CSE 438Embbeded Systems Programming: Binary Translation

Project Report

Ajey Achutha and Mahesh Kumar M

12/11/2012

Table of Contents

1 Introduction:

Binary translation is translation of code from one instruction set to another. The Binary Translation is very much necessary when a program needs to be ported from one architecture to another.
Binary translation also plays vital role in debugging, testing and optimization process of the application code.
The Binary Translation can be achieved in two ways: Static Binary Translation and Dynamic Binary Translation.

- The Static idea refers to the idea of converting the code to target architecture without having to run the code first. The static binary translation is done only once to all programs. Only if the target platform changes then static translation should be run on the program code. The details regarding the implementation of static binary translation are explained in greater depth in this document.
- Dynamic Binary Translation refers to the idea of translating the code to target instruction set during the run time of the program. Dynamic recompilation is explained in detail in this document. Also the pros and cons of both the approaches are also discussed.

2. Static Binary Translation:

The static binary translation takes the whole program code and target instruction set at once and translates it to the target architecture.

The major difficulties in building a binary translator or the things to be kept in mind during the design are:

1. Performance lost during translation should be kept to minimum.
2. As the instruction set is increasing day by day, writing a code to convert the code from one instruction set to another is itself a significant challenge.
3. The design is very challenging as the system should be quickly retargeted to new architectures.

The Peephole is one of the technique which is discussed in the paper solves the above problems. The Peephole rules are simple; replace a set of instructions with another set of instructions using pattern matching. Peephole rules are used in compiler optimizations also to replace set of sub optimal instruction with another equivalent set which is faster. The example is as follows: ld [r2]; addi 1; st [r2] => inc [er3] { r2 = er3 }
Reference: http://theory.stanford.edu/~sbansal/pubs/osdi08_html/index.html

In this example as we can see that left side 3 instructions is equivalent to only one instruction in the right side. The left part is RISC architecture and right part is CISC architecture. The equivalency is tested by checking the state of registers at the start of the code and at the end. The register r2 in RISC is mapped to er3 in CISC.

It is very difficult to write all patterns occurring in both architectures. So the model should be able to learn. This is achieved by using superoptimization. In Superoptimization the target instructions are enumerated and different patterns are generated by using the target instruction set. Since the enumeration can take exponential time, which might require ages to compute. So the enumeration is controlled by cutting down some of the patterns which can never be included to achieve the source instruction set equivalency. Since this is static binary translation, it should be fine to perform enumeration. Once a pattern is detected it can used in future by just simple pattern matching used in Peephole technique.

Design of Peephole Superoptimizer:
The design of Peephole Superoptimizers has three phases:

1. Harvester: This module collects the set of instructions from the target architecture set.
2. Enumerator: This module generates all possible enumerations of the target instructions collected.
3. Lookup table: This module keeps track of the optimized match found in target instruction set.

The conversion becomes much easier after the above things are built. The Enumeration is done for a set of maximum of 5-6 instructions because as this number increases the possible combinations increase in exponential order. After the above details are gathered, the conversion of instructions becomes much easier and faster. Since the enumeration of the target instruction set is exponential, a strategy should be implemented to cut down the number of enumerations.
This is required to reduce the amount of search to find equivalent instruction set. An example for this may be say in the source instruction set there is an ADD command, while searching equivalent instruction in target set, one should never consider the DEC (Decrement Instruction) command or its enumerations. These kinds of rules can reduce the search time by a big factor.
According to statistics from the paper it almost took 2 full days to generate all enumerations of instruction sets of length 3 for x86 architecture using a single processor computer.

The efficient usage of CPU registers is also desired. Since the number of registers may vary between architectures a careful consideration should be made for this regard. The table below shows the register map convention followed:

Register	Description
r1 -> eax	Register mapped to another register
r1 -> M1	Register mapped to memory location

Ms-> eax	Mapping memory location in source code to register in target architecture
r1->eax r2 ->eax	Invalid combination or mapping of registers.
Ms->Mt	Memory mapping

The below table shows the two scenarios of register mapping:

Source Inst	Registers	Mapping	Target Inst
mr r1,r2	r1 , r2	r1->eax r2->M1	mov1 M1, eax
subfc r1,r2,r1 addc r1,r1,r3	r1,r2,r3	r1->eax r2->ecx r3->edx	sub1 ecx,eax add1 edx,eax

There is cost encountered in moving from stage 1 mapping to stage 2 mapping. In stage 1 ‘r1’ register is mapped to eax, r2 mapped to M1. In the second stage ‘r2’s mapping needs to be changed. The cost encountered for the above change is generalized as shown below:

$$Cost(M) = cost(T) + \min(cost(Pi) + switch(Pi,M))$$

The cost(M) is the best cost register map. The cost(M) should be minimized for set of instructions of selected sequence length. Pi represents the predecessor register map scenario. Switch is between two stages the cost encountered. T is minimum cost translation for instructions.

