current fn and lr
    (mov (% fn) (% nfn))         ; fn = new function
    (vpush2 (% arg-z) (% arg-y)) ; push args to value stack
```

The macro **recover-fn-from-rip** uses the program counter (**rip**) to load the address of the function into **fn**. The x86 is unique among the architectures in that its instructions are of variable width and can't be parsed backwards by the garbage collector, a property which necessitates the use of this macro. The ARM loads the code vector address into **fn** simply by moving the address in **nfn** into **fn**.

After checking the number of arguments, both architectures create new stack frames. On the ARM, the new frame has the original value stack pointer (**vsp**), the address of the previous function (**fn**), and the link register (**lr**).

Here is an example of what a function frame would look like in simulated memory (note that the stack grows down):

```
----- end of frame
131000: 131032       ; FP
131008: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x302001AAB97F> ; FN
131016: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x302001AAB87F> ; LR
131024: 8192          ; VSP
----- end of frame
131032: 131064        ; FP
... stack continues below
```

A quick digression: We used a convoluted technique to implement our simulator. Instead of storing code in the simulated memory array and using a straightforward branching scheme, we "compiled" LAP code into a giant Lisp **tagbody** and used Lisp **go** for branching. For return addresses, the code used "go thunks," lexical closures of the form `(lambda () (go label))`, and these are the closures you see on the simulated stack in the example above.

While both the x86 and ARM have a control stack, value stack, and temporary stack, they use them in very different ways. On the x86, **rsp** points to the *value stack* whenever Lisp code is running and does not use the *control stack*. As such, the value stack behaves sort of like a combined control and value stack, holding both the function frame and any arguments. Like the ARMv7, the stacks on the ARMv8 will be used as their names imply, with *separate* value and control stacks. The control stack saves the function's *stack frame* while the value stack saves any *arguments*.

Using the simulators

The ARMv8 simulator was an especially useful tool for testing the handwritten LAP code. It should be noted that creating working LAP code in the simulator does not guarantee correctness on the actual target machine, as we made significant simplifications (for example, we only model register data widths when needed to preserve Lisp fixnum semantics) and we made assumptions about what memory will look like, which may need to be reevaluated. We addressed these limitations in the final phase of our project where we ran our assembly code on real hardware. Nevertheless, the simulator is a *practical* representation of the ARMv8 with the benefits of readability, minimal overhead, and simple debugging. We believe this simulator could continue to be a valuable tool through the next phase of this project, especially during the translation of *subprims*.

Writing LAP code functions

LAP code for new functions and subprims can be written using the macros **def-asm-lisp-fun** and **def-asm-fun** respectively. To put these functions and subprims together into a code block, we use the **def-simulator-code** macro which expands all the functions into a single **tagbody**.

```
(def-simulator-code expt-top-level
  :LISPFUNS (expt-caller-2-3
             my-expt)
  :BUILTINS (.spbuiltin-times
             ;; Below: unused for now, but referenced
             .spbuiltin-eq
             .spfix-overflow
             .spbuiltin-minus
             -spmakes128
             ;;-spmakes64
            ))
```

This is how to run an example from the REPL.

```
CL-USER> (sav8::expt-top-level)
8
```

Three things make for easy debugging with the simulator: 1) heavy use of macroexpansion, 2) the ability to insert print and break statements into the LAP code, and 3) simple helper functions **show-registers** and **show-memory** to see inside registers and memory.

Since an entire program macroexpands into a **tagbody**, we can see inside a whole group of functions at once, in addition to looking inside individual assembly language instructions. Though this was especially helpful during the beginning stages of constructing the simulator, it remains useful for making sure that new ARM instructions behave as expected. The requirement that the entire LAP example (including subprims) become a single **tagbody** required a somewhat complex underlying simulator design (this complexity can be seen in functions like **macroexpand-lap-tree** and **flatten-lap-tree** found in **simulator-common.lisp**).

As mentioned, one of the strengths of the simulator is the readability of the registers and memory. We aimed to make our simulated memory simple to understand, sometimes at the expense of accuracy. For instance, in the x86 simulator, we created (simulated) strings for symbol names. These appear character by character in memory, separate from the symbol structure itself. We changed this in the later simulators because we felt the loss of readability outweighed the improvements in accuracy.

Below is an example of the contents of simulated memory that was dumped by show-memory and annotated.

```
------ symbol ------
0: sym-header
8: CALL-LIST2 ;simulated symbol name as a Lisp symbol for readability
16: 77835  ; pointer to nil
24: 205    ; pointer to function object
32: 77835
40: 77835
48: 77835
56: 77835
------ symbol ------
64: sym-header
72: LIST2
80: 77835
88: 237    ; pointer to function object
96: 77835
104: 77835
112: 77835
120: 77835
```

```
------ symbol ------
128: sym-header
136: VERIFY-LIST2
144: 77835
152: 253    ; pointer to function object
160: 77835
168: 77835
176: 77835
184: 77835
------ call-list2 function object ------
192: fn-header
200: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x30200166B56F>
208: 78    ; list-2
216: 142   ; verify-list2
------ list2 function object ------
224: fn-header
232: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x30200166B52F>
------ verify-list2 function object ------
240: fn-header
248: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x30200166B4EF>
------ value stack ------
20456: 16
20464: 8
20472: 24547
------ (1 2) ------
24544: 24563  ; cdr = (2)
24552: 8      ; car = 1
24560: 77835  ; cdr = nil
24568: 16     ; car = 2
------ TCR ------
65536: The TCR
65752: 24576
65760: 24576
----- NIL and T -----
77824: NIL-header
77832: NIL
77840: 77835
77848: 77835    ; one unresolved issue: T overwrites the end of NIL
77856: T-header
77864: T
77872: 77870
77880: 77835
77888: 77835
77896: 77835
77904: 77835
77912: 77835
```

