Newcastle University e-prints

Date deposited: 19th August 2011

Version of file: Author final

Peer Review Status: Unknown

Citation for item:

Randell B. Hardware/Software Trade-offs: A general design principle? *Computer Architecture News* 1985, **13**(2), 19-21.

Further information on publisher website:

Publisher's copyright statement:

Use Policy:

Hardware/Software Tradeoffs: A General Design Principle?

by

Brian Randell

Computing Laboratory, The University of Newcastle upon Tyne

January 25, 1985

> "Hardware and software are logically equivalent. Any operation performed by software can also be built directly into the hardware and any instruction executed by the hardware can also be simulated in software. The decision to put certain features in hardware and others in software is based on such factors as cost, speed, reliability and frequency of change. There are no hard and fast rules to the effect that X must go into the hardware and Y must be programmed explicitly. Designers with different goals may, and often do, make different decisions ... the boundary between hardware and software is arbitrary and constantly changing. Today's software is tomorrow's hardware, and vice versa." [1]

Although I find much to agree with in the above quotation, I know of one general design "principle" that can provide guidance as to the positioning of the hardware/software boundary. It concerns testing for rare events, and can be stated as follows:

> "Simple frequent highly-skew conditional branches should be implemented in hardware".

In fact over the years a succession of examples of such situations have been identified, and incorporated into many instruction sets, such as I/O interrupts, arithmetic overflow, array bound checks, and page faulting. In all these cases, one could of course achieve the same effect by explicit tests and conditional branches. However the branches would be taken so infrequently that programmers have a great temptation to try and economise, both on space and time, by not performing the tests as frequently as is really needed. In the case of array bound and perhaps even arithmetic overflow checks, the temptation to remove the checks once the program is believed to be ready to go into production, i.e. when the results are first to be depended upon, is often too hard to resist.

I remember being aware of the above formulation of a rather obvious point at least a decade ago, and recall using it to justify the notion of a 'recovery cache'.[2] This cache mechanism incorporated hardware to check before each 'write to memory', whether this was the first 'write' to that location since a recovery region had been entered. (Just before such 'first writes', a copy of the contents of the location would be stored away, in case it became necessary to perform backwards error recovery - in effect the recovery cache mechanism can be regarded as performing incremental check pointing or, equivalently, as recording an optimized audit trail.) However I am not aware of seeing the "principle" in print anywhere, and on mentioning this to Gordon Bell, was encouraged to write it down – hence this note.

There is however one small snag, namely that the "principle" is in danger of losing whatever validity it currently has. The rationale behind it is that, in general, the software/hardware boundary is also the sequential/parallel boundary. Simple skew conditional branches, which with their associated tests might well involve several sequential instructions, can easily be performed in parallel with ordinary instruction execution at only a modest hardware cost, whilst

programs remain sequential, as far as the programmer is concerned. However novel highly parallel processor architectures are now all the vogue and, according to some people at least, are due to replace von Neumann-style architectures. The trouble is that when this happens I will have lost my principal design "principle"! To end on a question: what other general principles, preferably of long term validity, exist to guide hardware/software trade-offs?

References

1. A. S. Tanenbaum, *Structured Computer Organization (2nd Ed.)*, Prentice Hall, Englewood Cliffs, NJ, 1984.

2. J. J. Horning et aI, "A Program Structure for Error Detection and Recovery," in *Lecture Notes in Computer Science 16*, ed. E. Gelenbe and C. Kaiser, pp. 171-187, Springer-Verlag, Berlin, 1974.
21