3. Dynamic Recompilation:

The dynamic recompilation is found in many emulators and virtual machines. This feature enables to recompile certain portion of program during execution. The recompilation benefits because during recompilation the system modifies the code in such a way that it runs efficiently in the target platform. This advantage is not present in static compilers. The dynamic recompilers convert the machine code to target architecture during the runtime.

The source platform code to move the string from one memory location to other

```
beginning:
        mov A,[first string pointer]          // Copies the pointer location of source
        mov B,[second string pointer]         // Copies the pointer location of destination
loop:
        mov C,[A]                             // Copy the character to temporary variable
        mov [B],C                             // Copy from temporary variable to destination location
        inc A                                 // point to next character
        inc B
        cmp C,#0                              // check for null character
        jnz loop
end:
```

The target platform code which is using the MOVS instruction available in target platform and performs the string copy operation efficiently:

```
beginning:
        mov A,[first string pointer]          // Get the source address
        mov B,[second string pointer]         // Get the destination address
loop:
        movs [B],[A]                          // copy 16 bytes or 32 bytes depending on architecture
        jnz loop                              //Loop ends when a NULL character is encountered.
end:
```

Reference: http://en.wikipedia.org/wiki/Dynamic_recompilation

During dynamic recompilation the target platforms capability is utilized efficiently. In the above scenario the MOVS instruction which copies 16 or 32 bytes of data at once is utilized. The program gets modified to short and efficient code.

4. Pros and Cons of two types:

Static Binary Translation: The major advantage of this idea is that, the program execution is fast. The reason being unlike dynamic translation, the static translation translates code at once and during execution of the program the CPU does not have to work on translation. This make program run faster. The translation needs to be done only once if the target platform is constant.

Dynamic Binary Translation: The idea has higher level of detail when translation needs to be done. There will be indirect branching in the code which is only possible to know during run time.
But there is lot of overhead in this method as the code has to be translated multiple times whenever the code is run.

Binary Instrumentation for control of program execution path using PIN:
Pin is an Intel framework which can be used for dynamic program analysis. Pin uses dynamic binary instrumentation. The Binary Instrumentation is a way to monitor programs performance by inserting code during run time. The execution path change in the program is not part of tool but can be achieved by adding the code. This also enables us to control the thread executions. We can build our own scheduler using this feature.

The below code illustrates way to change the execution path during execution of the program:

```
if (INS Address(ins) == 0x400000)
{

INS InsertCall(ins, IPOINT BEFORE, AFUNPTR(JumpToFunc),
IARG CONTEXT, IARG END);

}
/* binary instrumentation tool - analysis routine */
void JumpToFunc(CONTEXT* ctxt)
{

PIN SetContextReg(ctxt, REG INST PTR, Func);
PIN ExecuteAt(ctxt);

}
/* application code */
int Func() { ... }
```

Reference: http://www.eecs.berkeley.edu/~krste/papers/pin-wbia.pdf

In the above example a call to JumpToFunc method is inserted at address 0x400000. Actually when executing the program sequentially if it hits the address 0x400000 instead of executing the instruction present in this address, a call to JumpToFunc method is executed. Inside JumpToFunc method the context is changed and Instruction Pointer is set to the location of the custom function Func(). This Func() can be used to debug the programs by checking the state of registers or checking other things. PIN SetContextReg and PIN ExecuteAt method from the PIN library help us in achieving this.

The PIN also has methods to save CheckPoint. PIN_SaveCheckPoint which stores the current state of the program and this might be helpful for going back to normal program execution after debug method is executed. So the procedure would be set/save the checkpoint and change the

context to different debugging method, after debugging change the context again to the saved checkpoint.

User Level Threads: The Binary Instrumentation Tool can be used to implement the thread library. The implementation should be able to schedule threads depending on processor, cache and memory. In a multiprocessor system if there are application threads running on host Operating system then the relative execution rate of the threads depends on the host's memory, processor availability and Operating system scheduler. The traditional instrumentation inserts the code in the threads affecting the relative rates of execution and increases latency. So control over execution of threads can solve the problem.

