Experiences in Programming the J-Machine

Daniel Maskit and Stephen Taylor
Scalable Concurrent Programming Laboratory
California Institute of Technology

April 30, 1993

1 Introduction

This document summarizes experiences gained in programming the J-Machine. It is intended to provide feedback on the strengths and weaknesses of the architecture. The intent of this document is to provide useful information to system architects to assist in the design of the next generation of fine-grained multicomputers.

2 Feedback

From a high-level perspective the J-Machine architecture functions quite well, and fully supports the major features described in [1]. These features and their chief benefits in our system are:

2.1 Things We Liked:

- *Message-Driven Processes* significantly decrease the amount of software necessary for process management.
- The *associative cache* makes it very simple to provide a mechanism for code and data caching.
- The *hardware message-sending* allows for a clean expression of communication concepts.
- The *hardware synchronization variables* provide a good foundation for implementing single-assignment variables and stream communications. Although for our purposes we only require 1 bit of information for defined vs. undefined.

2.2 Things We Used:

- The flexibility of the *fault mechanism*, coupled with the *user-defined data types*, allowed us to work around some idiosyncrasies of the machine.
- The *multiple priorities* allows for some level of multi-tasking without having to support process suspension. We currently use one priority for executing user code, and one priority for system tasks such as code transport and I/O.

2.3 Things we Didn't Like

In the process of implementing our experimental low-level programming system, however, we encountered a number of aspects of the architecture that posed significant problems. These include:

2.3.1 Message Buffers

- The distinction between message buffer space and regular memory necessitates either copying on receive or copying on suspend. This could also be addressed by allowing a process to issue a suspend call without freeing its space in the message buffer, although I suspect that that is a harder problem to solve.

  We would like to do:

  ```
  save a pointer to process state in memory.
  suspend
      .
      . process waits, then is awakened and finishes running
      .
  suspend
  ```

  Currently, we must:

  ```
  malloc space for message contents
  copy contents to heap
  suspend
      .
      . process waits, then is awakened and finishes running
      .
  suspend
  ```

- The addressing of messages that wrap around within the message buffer is only properly handled if a specific base register is used for such references. This situation necessitates either copying on receive or special-case handling when this condition occurs. If there were no special distinction between the message queue and other memory, the message wraparound could be avoided.

2.3.2 Integer Division

- The lack of an integer divide is problematic in that conversion from integer node numbers to physical node addresses requires division and modulo operations, and software division is expensive. We provide the programmer with the abstraction of nodes being sequentially numbered from 0 to $n-1$, and incur significant overhead in translating from virtual node numbers to physical node numbers.

2.3.3 Addressing

- The prohibition of negative offsets from address registers results in inefficient code generation under the standard C paradigm that there is a pointer with arguments above it, and local variables below it.

  Where we would like to use:

  ```
  move r2, [-1, a2];
  ```

  We have to use:

  ```
  move -1, r0;
  move r2, [r0, a2];
  ```

- The location of the offset bits in address words (<< 10) results in frequently incurring overhead when performing address calculations.

  We would like to do:

  ```
  add a1, 4, a1
  ```

  We have to do:

  ```
  move a1, r0
  ash r0, -10, r0
  add r0, 4, r0
  ash r0, 10, r0
  move r0, a1
  ```

2.3.4 Byte Addressing

- The lack of byte addressing makes string operations awkward and/or inefficient. For example, the string "Hello, world\n", becomes:

  ```
  DC INT:$48
  DC INT:$65
  DC INT:$6c
  DC INT:$6c
  DC INT:$6f
  DC INT:$2c
  DC INT:$20
  DC INT:$77
  DC INT:$6f
  DC INT:$72
  DC INT:$6c
  DC INT:$64
  DC INT:$a
  DC INT:$0
  ```

  This is 14 words of storage for this trivial phrase.

  In addition, the lack of byte addressing significantly complicates the porting of code that is written with the assumption that there are 8-bit bytes available.

  We believe that these problems could be overcome if there were some simple way of accessing the individual bytes within a word once the word is placed in a register.

2.3.5 Immediate Values

- The restriction on the assembler for range of immediates is awkward in that performing an operation such as a shift by 20 bits requires 2 instructions (this is necessary in manipulations such as accessing the high instruction in a word, or the high half of a word. For example, our function IDs use the low 20 bits for the function address at its home node, and 12 bits above this for the address of the home node. Accessing this address requires shifting down 20 and then masking out 12 bits.

  We would like to do:

  ```
  ash r1, -20, r2
  ```

  We have to do:

  ```
  ash r1, -10, r2
  ash r2, -10, r2
  ```

2.3.6 Floating Point

- Enough has been said about the problems in attaining acceptable performance for scientific computations without hardware floating point. We would like to have hardware double-precision which is easily extensible to greater precision. And we would like the floating-point to use standard IEEE formats that we can use existing libraries for most of our required functionality.

2.3.7 Registers

- The relatively small number of registers makes efficient code generation difficult.
- The distinction between address and data registers, and the different capabilities of the two registers also cause code generation problems. This can be seen in the example for address arithmetic, where in order to do:

  ```
  add a1, 4, a1
  ```

  we must do:

  ```
  move a1, r0
  wtag r0, INT, r0
  ash r0, -10, r0
  add r0, 4, r0
  ash r0, 10, r0
  wtag r0, ADDR, r0
  move r0, a1
  ```

2.3.8 Unsigned Comparisons

- Unsigned comparisons would be useful. Where we would like to do:

  ```
  ltu r1, r2, r1
  ```

  We actually have:

  ```
      int ult(Register x, Register y)
  {
      Register sx = x >> 31;
      Register sy = y >> 31;

      return ( sx - sy )?sy:(x < y);
  }
  ```

2.3.9 Processor Status Flags

- It would be nice to be able to do direct setting of the processor status flags without going through an intermediate register:

  We would like to do:

  ```
  move 1, U
  ```

  but must instead do:

  ```
  move 1, r0
  move r0, U
  ```

2.3.10 Internode Addressing

- It would be desirable if there were some way to send messages to, and access the memory at, another node without having to know that nodes physical address.

3 Conclusion

Although the bulk of this report is dedicated to complaints about the J-Machine, we wish to emphasize that this does not reflect displeasure with the machine. We have succeeded in bringing up some fairly large applications on the machine, and are confident that it will continue to be a useful platform for experimentation. The only problems that we have been unable to solve to our satisfaction are avoiding copying upon message reception, avoiding copying on process suspension, and floating-point performance. We are confident that these problems will disappear in the next generation of machines.

4 Acknowledgements

This work owes a great debt to Mike Noakes of the Concurrent VLSI Architecture project at MIT, who has offered much patience and calm assistance in furthering our understanding of the intricacies of the J-Machine Architecture. We would also like to express thanks to Yair Zadik at Caltech who helped in clarifying some of our suggestions for improvement.

References

[1] Dally, W. J., et al., "The J-Machine: A Fine-grain Concurrent Computer," *Information Processing 89*, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.