Verifying Architecture

Jaein Jeong
Johnathon Jamison
{jaein,jjamison}@cs.berkeley.edu

1. Introduction

As processors get smaller and faster, they become more vulnerable to transient errors. Minor imperfections in a chip, cosmic rays, or similar phenomenon can cause transistors to occasionally produce wrong results. This does not mean that we cannot use advanced processors because we are afraid of those errors. We can detect those transient errors more stable processors and execute instructions again if an error occurs. If the probability of errors are very low, the overhead of additional verifying processors won't be high. DIVA[1] showed that the idea is feasible. DIVA has a second, slower processor which verifies the output of each individual instruction. We used the idea that we can verify groups of instructions. It is implemented as a dual-processor system with SimpleScalar[2]. A proper system could produce executable programs with no intervention of an operator. Currently, multiple compiler passes and human intervention is required. Our implementation works on a small scale. We believe that the verifying architecture can be applied to a real system with modifications.

SimpleScalar is a processor simulation tool set developed by the University of Wisconsin. It simulates various features of modern processors, like caches, a TLB, and branch prediction. The development environment includes a compiler (gcc) and library so researches can develop or port programs. SimpleScalar is easy to extend. Since it was first released in 1996, SimpleScalar has had many features added. The most recent extension simulates multiprocessors, which is very useful for implementing a main processor and its verifying processor.

Proof-carrying code[3] is a system by which a proof of safety accompanies executable code. Code is annotated with invariants that, if they hold, prove the code to be safe. Workstations can verify that untrusted code meets safety restrictions by analyzing the proof with the code. Our idea is similar to proof-carrying code in that executable code is annotated with invariants which must hold at that point in the code. The processor executes instructions, and reexecutes instructions when an error causes an invariant to be false.

2. Our work

2.1. Assumptions

As processors get faster and smaller, it is more prone to errors. A processor can have transient errors as well as permanent errors. We are only going to address the case where the processor has transient errors. For example, an alpha particle can cause a malfunction in a processor circuit. However, this does not mean that we cannot use the processor. Generally, we assume the processor operates correctly most of the time, and only fails on occasion. Therefore, the
processor can accomplish its tasks correctly if it is verified by a more stable processor. As the probability of an transient error is low, the verifying processor doesn't need to verify all each instruction. The verifying processor will only execute a small set of instructions: the invariant. This will catch errors with high probability. If the invariant doesn't hold, the main processor executes the instructions again. The overhead of reexecuting instructions is not problematic, considering that errors occur rarely.

2.2. System Structure

We implemented our idea as two communicating processors. In SimpleScalar, processes communicate by passing messages (shared memory is not yet implemented). The main processor executes instructions and then sends the verifying processor all its registers. If the verifying processor confirms that the execution was correct, the main processor continues to execute instructions. If not, the main processor loads the old register values back and reexecutes its instructions. This mechanism is shown in Fig. 1.

Two communicating processors are specified in a script such as that in Figure 2.

```
cluster jeckel {

  processor main outorder 1 {
    -mem:lat 40 1
    -command my_command
    -btrace main.trace

    input { min[0] }
    output { mout[0] }
  }

  processor verifier outorder 1 {
    -mem:lat 40 1
    -command my_verifier
    -btrace verifier.trace

    input { vin[0] }
    output { vout[0] }
  }

  jeckel.main.mout[0]=>jeckel.verifier.vin[0];
  jeckel.verifier.vout[0]=>jeckel.main.min[0];
}
```
The processor configuration file specifies the processor and message queue names and maps the queues between processors. In the example above, the processor main sends messages to message queue mout and receives messages from min. The process running on main was loaded from the object file my_command. The processor verifier sends messages to vout and receives from vin with object file my_verifier. Any messages written to mout will arrive in vin and any written to vout will arrive in min. The two processors are enclosed in a cluster jeckel. In the future, SimpleScalar will be modified so processors within a single cluster will share memory. As that is not currently implemented, clusters are useless.

2.3. Writing a program for SimpleScalar

Since gcc can not handle everything we want it to do, we must compile to assembly language and intervene at that point. gcc generates assembly code when it is given the -S option. After we change the assembly code for our purposes, we compile the modified assembly code to object code.

In SimpleScalar, a process sends and receives messages with the system calls qread and qwrite. Since they are not fully integrated in SimpleScalar at this time, we had to insert the syscall instruction and pass the arguments by explicitly filling registers. Figures 3 and 4 shows the code to write and read messages from a queue.

