Fallacies and Pitfalls In Building Supercomputers

Kenneth E. Batcher
Dept. Of Computer Science
Kent State University
Kent OH 44224
www.cs.kent.edu/~batcher/

STARAN
• An associative processor with content-addressable data.
• I/O reads and writes data in the Word Mode.
• For content addressing data can also be read and written in the Bit-Slice Mode.
• A Multi-Dimensional Access (MDA) memory for the RAM.

MDA Memory
• Used off-the-shelf RAM chips with data scrambled a certain way in memory.
• Idea came from an ILLIAC-IV paper.
• Their scrambling pattern was changed to simplify addressing - N adders were replaced by (log N) XOR-gates.
• Needs a network to unscramble/scramble data when reading/writing memory.
• But that network is needed anyway for inter-PE communication.

STARAN Array Unit
ASPRO

- ASPRO is an airborne STARAN with 1792 processing elements.
- Built in the late 1970’s.
- Used VLSI - STARAN in a shoebox.
- Over 170 systems delivered for use in: aircraft early warning radar surveillance; and command & control processing.

The Massively Parallel Processor (MPP)

- Designed in response to an RFP from NASA-Goddard Space Flight Center.
- A special-purpose machine for processing imagery from satellites.
- 16,384 bit-serial PE’s in a 2-D mesh.
- Plus a large corner-turning memory to re-format image data.

The Time Frame

1972 - STARAN was demonstrated at the TRANSPO exhibition.
1983 - The MPP was delivered to NASA-Goddard Space Flight Center.
1990 - Hennessy and Patterson wrote their Computer Architecture: A Quantitative Analysis (QA) book - each chapter has a section called Fallacies and Pitfalls.

Mistakes and Excuses

- We made a number of mistakes in the designs of the STARAN and the MPP.
- Excuse 1: We didn’t have a chance to read the QA book.
- Excuse 2: Many others also made similar mistakes according to the Fallacies and Pitfalls sections of the QA book.
Special-Purpose Designs

- STARAN started as a machine for an air traffic control application.
- The MPP was a machine for an image processing application.
- Didn’t change their designs when it was decided to also market them as general-purpose machines.

STARAN Address Space

- Application programmers said that 256 bits/PE was more than enough for air traffic control.
- We used 8-bit addresses with no way of expanding their length.
- As it turned out, the application needed more bits - programmers had to use two PE’s for each item.

The MPP Address Space

- The RFP specified 256 bits/PE but we knew from our STARAN experience that that wasn’t nearly enough.
- We supplied 1024 bits/PE and used 16-bit addresses so memory could be expanded up to 65,536 bits/PE.

Lack of Software Support

- No software input on the Instruction Set Architectures.
- In fact, hardly any consideration of software anywhere.
- Since each instruction operates on an array of data we felt that users would be satisfied with just an assembly language.
NASA-Goddard Knew Better

• Fortunately, the MPP customer knew better and gave Prof. Anthony Reeves of Cornell a grant to develop a Parallel Pascal language and the front end of a compiler.
• Almost every user of the MPP used his Parallel Pascal.

MPP Memory-Processor Gap

• SRAM technology at the time limited the clock speed to 10 MHz - the PE logic could’ve run at a much higher rate.
• We should’ve considered adding registers and/or caching to the PE’s to close this gap.

Bad Exhibition Power

• Didn’t think that the main power at the TRANSPO exhibition would be so bad.
• The STARAN we were demonstrating was running perfectly the day before the exhibition started.
• But it went down a few minutes after the exhibition opened up.
• It took us 24 hours to bring it back up.