The custom thread execution can be divided into majorly two things, thread manager and thread scheduler. The main responsibilities of thread manager are to implement the actions like thread creation, cancellation, storage and mutexes to control the shared variable. The thread scheduler is designed to be a simple round robin queue with certain other conditions and rules required for scheduling policy. Now these custom threads can be controlled on various parameters like the time of execution of specific thread and relative preemption etc. The scheduler majorly deals with context switching between threads. During switching the threads might be accessing a common memory location. The locks provide safe manipulation of the variables. But the problem with the locks is that, it might result in some race conditions and deadlocks. When multiple threads are accessing and trying to modify same memory location then simply deny the change for one thread and restore the value before other thread starts to use the memory location. So we will track the memory location in three situations, beginning of the transaction, end of the transaction and transition stage. The tool can only checkpoint the processor state but our design should be able to monitor the memory location. All the threads must roll back the value if the commit is unsuccessful. These are the basic criterion to be given extra caution when designing the thread library.

5. Applications of Binary translation

Binary translation is applied for varied purposes. The basic idea behind the binary translation application is the concept of emulation/conversion of the source code to a target version. The conversion will be performed based on the machine code translation which is the basic concept of the entire process.
Binary translation can found in the following applications,

1. Memory and profiling tools for example Valgrind,
2. Virtualization of platforms such as in VMware, binary translation is one of the methods of virtualization
3. Emulation of a target system on a host system example, QUick EMUlator – QEMU
4. Distributions that perform simulation of cache, profiling, and tracing the code by making using of the instruction example the Intel pin API.
5. Java virtual machine (JVM)

Before going about detailing the application of binary translation in some of the above application, it is important to know few concepts related to binary translation that are used interchangeably with it.

Binary Instrumentation: More popular as dynamic binary instrumentation (DBI) as the binary translation involved is of dynamic type. Making use of the binary translation and giving it a code tracing capacity and error detection is the concept of binary instrumentation. The binary instrumentation enables application analysis to be built on it. There by giving capabilities for the programmer to analyze their application code and also to monitor the performance of the program by diagnosing the additional details those are fetched in the process of instrumentation, such as side effects of memory allocation, dangling references of pointers, allocation of memory beyond the necessity, accessing memory locations beyond the allocated areas and so on.

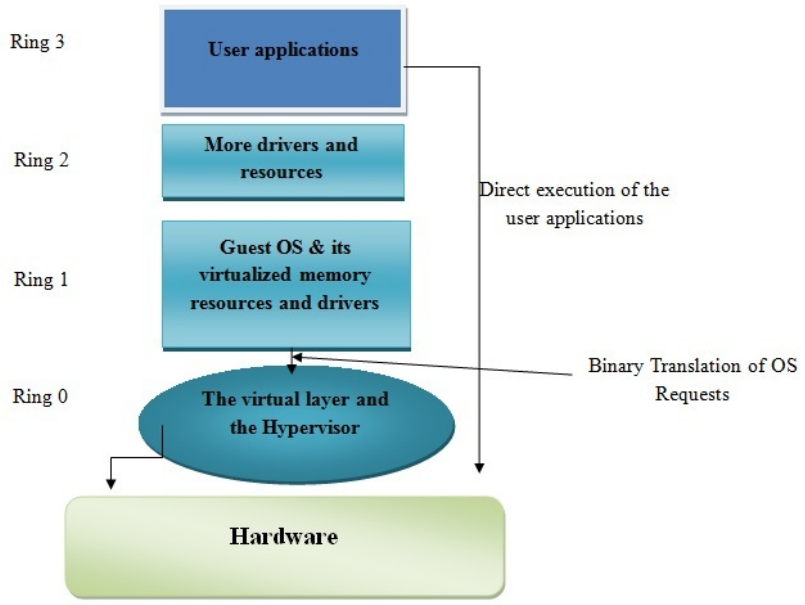
Binary Analysis: Synonymously used as dynamic binary analysis (DBA) for the same reason cited above. The binary instrumentation (or DBI) is by itself is not a tool or an application. On top of the DBI, analyses tools are interfaced or integrated, that are capable of analyzing the outcomes and specific details generated by the instrumentation process. So with the DBA interface, the binary instrumentation becomes powerful in its applications which are based on the binary translation.

In view of the above concepts, Binary Translation is generally also addressed as Dynamic Binary Analysis and Dynamic Binary Instrumentation at the application level of the technology. Though they are not the same still the base idea is the Binary Translation technology.