Two communicating processors are specified in a script such as that in Figure 2.

```assembly
addiu $2, $0, 258       # Set register 2 to 258 (system call number)
la      $4, MQO       # Set register 4 to the queue name
subu    $5, $16, 4    # Set register 5 to the queue message
move    $6, $0        # Set register 6 to queue index
syscall                 # make system call

$L2:
addiu $2, $0, 259       # Set reg 2 to 259 (system call number)
la      $4, MQI       # Set reg 4 to the queue name
addu    $5, $sp, 16    # Set reg 5 to the queue message
move    $6, $0        # Set reg 6 to queue index
syscall                 # make system call
bne     $7, $0, $L2    # reg 7 is 0 if a message exists, otherwise
# it is 1. We loop until there is a message.
```

The queue name is a string where the first byte is a length byte; the length excludes the terminating null byte. The queue message has its length in the first four bytes followed by the contents. As SimpleScalar is a little-endian machine, the least significant byte of the length is the first byte.

Figure 5 contains an example.
2.4. Programming interface for C

As assembly language programming is error prone and unproductive, we wrote an interface for C with macros and inline assembly. These functions correctly map arguments to registers. We have several variations. The \texttt{b} suffix indicates a blocking version. \texttt{ne} indicates a non-error-returning version.

```c
#define qread(messagelength,queuename,message,queuenumber,queueerror)  
  ({asm volatile("addiu \t$2,$0,259":="2");  
    messagelength = syscall(queuename,message,queuenumber,0);  
    asm volatile("move \t%0,$7":="g" (queueerror));})
#define qreadne(messagelength,queuename,message,queuenumber)            
  ({asm volatile("addiu \t$2,$0,259":="2");  
    messagelength = syscall(queuename,message,queuenumber,0);})
#define qreadb(messagelength,queuename,message,queuenumber,queueerror)  
  ({do { 
      qread(messagelength,queuename,message,queuenumber,queueerror); 
    } while(queueerror);})
#define qreadbne(messagelength,queuename,message,queuenumber)            
  ({int queueerror;  
    do { 
      qread(messagelength,queuename,message,queuenumber,queueerror); 
    } while(queueerror);})
#define qwrite(queuename,message,queuenumber,queueerror)               
  ({asm volatile("addiu \t$2,$0,258":="2");  
    syscall(queuename,message,queuenumber,0);  
    asm volatile("move \t%0,$7":="g" (queueerror));})
#define qwritene(queuename,message,queuenumber)                         
  ({asm volatile("addiu \t$2,$0,258":="2");  
    syscall(queuename,message,queuenumber,0);})
```

Fig. 6: C interface for queue read and write

\texttt{syscall} is translated to \texttt{jal syscall} in the assembly file. Since \texttt{syscall} is the proper assembly command, a correction must be made. The following Perl script makes the modification. It also comments out any call to \texttt{test}. This is so we can insert a bogus call to \texttt{test} so \texttt{gcc} will not optimize away variables.

```perl
#!/usr/bin/perl
```
while(<>) {
    s/jal\tsyscall/syscall/;
    s/\tjal\tttest/#\tjal\tttest/;
    print;
}

Fig. 7: Perl script to change syscall instruction

After running the Perl script, we can compile the assembly code without further modification.

2.5. Multiprocessor SimpleScalar Program in C

Writing a multiprocessor program is not very difficult if we use the C interface as shown in the following examples.

```c
#include "queue_calls.h"

long regs[32];
char msg[]="\006\000\000\000\000\000\000\000cool\n";
long nullmsg[]={0};
char MQI[]="\003min";
char MQO[]="\004mout";

int main(void) {
    int i, error, length;
    for(i = 0; i < 32; i++) {
        regs[i]=0;
    }
    qwrite(MQO,msg,0,error);
    do {
        qread(length,MQI,regs,0,error);
        if(length == 0)
            break;
        printf(regs+1);
        qwrite(VQO,sucmsg,0,error);
        printf("1\n");
    } while(error);
    qwrite(MQO,nullmsg,0,error);
    printf("done\n");
}
```

Fig. 8: A program running on the two processors

In the two communicating programs, the queue reads and writes should match each other. Also, the queue read should wait until there is a message in the queue.

2.6. Passing invariants

Up to now, we have just shown programs that send and receive data. But how can the main processor send an invariant to the verifying processor? We propose two methods.

