Hardware Support for Poly and ML

David C.J. Matthews

So far Poly has been implemented on the Vax and the Sun (MC68020) using native code. There is also a version which generates a byte code that can be interpreted. Even this version, however, was intended as an aid to porting the system from the Vax to the Sun and was designed to be very similar to the native code versions. It would be a very interesting exercise to design an architecture specifically for Poly and ML and to see how would differ from a conventional machine. The rest of this note proposes a possible architecture based on the abstract machine used in the byte-code system.

The architecture was developed from a code-generator for a conventional machine so the instructions are fairly low level. It would be possible to provide instructions which would be closer to the original Poly or ML code, such as an instruction to extract all the values from a tuple onto the stack as a single operation, but there are advantages, particular in the machine-independent optimiser, of having a simple instruction set. It is always open to the designer of the hardware to provide additional instructions which implement a combination of the primitive instructions, and indeed that would probably be necessary to achieve reasonable performance. That should be done using data about the occurrences of particular instructions and operands in practice.

The basic machine is intended to be extremely simple, with few registers, no complicated addressing modes and relying heavily on a second processor for many of its operations. This would allow the machine to be developed by stages, with an initial version supporting only the basic instructions, but later versions performing more of the functions of the second processor.

Finally there are certain operations which are possible with specialised hardware which are not possible on a conventional computer, particularly to do with virtual memory and asynchronous garbage collection. Some suggestions along these lines have been made but there are no specific proposals.

1 Overview

The abstract machine can be thought of as two separate processors with access to a common memory. It might be implemented as one or several real processors. The *primary* processor has a small set of primitive instructions to do operations such as jumps, function calls and loads. The *secondary* processor is activated to deal with virtual address faults and to perform various *primitive functions.* It is also responsible for garbage collection. The *primitive functions* are operations that behave like functions in Poly or ML but cannot be written using the instructions of the primary processor. They extend from simple operations like adding together two integers through to complex operations like opening a file. Many of the simpler operations might be handled for efficiency reasons by the primary processor but there are advantages in, conceptually at any rate, dividing the operations up in this way.

2 The Primary Processor

The primary processor is a stack machine with a single address space. Each process has its own stack segment. The stack pointer, *sp,* points to the top entry on the stack which is used for instruction operands, function parameters and results, return addresses and for stacking exceptions. The program counter, *pc,* points to the current instruction in the code. The handler register, *hr,* points into the stack at the current exception entry. The stack base register, *sb,* points to the base of the current stack segment. The temporary register, *t,* is only used during exception handling. Store is allocated from a contiguous heap segment pointed to by the allocation register, *ar,* with the limit of the heap indicated by the heap limit register, *lr.*

The instructions of the primary processor are fairly limited (Table 2).

3 Secondary Processor

The secondary processor is responsible for input/output and handling virtual memory faults. Many of its functions might well be implemented in the same physical processor as the primary processor. Table 3 gives a list of the functions currently implemented in the Poly system.

Apart from *nullorzero* which is a null string, all these entries are functions. The arguments are popped from the stack, and any result returned on the stack. To

jump	Unconditional jump.
jump_false	Pop the top of the stack and jump if it is false.
case	Indexed jump.
load_local	Load a value at a given offset from the top of the stack.
indirect	Indirect load using the address on the top of the stack and a given offset.
load_constant	Load an immediate value.
const_addr	Load a value from the constant vector.
call_closure	Call a procedure using the closure on the stack.
call_sl	Call a procedure using a static link.
return	Return from a procedure, popping a number of arguments.
get_store	Allocate a new object of a given size.
move_to_vec	Store a value in a new object.
lock	Lock a new object when all the values have been put in.
set_handler	Set up an exception handler.
del_handler	Delete the exception handler.
raise_ex	Raise an exception.
load_ex	Load the last exception name.
reset	Remove values from the stack.
reset_r	Remove values from the stack, sliding the top value down.
set_stack_val	Change a value on the stack (only used in tail-recursion).
pad	A no-operation.

Table 1: Primitive Opcodes

Short integer arithmetic
mul_int, plus_int, minus_int, div_int, mod_int, neg_int, int_eq,
int_neq, int_geq, int_leq, int_gtr, int_lss, printi, repri, convint

Boolean operations
or_bool, and_bool, not_bool, printb, reprb

File operations
openstream, readstream, writestream, closestream, eofstream, lookahead,
stdin, stdout, popenstream, pclosestream

Process operations
fork_process, choice_process, kill_self, int_process, send_on_channel, receive_on_channel

Arbitrary precision arithmetic
aplus, aminus, amul, adiv, amod, aneg, testa, printl, lconvint, reprl

Real number operations
Add_real, Sub_real, Mul_real, Div_real, Comp_real, Neg_real, Print_real, Repr_real,
conv_real, real_to_int, int_to_real, sqrt_real, sin_real, cos_real, arctan_real, exp_real, ln_real

Character operations
succ_char, pred_char, convch

String operations
string_length, prints, strconcat, string_sub, substring, teststreq,
teststrneq, teststrgtr, teststrlss, teststrgeq, teststrleq

Word operations
mul_word, plus_word, minus_word, div_word, mod_word, neg_word, or_word,
and_word, xor_word, shift_right_word, shift_left_word, word_neq,
word_geq, word_leq, word_gtr, word_lss, word_eq, printw, reprw, wconvint

Indexed loads and stores
load_byte, load_word, assign_byte, assign_word

Other operations
exit, install_root, chdir, sysexec, callcode, filedate, gettime, filemode,
ptime, repr_time, io_operation, commit, createf, getstore, lockseg,profiler

The null string
nullorzero

Table 2: Primitive Functions

do this the secondary processor must be able to find the top of the stack and adjust the stack pointer. The interface between the two processors involves the primary processor saving the registers in the base of the stack segment and passing the contents of the stack base register, *sb,* with the function required, to the secondary processor. The secondary processor can use this to find the arguments on the stack, adjust the registers, and return control to the primary processor. The result it returns is the stack base register, allowing the secondary processor to change the process being run by the primary processor.

As well as a request to the secondary processor to perform a specific function the primary processor may also transfer control to it in a number of other cases. If the stack segment for the current process is about to overflow, if the heap has run out, if a virtual memory trap occurs or if the secondary processor interrupts the primary processor, the secondary processor can call to the primary processor. These special cases can be represented by special function values.

The great advantage of separating the functionality between the primary and secondary processors is flexibility. It would seem to be convenient to have a separate physical processor to deal with input/output and it is sensible to define the interface between the processors at a reasonably high level. This processor could be a conventional computer possibly running a standard operating system and regarding the Poly primary processor as a peripheral. Once a device driver has been written to allow the secondary processor to read and write to the memory of the primary processor, to interrupt it and accept requests, it is then relatively easy to add functions to Poly or ML running in the primary processor. It might, for example, be convenient for the first implementation of the primary processor to be a very simple machine supporting only the primitive instruction set and passing all the primitive functions to the secondary processor. Further implementations might support some of the functions, such as adding short integer values, within the primary processor. This would be completely transparent as far rest of the system is concerned. Equally, new primitive functions can be provided, to drive a window system for example, simply by writing suitable code in the secondary processor and using an escape mechanism in Poly or ML to associate a particular function number with a function of a particular type. Since the primary processor will not recognise this function number it will be passed to the secondary processor to be executed there.

4 Specialised Hardware Operations

Implementing a language on special hardware can allow for operations which would be too expensive on a conventional machine. There are two specific areas which would be worth pursuing.

The present Poly and ML system makes use of a persistent storage system to allow for long-term storage of data. Objects are read in from disc as required, and then modified objects can be written out. Because the type systems of Poly and ML restrict the kinds of objects that can be modified to references, arrays and their analogues, it is possible to make this quite efficient. Hardware support for this could improve performance still more.

The other area in which hardware support would be useful would be to provide for asynchronous garbage collection. There are various methods for doing this but they all require too large an overhead on conventional machines. This is basically because they require special action when an assignment is made while the garbage collector is active. With special hardware, however, it would be possible to arrange for the hardware to execute different versions of the indexed store functions depending on the state of some register.