6. Binary Translation applications in detail

1. VMware and Binary translation

The basic view of the Virtual Machine at the abstract level will be the Implementation of guest operating systems on host system hardware. On single host hardware the need to run different operating systems triggers the need of a virtual machine. These individual operating systems have to be handled in an isolated manner. So to ensure the guest operating system works with almost if not similar performance as the host operating system, running the guest an OS instruction in the privileged mode is of prime necessity. The guest OS instructions those cannot which cannot be run in the lower privilege mode (non-virtualizable instructions) need the hardware abstraction for their execution. This where the binary translation fits the needs and delivers optimum performance. It provides with the virtual layer for the execution of the instructions for the guest OS.



The above picture depicts the how the instructions get executed in a virtual machine environment. Binary translation along with the direction execution mechanism enables the Full Virtualization. The virtual layer provides with a mechanism such that the guest OS completely disconnected from the hardware due to the presence of the virtual layer at Ring 0. The binary translation technology is invariably the main component of this layer which enables the virtualization mechanism by providing the platform for conversion of the OS requests to be executed on the hardware. By this it can observed that the guest OS will not have the knowledge about of virtualization and thus does not need any changes in the OS source. The non-virtualizable instructions are converted into the operating system instructions by the hypervisor and these are cached up for next references. Note that the user instructions are executed as such with the direct execution mechanism. A small note on the hypervisor – that it is the architecture on which the guest OS is installed as an virtual application layer, also the hypervisor will have the direct access to the hardware resources there by the performance is optimized along with other important features delivery. The four level hierarchy is provided by the hypervisor setup of the virtualization technique.
Virtualization using Hardware assist and using OS Assist (Paravirtualization) are the two other techniques of achieving virtualization.

Advantages of this technique over the other techniques of Virtualization are

1. As the hypervisor takes care of instructions translation, hardware assistance or the operating system assistance is not needed in this type of virtualization technique – this is the important aspect that distinguishes this type from others in virtualization and this is possible due binary translation.
2. To answer the question how binary translation – it is evident that the technique

provides the emulation of the instructions from the source instruction (guest OS instruction set) to target instruction sets (the native system hardware).

3. The guest OS is virtualized or can run on the native hardware, moving data from one guest OS to another would be easier (migration is easier to be precise), thus so is the portability aspect.

Few virtualization products as cited in the VMware website that are built on this technique are the virtualized products of VMware and also the Microsoft Virtual Server.
Thus it can be visualized that Binary Translation as a technology is very useful and effective in cases of applications related to Operating Systems and virtualization techniques.

2. Binary Translation in Valgrind:

Valgrind is profiling and debugging open source tool issued under general public license available at www.valgrind.org. The Valgrind implements dynamic binary translation and is interfaced with the dynamic binary instrumentation coupled with dynamic binary analysis capabilities to provide analysis of application code.
Breakup of the tool is short

1. Dynamic Binary Translation: The tool converts the application code from the source code type to the target code type using the binary translation.
2. Dynamic Binary Instrumentation: The instrumentation part of the tool is achieved using the shadow values that the tool builds around the input code.
3. Dynamic Binary Analysis: The tool is a reflection, perhaps more than the reflection, of the resources that the input program creates and provides with some useful analysis on the application code depicting the possible vulnerabilities in the code or by indicating possible corrections in incorrect vulnerability analysis.

Implementation of Binary Translation in detail – the shadow values that needs to be created for the DBI, the input code is converted into the machine level code and then the needed analysis is performed on this machine code. The memory registers that get involved in the process of execution path of the code are analyzed. This process corresponds to the first part of binary translation part as the input is converted into the machine level code for analysis.
The DBI is then built upon the shadow values of the memory registers, the read write operations and system calls. The allocation and deallocation of the memory uints are analyzed for further interpretation and possible mistakes or wrong doings in the code.
DBA is implemented on the shadow values and on all the memory registers and their side effects, to arrive at the conclusions/comments about the input program. The tool incorporates all the analysis part into the machine code obtained in the DBI phase and prepares the final machine code on which the analysis is completed. Once the machine code is obtained which contains all the portions of DBA, the code is converted back to the source code type and this forms the other part of the binary translation.
The tool is built on just-in-execution architecture and has eight phases of internal conversions and analysis portions. To indicate the capabilities of the binary translation it would be worth a mention that Valgrind can handle signals and be the master in case of the threads commanding that only one thread can run at any given instance.