In the first method, the main program sends the invariant instructions as a message. This is possible because we can enclose the invariant instructions with .rdata and .text directives and insert the length of the message after .rdata. We load $I1$ as our message, and so the
instructions are sent. The verifying processor then can load its registers with those sent by the
main processor, and do a jal to the message that was sent. We can ensure that the invariant code
always leaves its result in register 4. Then we can check that register and reply to the main
processor with the result.

```
#include "queue_calls.h"
long regs[2][32];
char VQI[]="\004vin";
char VQO[]="\005vout";
long sucmsg[]={1, 1};
int main(void)
{
    long i, length, error;
    for(;;) {
        qreadne(length,VQI,regs,0);
        if(length == 0)
            break;
        if(confirm(regs[currregs]+1)) {
            qwritene(VQO,sucmsg,0);
            currregs = 1 - currregs;
        } else
            regs[1 - currregs][1] = 0;
        qwritene(VQO,regs[1 - currregs],0);
        stats[1][0]++;
    }
}
```

Fig. 9: Invariant code and the generic verifier

In the second method, we generate a verifying program specific to the main program. When we
run the main program we just send the the contents of registers and the number designating
which invariant we are at. The verifying processor takes the invariant number, calculates the
result of the invariant, and replies. We use the assembly code for the main program to deduce the
meaning of the various registers. We could then write straight C code for the verifier.

```
#include "invariant.h"
int main(void)
{
    int i,sum,j;
    invstart();
    for(j=0;j<200;j++) {
        sum = 0;
        for(i=1;i <= 240; i++) {
            sum += i;
        }
    }
invariant(sum == 28920);
    printf("done\n");
invdone();
}
```

```
#include "queue_calls.h"
long regs[2][33];
char VQI[]="\004vin";
char VQO[]="\005vout";
long sucmsg[]={4, 1};
int main(void)
{
    long i, length, currregs=0,
stats[3][2]={{{0,0},{0,0}},0,0};
    for(;;) {
        qreadne(length,VQI,regs[currregs],0);
        if(length == 0)
            break;
        switch(regs[currregs][1]) {
            case 0:
                qwritene(VQO,sucmsg,0);
                break;
            case 1:
                break;
            case 2:
                break;
            case 3:
                break;
            case 4:
                break;
            case 5:
                break;
            case 6:
                break;
            case 7:
                break;
            case 8:
                break;
            case 9:
                break;
            case 10:
                break;
            case 11:
                break;
            case 12:
                break;
            case 13:
                break;
            case 14:
                break;
            case 15:
                break;
            case 16:
                break;
            case 17:
                break;
            case 18:
                break;
            case 19:
                break;
            case 20:
                break;
            case 21:
                break;
            case 22:
                break;
            case 23:
                break;
            case 24:
                break;
            case 25:
                break;
            case 26:
                break;
            case 27:
                break;
            case 28:
                break;
            case 29:
                break;
            case 30:
                break;
            case 31:
                break;
            case 32:
                break;
            case 33:
                break;
        }
    }
}
```
A bit of a problem exists for the first method. The problem is that the verifying program receives invariant instructions as data. To then attempt to execute those instructions would bring up the same issues as self-modifying code. To use this method, we would be required to flush caches, and in general be careful with what we were doing. We decided that the pitfalls of this method would make it more difficult to implement. Therefore, we chose the second method.

2.7. Using invariants

We maintain two sets of registers in the verifier, so that we can return the old register bank to the main processor in the event of an error. Also, given the implementation, not all the registers must be sent to the verifier, but only those that are required for the invariant and possible rollback. However, we do in fact send every register, as we did not want to do a bunch of analysis of the assembly code.

At this point, setting up the invariant in the verifier requires careful inspection of the assembly code in the main program. Also, heavy tweaking of the main code is needed to message filled and sent. We also need to have gcc think it is going to use the variables in the invariant, so we call a non-existant function test, and then comment out that portion of the assembly code. The implementation of invariants is in figure xxx. Where it says \#insert <something>, we insert assembly code to load the registers from or store them to the regs array. Also, we must change 0x7fffffff to a different number for each invariant. We hope to be able to automate much of
the code generation for invariant sending and register copying.

```c
#define invstart()                                    
({long length;                                       
  regs[1]=0;                                         
  asm volatile("#insert start reg load
    \""tla$t2,$I0\n""
    \""tsw$t2,%0":"=g"(regs[32]):"2");     
  qwritene(MQO,regs,0);                              
  qreadbne(length,MQI,reply,0);                       
  asm volatile("$I0:"::);})

#define invariant( expression )                     
({long length;                                       
  regs[1]=0x7fffffff;                                 
  asm volatile("#insert inv reg load
    \""tla$t2,$I7fffffff\n""
    \""tsw$t2,%0":"=g" (regs[32]):"2");    
  qwritene(MQO,regs,0);                              
  test(expression);                                  
  qreadbne(length,MQI,reply,0);                      
  if(!reply[1]) {                                    
    asm volatile("#insert inv reg restore
    \""tj$t2::"g"(reply[32]):"2");        
  }                                                  
  asm volatile("$I7fffffff:"::);})

#define invdone()                                     
({qwritene(MQO,nullmsg,0);})
```

Fig. 11: Implementation of the invariant macros

If you notice, this method sends a message and waits for a reply. We attempted to implement a more efficient method by doing work between the write and read. However, it is slightly buggy. See Future Work(Section 4.1) for more detail.

### 3. Results

#### 3.1. Test Runs

In order to evaluate our ideas, we used the programs in figure 10 and observed the running time by varying number of iterations and error rates(figure 12). To simulate errors, we had a random number generator return 0 or 1, and the verifier sent a failure message if a 1 was returned.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>base</th>
<th>0%</th>
<th>2%</th>
<th>4%</th>
<th>6%</th>
<th>8%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>9,688</td>
<td>22,627</td>
<td>23,607</td>
<td>24,184</td>
<td>24,731</td>
<td>25,313</td>
<td>25,865</td>
</tr>
<tr>
<td>60</td>
<td>20,088</td>
<td>33,022</td>
<td>34,376</td>
<td>35,151</td>
<td>35,936</td>
<td>36,686</td>
<td>37,471</td>
</tr>
<tr>
<td>240</td>
<td>62,088</td>
<td>75,022</td>
<td>77,846</td>
<td>79,461</td>
<td>81,086</td>
<td>82,676</td>
<td>84,301</td>
</tr>
<tr>
<td>960</td>
<td>230,088</td>
<td>242,627</td>
<td>252,251</td>
<td>257,226</td>
<td>262,201</td>
<td>267,161</td>
<td>272,121</td>
</tr>
<tr>
<td>3840</td>
<td>902,088</td>
<td>914,627</td>
<td>947,771</td>
<td>966,186</td>
<td>984,601</td>
<td>1,003,001</td>
<td>1,021,401</td>
</tr>
</tbody>
</table>

Figure 12: Test Run results
We ran the program with five different numbers of iterations: 15, 60, 240, 960, and 3840. For each iteration we varied the error rate: 0%, 2%, 4%, 6%, 8%, and 10%. The system of error rate 0% passes messages, but it doesn’t have any transient errors. The base system is an application that does the same calculations, but doesn’t pass messages or have any errors.

**3.2. Analysis of message passing overhead**

From the table we could get two facts that support our ideas.

First, the overhead of message passing is amortized as the size of an application program grows. As is shown in figures 13 and 14, the message passing overhead is much larger than the base case when the number of iterations is 15. But the difference of running time between the base system and the verifying system narrows when the iterations grows to 3840. That means our idea can be used in real applications without much overhead if the amount of computation between invariants is large.

Second, the overhead of message passing is a relatively small amount. The Fig 14 shows that the overhead of message passing increases proportional to error rate. If we assume that the error rate is small, the overhead associated with reexecution is small.

**4. Conclusion**

**4.1. Future work**

Although our program showed how two communicating processors can verify execution, more work is needed to apply this to a real application.

First, additional logic is needed. We ignore floating point registers, since they were never used. As real applications have both integer and floating point instructions, a processor needs to recover floating point registers. We can extend our idea to floating point instructions with little
difficulty. Also, memory recover logic is necessary. Small programs may be able to do all their work in registers, but any reasonable program goes beyond that. This can be done by keeping a memory write buffer for written memory values. On each successful evaluation of the invariants, the memory values in the write buffer are retired to the memory. If an invariants does not hold, then the memory values are discarded. There can be a coherence problem when there is more than one main processor, but techniques applied to cache coherence could probably be applied here. Maintaining the write buffer seems reasonable in a single main processor architecture with message passing, because the memory values are only referred to by the single main processor.

Second, we need to write a program that generates the verifying program automatically. We generated the verifying program in an ad hoc manner, which is unproductive and error prone, as well as inelegant. But we believe that this can be done without too much difficulty compared to memory value recovery, which will require in-depth modification of SimpleScalar.