A Bug in the Software

• We used a minicomputer as the front-end in STARAN - we didn’t think their software would have any bugs.
• But their routine to backup the disk onto magnetic tape crashed one night.
• A co-worker, Carl Mickelson, found that the routine used faulty Quicksort code that crashed the system with a Stack Overflow error.
QuickSort Bugs

• At first I thought that the faulty QuickSort code in the Disk Backup routine was only an isolated event.
• But decades later I joined the academic ranks and taught the Intro-to-Algorithms course a number of times.
• I found that most every textbook illustrates QuickSort with the faulty code that can cause a Stack Overflow!
• I call such a code a QuickSort Bug.

The Code of a QuickSort Bug

• Most every textbook illustrates QuickSort with the following pseudo-code or the code in some source language:

```python
QuickSort(A, Lo, Hi)
if (Lo < Hi) then
    pivot <-- Partition(A, Lo, Hi)
    QuickSort(A, Lo, pivot-1)
    QuickSort(A, pivot+1, Hi)
```

The Plague of QuickSort Bugs

• Google® returns over 50,000 hits when given the search terms:
  “QuickSort” & “Stack” & “Overflow”
• A random sample of these hits suggests that thousands of users have reported that there QuickSort program suddenly crashed with a Stack-Overflow error.
• The chance that a given call to a QuickSort Bug causes a Stack-Overflow is very small - it must take millions of calls to generate thousands of Stack-Overflows.

The Bitonic Sort Mistake

• Bitonic combines a Latin prefix with a Greek root - the word should really be ditonic.
• I didn’t know my Latin from my Greek.
• Excuse 1: My Electrical Engineering education never covered Latin and Greek.
• Excuse 2: Others have also used the word bitonic with no comments that it’s wrong.
• One result of this mistake is that Google® returns very few false hits when searching for bitonic.
Conclusions

- I made a number of mistakes but I think I have good excuses for the mistakes.
- If I ever design another parallel processor or supercomputer I would try to make it the ideal machine.
- A machine where the typical user doesn’t know (nor needs to know) that it’s a parallel processor or supercomputer.

The End?

This is the end of my talk unless we have more time and some people want to know more about Quicksort Bugs.

The Quicksort-Bug

Most every textbook illustrates Quicksort with the following pseudo-code (or the code in some source language.)

```plaintext
QuickSort(A, Lo, Hi)
  while Lo < Hi do
    pivot <-- Partition(A, Lo, Hi)
    QuickSort(A, Lo, pivot-1)
    QuickSort(A, pivot+1, Hi)
```

What is a Stack Overflow?

- When a computer runs any program, every procedure call pushes a stack frame on a control stack to return control back to the caller at the end of the called procedure.
- Each stack frame saves the values of the local variables and the registers used by the caller: it contains hundreds of bytes.
The Stack Overflow Problem of a Quicksort-Bug

- A Quicksort-Bug can push $\theta(n)$ stack frames onto the control stack.
- Each frame has hundreds of bytes so the control stack can grow many times larger than the array being sorted!
- If $n$ is large the control stack may overflow its allotted storage.

Eliminating Tail Recursion

- A recursive program is said to be tail-recursive if it ends with a call to itself.
- The call to itself puts another frame on the control stack which merely executes the same program with a new set of inputs.
- Instead of calling itself the program can be changed to repeat with a new set of inputs.
- The change eliminates the extra frame that tail-recursion puts on the control stack.

Fixing the Stack Overflow Problem of Quicksort

- Quicksort calls itself twice - once for the left sub-array and again for the right sub-array. Eliminating tail recursion removes only one of these calls.
- It’s best to remove the call for the longer sub-array - the stack size of the call for the shorter sub-array is limited to $\theta(\log n)$.

Modified Quicksort

Besides eliminating stack overflows this code runs quicker because it makes less recursive calls.

```plaintext
Quicksort(A, Lo, Hi)
while Lo < Hi do
    pivot <-- Partition(A, Lo, Hi)
    if (pivot+pivot < Lo+Hi)
        then Quicksort(A, Lo, pivot-1)
        Lo <-- pivot+1
    else Quicksort(A, pivot+1, Hi)
        Hi <-- pivot-1
```