```
----- control stack -----
131008: 205
131016: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x30200166B46F>
131024: 20480
131032: 131064
131040: FN
131048: #<COMPILED-LEXICAL-CLOSURE (INTERNAL LIST-TOP-LEVEL) #x30200166B4AF>
131056: 20480
131064: FP
```

From simulator to real hardware

As previously mentioned, the simulators did not guarantee LAP code correctness. To address this, in the final phase of our project, we converted our LAP code to GNU ‘as’ assembly syntax, and ran the resulting **.s** file on a real ARMv8 machine (a Raspberry Pi 400). The translator can be found in the file **lap-to-as-translator.lisp**. It is called by each LAP function and prints the function body in assembly code.

In addition to translating the LAP code, we needed to initialize memory with symbols, functions, T, NIL, and the TCR. In our simulator, this is done at compile time by the function **init-simulator**. We wrote the function **dump-memory-to-as** which reads simulated memory after initialization and prints its contents in ‘as’ assembly language syntax.

Using our automated syntax translator and memory dumper, we created and ran an assembly file for our exponentiation example which takes two arguments and calculates a^b as follows.

```
pi@raspberrypi:~$ ./expt 2 3
pi@raspberrypi:~$ echo $?
8
```

Next steps

Since our approach was one of depth over breadth, many aspects of the x86 compiler and runtime system remain unexplored. We stuck with two simple examples that used minimal subprims and didn’t call out to Lisp built-ins. We did not consider Lisp features like **catch** and **throw**, complex argument lists, or *multiple values* (aside: on the ARMv8, register **x8** which is designated for *indirect results* (e.g. for returning a C++ *struct*) might be appropriate for implementing Lisp multiple values). We believe our simulators could be helpful resources for understanding, writing, and debugging LAP code, especially when it comes to writing ARMv8 subprims. Our architecture simulators were deliberately limited in scope to execute only the few examples we ran, but new machine instructions can easily be added as needed.

Our concluding recommendation is to resist the temptation to re-architect the CCL kernel (e.g. implement high tags) and to make the ARMv8 port match the x86 kernel as faithfully as

possible, so we can heavily draw from the existing runtime system and garbage collector rather than rewrite the whole thing.

Credits

I would like to thank my project advisor Tim McNerney for his constant guidance while I studied the details of CCL's x86 and ARMv7 compiled code and kernel architecture, worked on the simulators, and hand-wrote the ARMv8 LAP examples. It was his idea to create a LAP simulator as an analysis tool, and without this suggestion, the project would not have been what it is. Creating the simulators gave me an interactive tool with which to dive into the most intricate details of CCL's implementation, and the project would have been a lot harder and less rewarding had I spent the summer staring at compiler output. He accepts all blame (his words) for the more convoluted architectural details of the LAP simulator, like writing macros that explicitly run **macroexpand** (and recursively, no less), and the whole notion of using a single, giant **tagbody** and "go thunks" like `(lambda () (go` *continuation*`))`, which functions as a "return address."

Appendix

To build and run the Lisp simulator, LAP examples, and LAP-to-gnu-as translator, modify the `sysdcl.lisp` file to reflect your local file system structure, load this file into CCL, then run

```
(make-system 'simulator)
```

in the `CL-USER` package. As a *side-effect* of compilation and loading, this will run the ARMv8 LAP examples and emit the GNU **as** .s (assembly language) code *onto the Lisp console*.

To run the assembly language on an actual ARMv8 under Linux,(*) cut and paste this into a file, then update the `.file` directive to match the target location of this file (**gdb** cares about this for setting breakpoints). Finally, move this file to ARMv8 Linux target (e.g. a Raspberry Pi 400 *with 64-bit* OS), along with C files, `expt.c` and `list2.c`. On the target system, *assuming* you named the emitted file `both-rc14.s`, type the following shell command to build:

```
gcc -g both-rc14.s expt.c -o expt
```

To run the exponentiation calculator, and see its result requires two shell commands:

```
./expt 5 3
echo $?
```

(A simple exercise for the eager reader: modify `expt.c` to use *printf* so results can be >255.)
To compile the `call-list2` example: `gcc -g both-rc14.s list2.c -o list2`
The result of this example is only the type bits of the result: 14 for a symbol (T), 11 for NIL.
(*) This will probably work under MacOS Xcode *command-line tools*. We haven't tested this.

Files

Size (bytes)	File Name	Description
1907	`sysdcl.lisp`	Dependencies and trivial make-system
1631	`pkgdcl.lisp`	Package declarations
12261	`simulator-common.lisp`	Simulator utilities, e.g. `flatten-lap-tree`
24307	`simulator-x86.lisp`	x86-64 simulator
27212	`lap-tests-x86.lisp`	Examples from CCL x86-64 compiler
22401	`simulator-arm32.lisp`	32-bit ARM simulator
9070	`lap-tests-arm32.lisp`	Examples from CCL ARM compiler
39770	`simulator-arm64.lisp`	64-bit ARMv8 simulator
18149	`lap-tests-arm64.lisp`	*Handwritten* ARMv8 examples
11147	`lap-to-as-translator.lisp`	LAP to GNU `as` syntax translator
14549	`both-rc14.s`	*Pre-built* assembly file for GNU `as`
501	`expt.c`	x^y calculator, use `my-expt` in `both-rc14.s`
276	`list2.c`	Test `call-list2` example and verifier –"–
822	`test-call-return-via-struct.c`	Independent test code to see how GNU `binutils` allocates memory and `gcc` code can call a function through a struct slot