The Use of Overloaded Software Operators for Error Detection and Correction

Scott A. McDermott, Luis Jordan, Kalle Anderson
18th Annual AIAA/USU Conference on Small Satellites
August, 2004
Current Space Computing Hardware

To handle Single Event Upsets, custom hardware is needed on computing devices.
**What Microprocessors are Good At**

- Speed, and speed per Watt, are increasing dramatically
- On-chip memory is increasing in large steps
- Off-chip memory access time is not keeping pace
- Hardware EDAC leaves the microprocessor idling while it accesses external memory

*Hardware EDAC and processor technology are diverging*

Microprocessor parameters over time; PowerPC 603 to 7447a
If an efficient software method for performing EDAC were available, the trend would reverse

- Increases in on-chip memory and MIPS/W would result in more useful, instead of more wasted, processor performance
- It would not matter where variables were stored
  - on-chip memory
  - off-chip memory
  - ALU data registers themselves

Downside: the processor must spend cycles during every variable operation checking to see if the variable is healthy

How do we do this efficiently?
Overloaded Operators

- **Operator Overloading:** the ability to redefine the meanings of symbols otherwise recognized by the compiler
  - Tells compiler: “When an expression such as \( X = A + B + C \) is encountered, use these special functions for ‘\( = \)’ and ‘\(+\)’ instead of the usual one.”
  - The special functions then perform the error correction and error-bit creation tasks required for EDAC.

- **Start with the language’s own base types**
  - Integer, floating point, character…

- **Create protected versions of each base type**
  - ProtInt, ProtFloat, ProtChar…
  - Includes enough bits for base type plus error-check bits

- **Overload all the base-type operations for the protected types**
  - Addition, subtraction, assignment, reference…
  - Performs error check on all operands that are read, and calculates error-check bits for all operands that are written
Overloaded Operator Flowchart

**Operator \( \gamma \)**

1. Read operand1
2. Read operand2 (if present)
3. Calc \( \gamma(\text{op1,op2}) \) using base-type operation
4. Calc error-check bits on result
5. Return protected result

Result may be “exposed” here, depending on the ECC method; comparable to hardware EDAC, where variables are exposed whenever they are in the ALU registers.

**EDAC Read Function**

- Read operand
- This can take “arbitrarily long”
- 0 errors?
- 1 error?
- Exception
- Return source data
- Correct SEU

**Framework for EDAC-Enhanced Operators**

*All operators for a given base data type will be overloaded to implement this kind of function*
### Types of Software Error Checking

<table>
<thead>
<tr>
<th>Method (how it does variable read)</th>
<th>Instr Ratio</th>
<th>Mem Ratio</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-EDAC (simple load)</td>
<td>1X</td>
<td>1X</td>
<td>Assumes EDAC handled by hardware; cannot do EDAC on internal memory.</td>
</tr>
<tr>
<td>Triple save (load 3 copies of variable and vote)</td>
<td>7X</td>
<td>3X</td>
<td>Variables are never exposed. Can be fooled by some dual-errors, but otherwise is a very strong approach.</td>
</tr>
<tr>
<td>Triple XOR (extra word that is a 1’s-comp. combination of source word)</td>
<td>8X</td>
<td>2X</td>
<td>Achieves Single Error Correction, Double Error Detection. Good overall cycle/memory balance.</td>
</tr>
<tr>
<td>Hamming (system used in hardware EDAC)</td>
<td>&gt;&gt;50X</td>
<td>1.25X</td>
<td>For comparison only; not a sensible Error Correction Code for software.</td>
</tr>
</tbody>
</table>

*These must be balanced against the idle time and lower protection of hardware EDAC*
Protected Type Declaration

template <class Type> class Prot {
public:
  Prot<Type> (Type val) {src=val; errmake();}
...
  virtual Prot<Type> operator+();
  virtual Prot<Type> operator+(Prot<Type> val);
  virtual Prot<Type> operator-();
  virtual Prot<Type> operator-(Prot<Type> val);
...
  virtual ProtBool operator>(Prot<Type> val);
  virtual ProtBool operator<(Prot<Type> val);
...
  virtual ProtPtr<Prot<Type>> operator&();

protected:
  virtual Type read();
  virtual void errmake();
  virtual void UpsetHandler();

  Type src;
  Type err;
};
Protected Type Operator Definitions

template <class Type> Prot<Type>
Prot<Type>::operator+() {
    this->read();
    return(*this);
}

template <class Type> Prot<Type>
Prot<Type>::operator+(Prot<Type> val) {
    Prot<Type> ret;
    ret.src=read()+val.read();
    ret.errmake();
    return(ret);
}
Protected Type read() function

/* Triple XOR ECC */

template <class Type> Type
Prot<Type>::read() {
    if ((src ^ (src<<1) ^ (src<<2) ^
        ((src>>(8*sizeof(src))-1))&1) ^
        ((src>>(8*sizeof(src))-2))&3)) ^
        err) UpsetHandler();

    return(src);
}

lwz   r1,0(r7) ; load src value
rotlwi r2,r1,1 ; rotate by 1 bit
xor   r3,r1,r2 ; and XOR
rotlwi r2,r1,2 ; rotate by 2 bits
xor   r3,r3,r2 ; and XOR
lwz   r2,1(r7) ; load err value
xor.  r3,r3,r2 ; and XOR
bcl   4,2,UpsetHandler ; should be 0
Use of Protected Type

typedef Prot<Float> pfloat;

typedef struct {
    pfloat a1,a2;
    pfloat b1,b2;
} matrix2b2;

pfloat CalcDeterminant(matrix2b2 min) {
    pfloat det;
    det = (a1*b2) – (a2*b1);
    return(det);
}

For very frequently used structures, protection can be at the structure level.
Conclusion

- By using software EDAC instead of hardware,
  - Microprocessor advances are more useful instead of more wasted
  - On-processor registers (ALU, L1 cache) are protected
  - Single-chip microcontrollers and COTS boards can have EDAC

- By using overloaded operators to achieve software EDAC,
  - Operational code looks the same as it always has
  - Details of EDAC implementation are hidden within protected variable
  - EDAC implementation can change depending on the variable type

- The catches are,
  - Still slower than hardware EDAC
  - Increased software and data memory size