Example of Valgrind usage on a short c program:

```
#include <stdlib.h>
int main()
{
        char *chararr = malloc(25);
        int i,k,temp=221;
        if(i==0)
                if(k==-1)
                        temp=temp*3/7+8;
        k=23;
        for(i=0;i<25;i++){
                *(chararr+i)=k;
                k=k+3*4;
        }

        chararr[35]='a';

        int *an_int_array;
        for (i=0; i < 10; i++){
           an_int_array = malloc(sizeof(int) * 162);
         }
        return 0;
}
```

Clearly there are few visible errors in the above code. If these are error are placed in different parts and in large code file they would almost be hidden. Also, when the code is compiled it does not provide any errors, or warning information as depicted in the below pic.

```
maheshm@chathurthi:~/Courses/ESP/valgrind_programs$ gcc -o example example.c
maheshm@chathurthi:~/Courses/ESP/valgrind_programs$ ls
example  example.c
maheshm@chathurthi:~/Courses/ESP/valgrind_programs$ ./example
maheshm@chathurthi:~/Courses/ESP/valgrind_programs$
```

When the Valgrind tool is used on the code the output would looks something like as shown in the picture below. There can few interesting observations made by the tool which could be crucial in the code development and the application implementation.

- ✓ The message **"Conditional jump or move depends on uninitialized values**" corresponds to the line of codes ***if(i==0) , if(k==-1)*** where the integers i and k are not initialized or assigned and the if clause is being used.

- ✓ The message "**Invalid write of size 1**" corresponds to the line of codes ***chararr[35]='a';*** where the allocated memory is for 25 and we are writing an entry at 35th index which is not allocated and may lead to errors in the application using this code.

- ✓ The message "**25 bytes in 1 block are definitely lost**" corresponds error that the memory initialized for the char array is not freed – free(chararr); missing instruction.

- ✓ The message "**6480 bytes in 10 blocks are definitely lost in loss record 2 of 2**" corresponds to the possibly incorrect huge memory allocation in the line of code ***an_int_array = malloc(sizeof(int) * 162);*** and while the same is not being freed again at the end of the program and this is indicated by the message in **Leak Summar** "***definitely lost: 6505 bytes in 11 blocks***".

```
maheshm@chathurthi:~/Courses/ESP/valgrind_programs$ valgrind --tool=memcheck --leak-check=yes ./example
==16134== Memcheck, a memory error detector
==16134== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==16134== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==16134== Command: ./example
==16134==
==16134== Conditional jump or move depends on uninitialised value(s)
==16134==    at 0x400515: main (example.c:6)
==16134==
==16134== Invalid write of size 1
==16134==    at 0x40057D: main (example.c:15)
==16134==  Address 0x51f3063 is 10 bytes after a block of size 25 alloc'd
==16134==    at 0x4C2C66F: malloc (vg_replace_malloc.c:270)
==16134==    by 0x400505: main (example.c:4)
==16134==
==16134==
==16134== HEAP SUMMARY:
==16134==     in use at exit: 6,505 bytes in 11 blocks
==16134==   total heap usage: 11 allocs, 0 frees, 6,505 bytes allocated
==16134==
==16134== 25 bytes in 1 blocks are definitely lost in loss record 1 of 2
==16134==    at 0x4C2C66F: malloc (vg_replace_malloc.c:270)
==16134==    by 0x400505: main (example.c:4)
==16134==
==16134== 6,480 bytes in 10 blocks are definitely lost in loss record 2 of 2
==16134==    at 0x4C2C66F: malloc (vg_replace_malloc.c:270)
==16134==    by 0x400592: main (example.c:19)
==16134==
==16134== LEAK SUMMARY:
==16134==    definitely lost: 6,505 bytes in 11 blocks
==16134==    indirectly lost: 0 bytes in 0 blocks
==16134==      possibly lost: 0 bytes in 0 blocks
==16134==    still reachable: 0 bytes in 0 blocks
==16134==         suppressed: 0 bytes in 0 blocks
==16134==
==16134== For counts of detected and suppressed errors, rerun with: -v
==16134== Use --track-origins=yes to see where uninitialised values come from
==16134== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
maheshm@chathurthi:~/Courses/ESP/valgrind_programs$
```

7. Conclusion:

Binary translation is thus an essential and effective technology which can be implemented in various areas ranging from, but not limited to, operating systems virtualization, to hardware architecture emulation, to profiling and debugging tools. Though they consume lot of memory and include some overhead in forms of Dynamic Binary Analysis and Dynamic Binary Instrumentation, the tools built on the technology are of real time use and can be applied with ease to obtain effective results.

8. References:

http://www.eecs.berkeley.edu/~krste/papers/pin-wbia.pdf
http://theory.stanford.edu/~sbansal/pubs/osdi08_html/index.html
http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf
Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation
http://en.wikipedia.org/wiki/Binary_translation
http://en.wikipedia.org/wiki/Dynamic_recompilation
http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool