Ben Panning
ID: Bjp2122 (CVN)
COMS 4115: Compilers
Project Proposal

MOC-V

A Language for Memory Ordering and Coherency Validation

Summary

Increasing complexity and parallelism within the design of x86 processors has created a requirement for ever more sophisticated methods for validating memory ordering and coherency of those designs.

There are two main methods currently used to ensure functional correctness of x86 processors in terms of memory ordering and coherency. These methods are random instruction generators and focused tests.

Random instruction generators generate sequences of x86 instructions, and compare the resulting memory map to a simulated result for the same sequence. This has the benefit of a high degree of randomization, however many x86 logic bugs can be found much more quickly through directing aspects of the test sequence.

Focused testing involves creating a specific sequence of instructions to hit specific test cases. This method has the benefit of being directed towards cases of interest, however lacks the randomization required to be effective against wide ranges of potential logical bugs.

A third, as of yet untested, potential method for creating sequences of code to test memory ordering and coherency would be to develop a language that has the benefit of providing engineers with the ability to easily define a directed memory test with the added benefit of being able to describe a high-degree of randomization.

Language Requirements

The proposed language is one that allows users to easily create stressful code sequences to test memory coherency and ordering. As such, it must meet the following requirements:

1. Ability to specify ranges of memory to be tested and easily access this range
    a. Physical address start location and range
    b. A mask like feature for specifying patterns of memory to be tested
        i. For instance, test every odd dword in a 32KB range – allowing random accesses to the even odd dwords to determine if false sharing of cache lines effects memory coherency
    c. Easy access to test memory space using pointers
        i. Cast-able to types signifying data size (using ordinal byte notation)

        ii. "Safe" pointers in that the address is done mod the segment size, meaning "over-running" the buffer wraps around instead of generating a protection fault
2. Ability to provide memory checking to the users specification
    a. Checksum and memory range compares
    b. Simulation done by compiler for checksum comparison – no need to be done at run-time
3. Ability to specify regions for code sequence randomization
    a. User specification for instructions, memory ranges to interact with, ordering of instructions, probability of instructions occurring, etc
    b. User specification for regions in which to generate random code sequences – it is not arbitrarily done

The language thus breaks down into three basic requirements: the ability to interact at a low level and create stressful memory algorithms, the ability to check the results of said algorithms, and the ability to specify sequences and regions for randomization.

Implementation Scope

To make this project something that can be accomplished within a single semester, the scope of its implementation will be narrowed. Here is a list of items that are intended to be included with this project, as well as a few restrictions:

1. A lot of work will be put into the back end of the compiler – it is intended that the compiler will produce an assembly file, which can be assembled and linked to create an executable
2. The generated assembly will be compatible with real mode (virtual 8086 mode). This will include the extended 32-bit register set.
3. All programs created in this language will be run in Virtual 8086 mode or real mode. This will allow programs to be run either from a Windows command prompt or from a DOS prompt (when booted to DOS in real mode). A real implementation of this language would of course support protected mode – but the complexities of getting into protected mode and setting up segmentation and paging will prevent it from being part of a language that must be devised with a compiler in one semester.
4. The range of instructions that can be used within the random sequences will be limited to a set of memory operations and a set of arithmetic operations. These instructions are as follows:
    a. xchg (atomic register/memory swap)
    b. inc (atomic register/memory increments)
    c. dec (atomic register/memory decrements)
    d. mov (simple register/memory copy)
    e. add (register addition)
    f. xor (register bitwise exclusive or)
    g. and (register bitwise and)

5. The coherency aspects of this language would require the ability to synchronize and run many different logical CPU's at the same time. Since we intend to run this in a DOS environment (with the OS sitting in the background), this is not really possible – so multi-threading will not be supported. This will be more of a "proof of concept" to see if the idea is feasible. If a language like this were ever implemented, it would have the benefit of a small init sequence to get it into protected mode and would then include its own thread synchronization during each test – this will not be supported in this project, though.

Coding Examples

The following are a number of code snippets meant to illustrate how the language would be used. The language is similar to C, and comments have been interspersed with the code to explain what is happening:

```
// Declare a range of memory to be tested (mrt is a type - memory range
// for testing)
mrt test_range(0x1000) = 0x0;  // Declare a 4KB test range, initialize
                               // it to 0x0

// Declare a set of pointers to that memory range for test
ord1* ptr1 = test_range:0x0;  // 1-byte pointer to the start of the
                              // test segment
ord2* ptr2 = test_range:0x100;  // 2-byte pointer to 0x100 bytes into
                                // the test segment
ord4* ptr4 = test_range:0x400;  // 4-byte pointer to 0x400 bytes into
                                // the test segment

// Perform a series of writes into the test area
/*
 * Pointers can only be set to an address within a memory range setup
 * for testing.  These ranges can be setup for automatic consistency
 * checking.  Non-pointer declarations can not be setup for automatic
 * consistency checking, but they can be convenient for constructing
 * the algorithms used for memory testing
 */
for(ord4 count = 0; count < 0x1000; count++)
{
      // Set the memory pointed to by ptr1 to 'count', then increment
      // ptr1 by 1-byte
      (*ptr1++) = count;
      // Set the memory pointed to by ptr2 to 'count', then increment
      // ptr2 by 2-bytes
      (*ptr2++) = count;
      // NOTE:  It is intended that pointers be "safe" within
      // this language in that they can only point to tested regions of
      // memory, and when incremented past the end of a region they
      // wrap around to the beginning of the buffer
}
```

```
// Define a random code block (rcb) to generate random sequences of
// instructions
rcb random_code1
{
      ord4* ptr = test_range:rand;  // Random offset into test segment
      for(ord4 count = 0; count < 4; count++)
            *ptr++ = *(ptr+0x16);
}

// Inserting a sequence of random instructions into a test
[(17)random_code1 & (55)random_code2 & (12)random_code3];
// 17% chance of rcb1 AND 55% chance of rcb2 AND 12% chance of rcb3
```

Full Code Example

```
mrt test_segment(0x1000) = 0x0;

// Declare a random code block for random memory copies
rcb random_code1
{
      ord4* ptr = test_segment:rand;
      for(ord4 count = 0; count < 4; count++)
            *ptr++ = (*ptr + 0x10);
}

rcb random_code2
{
      ord2* ptr = test_segment:rand
      ord2 temp = 0;
      for(ord4 count = 0; count < (rand%30); count++)
      {
            temp = *(ptr + 30);
            temp = *ptr + temp;
            *ptr = temp;
            // Random code blocks can contain other random code blocks
            // that are evaluated within the main program – that is,
            // each time “random_code2” block is inserted into the test
            // program it will generate a new “random” sequence for the
            // following
            [(70)random_code1];
      }
}

// The test program would start at “main”
void main()
{
      ord4* ptr = test_segment:0;
      // 30% chance of random_code1 being inserted here, AND 70% chance
      // of random_code2 being inserted here
      [(30)random_code1 & (70)random_code2];
      for(ord4 count = 0; count < 0x400; count++)
      {
            *ptr++ = (*ptr)++;
            // 100% chance of either random_code1 being inserted here
            // OR random_code2 being inserted here, but not both
            [(100)random_code1 | (100)random_code2];
```

```
        }

        // The code to check for consistency would automatically be
        // generated at the end of the main test function
}
```