Project Report

Software Techniques to reduce instruction duplication for soft errors

Project Members:

Muhammad Mutahir Latif (mmlatif@ncsu.edu)
Ravi Ramaseshan (rramase@ncsu.edu)

Project Website: http://www4.ncsu.edu/~rramase/RED714.htm

Solved Issues:

Ravi:

- OpenImpact compiler integration with the fault tolerant module almost complete.

Muhammad:

- SEU injecting and timing framework developed.

Identified the following constructs to reduce instruction / data duplication:

1. Switch statement
2. For-loop
3. Pointer traversal
4. Function parameters
5. Register packing

Open Issues:

- Getting the OpenImpact compiler to generate fault tolerant binaries or assembly code.
- Integrating techniques into the OpenImpact compiler.

Steps for attacking them in the future:

- Turn off optimizations happening in the later optimization phases of OpenImpact so that the redundant code is not eliminated.
- Understand the IRs of OpenImpact and the code generation process and integrate the identified techniques into the compiler.

Schemes implemented

Switch statement

Use of the switch programming construct is used frequently in programs. In the switch construct a variable is checked and the value is matched with the case values specified. The program flow then moves to the block of code assigned to that particular case. Under the SWIFT and EDDI schemes the variable that is switched on has to be duplicated and at each case block the original and duplicated variable must be the same as shown in the code snippet below

```
enum enum_type {A = 0, B, C, D}
switch(x)
{
     case A: if(x!==xp)
                  error;
             break;
     case B: if(x!==xp)
                  error;
             break;

     case C: if(x!==xp)
                  error;
             break;
}
```

From the code snippet we can see that for each block the variable x has to be checked with its duplicate in order to verify that the switched variable x or its duplicate did not fault due to a soft error.

It is a common practice to declare the possible case values as enumerated type in C/C++. The enumerated values are usually assigned incrementally or are assigned unique integer values. If the number of enumerated values can be bound by the word size of a platform and in cases where the switch statement does not use a default case for invalid values, then the whole switch construct can be transformed into one where the switch variable need not be duplicated and therefore not compared with a duplicate to verify correctness. In order to do that the enumerated values would have to be changed to a format where each enum value has only one bit set. These new enum values could be used in the case statements. This way any SEU that occurs on the switched variable will have non 1 number of bits set. This would allow us to distinguish a faulted variable from a correct variable, and hence we could add a default block in the cases, which would only execute if the there was a soft error.

```
enum enum_type {A = 0x1, B=0x2, C=0x4, D=0x8}
switch(x)
{
     case A: break;
     case B: break;
     case C: break;
     default:
          error
}
```

For loop

By using the single bit set setting technique to differentiate between faulted and correct variable, as mentioned in the switch model, we can transform loops that are bounded by the word size of the platform into loops where we would not need duplicates of the iterator variable. For for loops The EDDI and SWIFT techniques would duplicate the iterator and perform the same operations on it as the original iterator, which can incur a slight cost as shown below

```
for( i=0 , ip = 0 ;
     i < 10;
     i++, ip++)
{
     if( i != ip )
          error
     /* do work  */
}
```

if the upper bound of the for loop is less than the word size of the platform and the iterator is not used extensively inside the for loop block, then the iterator traversal can be transformed from an increment to a single bit shift. By keeping the iterator as a variable with only 1 bit set we can discard the duplicates in EDDI and SWIFT. A faulted iterator can be detected if it has more than one bit set and this can be done by ANDing a number with its 2’s complement to get a non zero value as shown in the code snippet below

```
for( i=1 ;
     i <  0x800;
     i = i <<1)
{
     if( i ^ ( i – 1 ) )
          error
     /* do work  */
}
```

Pointer Traversal

One programming model which is used very frequently is to allocate a contiguous structures in memory and then use pointers to traverse the structures in that array. In EDDI and swift techniques the pointer would have to be duplicated and verified by comparison with the duplicate.

```
struct{char a,b} st
st list[100], *ptrav, *ptrav`;
for( ptrav = list , ptrav` = list;
     ptrav<&(list[99]);
     ptrav++ , ptrav`++ )
{
     if(ptrav != ptrav`)
          error
}
```

If the size of the structures traversed can be altered so that they are never a power of 2 then we can use pointer subtractions to determine if a SEU occurred on the pointer. We can prove that if the size of the structure is not a power of 2 then the difference between 2 pointers pointing to different structures in a contiguous array will always be a multiple of the size of the structure. Any SEU on the pointer will make this condition false and we can detect this occurrence.

```
struct {char a,b,stuff} st
st list[100],*ptrav;
for( ptrav=list;
     ptrav<&(list[99]);
     ptrav++)
{
     if( ! (ptrav-list) % sizeof(st))
          error
}
```

Proof:

Let the size of the element be p, so the difference between the address of the element being accessed and the start of the array is pk (for some integer k).

If a bit flip does not occur then the value of the difference (pk) will be divisible by p.

The claim here is that after a bit flips in the pointer neither of the possible new values of the difference from the start, viz. (pk+2^m) or (pk-2^m), are still divisible by p if p is not a power of 2.

Proof by contradiction:

Let p be an integer which is not a power of 2.

Assume that the SEU differences were divisible by p then

pk (+/-) 2^m = pk'

=> p (k (+/-) k') = 2^m

=> (k (+/-) k') is a power of 2 and
p is a power of 2.

However, p is not a power of 2.

Hence the SEU differences are not divisible by p where p is an integer which is not a power of 2.

In this case we can assume that we have a bit of hardware support to calculate whether the difference between 2 pointers is a multiple of the size of a struct. The only potential problem that we could face in implementing this scheme would be an increase in the memory utilized for the structure arrays and a penalty incurred due to misaligned loads (structures and no longer multiple of 2s)

Register Packing

Just as intel uses register packing to save on operations performed on multiple memory locations, we can also use the same technique to introduce fault tolerance for variables whose values can be bounded by the maximum value possible if only half of the bits of the platforms word size are used. Another condition for using such a case would be that the operations performed on the MSB would never have an effect on the LSBs, for example binary operations, unsigned addition etc.

Take for example the following snippet of code which contains EDDI and SWIFT techniques for fault tolerance. Now we can see that the values of x,y,z will never use their upper half MSBs (assuming 32 bit integer) and the operations performed on it are such that the upper half MSBs will never interfere with the lower half LSBs.

```
int x = x'= 0xFF00,y= y'= ADEF,z,z';
z =  x^y
z' = x'^y`
if(z!=z`)
     error
```

We can transform this code into the following

```
int x'= 0xFF00FF00,y= ADEFADEF,z;'
z =  x^y
if(16MSB(Z) != 16LSB(z) )
     error
```

By packing the duplicates into the variables unused bits we save on the duplicated operations and in some cases on duplicated memory.

Function parameters

In a lot of cases in programming functions use arguments that are declared as constants, meaning the parameters passed will never change in the function, rather they will only be used to calculate other values. In order to verify that the arguments passed to such functions do not change while a function is called EDDI and SWIFT techniques would duplicate the arguments as shown in the example below.

```
int foo(int a,int b, int c, int a`,int b`, int c`)
{
     int x , x`;
     x= a+b+c;
     x' = a' + b'+ c';
     if( x != x`)
          error
}
```

These extra parameters passing can incur some cost in performance since the arguments have to be pushed in the stack. One way of solving this problem would be to XOR the const arguments and only pass that as a check for fault tolerance. At the end of the function we could again perform the XOR operation on the arguments and compare with the passed XOR hash. This could save us a number of operations and reduce some of the duplications

```
int foo(int a,int b, int c, int XORabc)
{
     int x,x';
     x= a+b+c;
     x' = a + b+ c;
     if(XORabc != a^b^c)
          error
}
```

SEU injection framework

The SEU injection framework is an application in which certain types of variables can be injected with a soft error, to test the correctness of EDDI based fault tolerance and our schemes fault tolerance.

The framework supports SEU injection into Global variables and the variables declared in main function. The framework keeps a list of structures, where each structure contains the name of the variable to be injected with SEU and the address in memory where it is located.

In order to set up the data structure the user has to implement 2 macros. The macro ADDVARIABLETOLIST(num_vars) will allocate memory to store “num_vars” number of variables in its list. In order to add any variable or memory location to the list the user has to use the macro SETUPVAR(varname). The macro will add the information of the variable in the variable list.

At the application start up, the program will display the variables, their addresses and the index in the list where the info is stored. It will next spawn a thread which will wait for a socket connection and receive the index of the variable on which to perform a bit flip. The thread will then flip a bit of the variable at the index specified. The thread will resume back the waiting state to listen for more incoming socket connections. The main program after spawning the thread will then resume execution. In order to provide a large window of opportunity for performing an SEU the main thread will keep executing in an infinite loop.

To send a command to the framework for injecting SEU another small application has to be executed. It takes 2 program arguments: The hostname of the system where the framework is running and the index of the variable which is to be corrupted. The application will create a socket connection with the framework and send the index of the variable to be corrupted. To perform a SEU on the variables randomly the user can specify -1 as the index.

OpenImpact SWIFT Integration

Code Used

We inherited the src/ directory of the IMPACT sources used for the SWIFT project from UIUC through Prof. F. Mueller. The base version of the IMPACT compiler used was *unknown*.

We downloaded the latest version of the OpenImpact compiler (openimpact-1.0rc4.tar.bz2) from the UIUC OpenImpact website and integrated the changes to some files as well as the Fault Tolerance module into our OpenImpact source tree.

Problems Faced

Due to the inavailability of the original code base version used in the SWIFT project, upgrading to the latest version of OpenImpact was very difficult because there was no base version with which we could compare the SWIFT-specific changes made.

The build process of the OpenImpact compiler changed significantly (use Makefile.am instead of Imakefile) from the IMPACT compiler used to the SWIFT and the Release Candidate 4 which we used.

Unclear use of the Fault Tolerance module in the OICC driver to generate SWIFT code. Without this information the optimization stages of the OpenImpact compiler optimizes out the redundancy introduced by SWIFT.

Integration ChangeLog

Lcode/Lcode/l_lcode.c
* (L_init_symbol): Adding symbols for L_MACRO_CF and L_MAC_CF.
* (L_macro_name): Adding a case for L_MAC_CF.

Lcode/Lcode/l_pred_flow.c
* (PF_add_standard_operands): Inserting some code added by GAREIS.

Lcode/Lcode/r_dataflow.c
* (D_delete_DF_dead_code): Inserting some code added by GAREIS.

Lcode/codegen/Ltahoe/phase2_func.c
* Inserting some ifdefed headers added by SZU
* (Ltahoe_remove_dead_pred_defs): Inserted a printf added by GAREIS,
* (O_process_func): Inserted some printf looking LTD statements.
* Inserted a condition L_do_software_pipeline for a call to L_local_redundant _load.

* Added some indef 0 code by SZU
* Have not changed a call to SM_bundle_fn to SMH_bundle_func.
* (O_finalize): Inserted a SMH_finalize call.

Lcode/codegen/Ltahoe/phase2_memstk.c
* (O_update_stack_const_in_prev_oper): Inserted a lot of printf type statements.
* (O_update_stack_references): More printf type things by GAREIS.

Lcode/codegen/Regalloc/r_regalloc.c
* (R_spill_all_virtual_registers): Another printing type thing inserted.
* Lots of comments involving GAREIS.
* There seems to be a lot of data structure changes. Like sets being replaced by lists. So although the diff looks big, most of it hadn't (I hope) to be merged in.

Lcode/sched/SM/sm.c
* (SM_init_sm_op_fields): Added OP_FLAG_CHK flag.

Lcode/tools/FT
* Copied and renamed this directory from Linduct.
* Replaced l_codegen.c with that from the inherited SWIFT Fault Tolerant module.
* Added FT to the build process by introducing FT wherever Linduct was used.

params/STD_PARAMS.IPF-MCKINLEY
* (FT) Added standard parameters for the FT module using the inherited code.

Status

The SWIFT code base has been upgraded to the latest version of OpenImpact. The SWIFTed OpenImpact builds and installs cleanly. The Fault Tolerant module also build generating the program FT. If we supply an Lcode file to this FT program, we can see (in Lcode) instruction duplication being generated.

Future Work

To find out which optimizations need to be turned off so that the redundancy introduced by SWIFT does not get optimized in the later phases of compilation.

Hints: "Turn off partial redundancy elimination in the last phases of Impact."

New Techniques Applied: Bubble Sort

Aim

To apply our instruction / data duplication techniques to a real program apart from the micro-benchmarks those were used to verify the feasibility and utility of the technique.

The objective of this experiment was to illustrate the utility of our optimizations over regular instruction duplication techniques in more general programs.

Experimental Setup

We used SWIFT as our base instruction duplication technique. We hand-coded (in C), SWIFTlike instruction duplication. We did not incorportate the control flow checking in SWIFT.

Out program was compiled using GCC at optimization levels `O0` and `O1`. For the pointer scheme, we passed the `-fpack-struct` so that the compiler does not pad the structure so that we could control the size of the structure as demanded by the technique.

We used the timestamp counter register on Intel machines to obtain the execution times of our experiments.

Program

(`bubble_swift.c` and `bubble_new_schemes.c`)

We chose bubble sort as the macro-benchmark to which we were going to apply both SWIFTlike transformations as well as our optimizations over SWIFT.

We used the for-loop technique for the induction variables in the two for-loops. Pointers were used for manipulating the array elements so that we could employ our pointer technique.

We use the worst case of bubble sort for our analysis.

Results

The original structure contained a single integer and was of size 4 bytes. The padding field in the results below show the number of bytes used to pad the structure so that its size is not a power of two.

	SWIFT		NEW TECHNIQUES			
Optimization	O0	O1	O0	O1	O0	O1
Time	10282	8223	15329	9802	8847	8104
Padding	-	-	1	8	1	8

Analysis

The results for a particular optimization level suggest the following:

a. If we pad the structure with the minimal number of bytes required for the pointer-scheme (ie. 1), we see that our technique exhibits a worse runtime than the corresponding SWIFT program.

b. However, if we pad the structure such that its size is a multiple of 4 (in this case the smallest such number was 8), our scheme was faster (4.6% and 1.5% for O0 and O1 respectively) than the corresponding SWIFT program.

This behavior can be attributed to the mis-aligned loads that occur in the case that the structure size is not a multiple of 4.

Inferences and Conclusion

The constructs can be used innovatively in conjunction with each other to give more optimized code than if used separately.

The minimal padding pointer scheme would prove beneficial only when the size of the structure matches the alignment constraints of the target architecture.

However, we assume that most programs would involve structures of larger sizes. We also note that the density of numbers which are powers of two decreases rapidly as the numbers become larger. Hence the probability of us having to pad structures with unreasonably large padding should be low. Thus we conclude that the pointer scheme would be beneficial even on architectures with alignment constraints.