Last, in order to get the best performance in the main processor, the main processor should not check for the reply from the verifying processor immediately after sending the invariant message. Rather, it should continue execution until it has reached the time for sending another invariant message. By this time, the reply should have arrived back, and the main processor need not wait. Then the read can be done, and the rollback if necessary. If no rollback is necessary, then the new invariant is sent, and execution continues. An implementation is in figure yyy. However, we did not have the rollback working quite right.

```c
#define invstart()                                       \   
  ({long length;                                      \  
    asm volatile("$I0:"::);                            \  
    regs[1]=0;                                         \  
    asm volatile("#insert start reg load
"          \  
      "\tla\t$s2,$I0\n"                                \  
      "\tsw\t$s2,0"="=g"{regs[32]}::"2");           \  
    qwritene(MQO,regs,0);})                            \  

#define invariant(expression)                           \   
  ({long length;                                      \  
    qreadbne(length,MQI,reply,0);                     \  
    if(!reply[1]) {                                  \  
      asm volatile("#insert inv reg restore
"          \  
        "\tlw\t$t2,0\n"                                \  
        "\tj\t$t2":"=g"{reply[32]}::"2");          \  
    }                                                  \  
    asm volatile("$17fffffff:"::);                   \  
    regs[1]=0x17fffffff;                               \  
    asm volatile("#insert inv reg load
"          \  
      "\tl$a\t$s2,0x17fffffff\n"                    \  
      "\tsw\t$s2,0"="=g"{regs[32]}::"2");          \  
    qwritene(MQO,regs,0); test(expression);\})       \  

#define invdone()                                        \   
  ({long length;                                      \  
    qreadbne(length,MQI,reply,0);                     \  
    if(!reply[1]) {                                  \  
      asm volatile("#insert inv reg restore
"          \  
      "\tla\t$s2,0x17fffffff\n"                       \  
      "\tsw\t$s2,0"="=g"{regs[32]}::"2");          \  
    }                                                  \  
    asm volatile("$17fffffff:"::);                   \  
    regs[1]=0x7fffffff;                               \  
    asm volatile("#insert inv reg load
"          \  
      "\tl$a\t$s2,0x7fffffff\n"                    \  
      "\tsw\t$s2,0"="=g"{regs[32]}::"2");          \  
    qwritene(MQO,regs,0); test(expression);\})       \  
```

4.2. Tidbits

We had some interesting tidbits in this project.

First, the message passing mechanism took a little time to understand. Two communicating programs do not operate correctly if the writes and reads do not match well. Further, the queue read code should be written so that it does not assume that there is a message in the queue. Originally, we had assumed that `qread` was a blocking read.

Second, we found we could extend the C program with the `asm` directive so that the program can be written without modifying the intermediate assembly file by hand. Combined with a script to modify the assembly code, this technique helped to speed writing programs. We also were able to use it to flag places where hand modification should occur. However, the `asm` directive may cause serious side effects when used incorrectly.

Third, we found several bugs in SimpleScalar. Some them were significant and caused our programs to not operate. One caused early termination of the simulator. Another was a large memory leak which caused segmentation faults. We could execute the programs correctly after we received revised versions of SimpleScalar. At this point, there is still an outstanding memory leak in the simulator.

4.3. Thoughts

This seems like an energy intensive method of verification. At the level we are doing work, there are more efficient ways, such as DIVA, or dual processors, with a rollback whenever they don’t agree. Our method could be useful in a couple ways. One is if efficiency is not important, like computational fabrics. Another is if there were multiple main processors, to amortize the cost of having a verifier processor. Only if you have a large amount of computation between invariants do you have reasonable efficiency.

Next, invariants are not easy to come up with. We talked about several programs, and the one we chose was easy to create invariants for. However, some of the others were not. The invariants reflect the structure of a program and are difficult to generate automatically. With proof-carrying code, much of the proof is inherent in the code. The choice of a safe language provides most of the proof of safety as part of the code itself. Even in proof-carrying code, unusual conditions were written by hand. We also wrote invariants by hand, but we tried to minimize human intervention as much as we could. A way must be come up with to lower the amount of human thought needed for invariants.

5. Summary
The decreasing feature size of processors makes it necessary to verify the execution of a processor. There have been some efforts to address the problem, like DIVA. We approached this problem by simulating a multiprocessor system in SimpleScalar. We configured the simulator and wrote application programs running on it. Although our applications were small scale, they were sufficient to show that a processor can execute instructions and be verified by another processor. The increased overhead by passing messages were negligible as the size of programs grows. And the increased overhead by error handling was small when the errors occur on a rare occasion. We expect our idea can be applied to a real application if we can rollback memory values.

6. Acknowledgement

Our work is based on the SimpleScalar multiprocessor extension, which Mark Whitney wrote and is currently working on. He helped us to configure SimpleScalar and fixed bugs.

References