Binary Translation
by the concept not by the title

Presenters:
Ajey Achutha
Mahesh Muniyappa
Outline

- Introduction
- Dynamic Recompilation
- Implement Static Binary Translator
- Controlling Program execution using Binary Instrumentation
- At the application level
- Few examples – VM Ware and Valgrind
Introduction

• Defn: Binary translation is translation of code from one instruction set to another.

• Types of Binary Translation:
  o Static Binary Translation
  o Dynamic Binary Translation

• Static Binary Translation
  o The translation is achieved by converting all the source code to target architecture without running the code.

• Dynamic Binary Translation
  o The translation is achieved during the runtime.
Dynamic Recompilation

• Defn: Dynamic Recompilation is a feature which enables the emulators to recompile certain portions of the code during program execution.

• Tasks of Dynamic recompiler:
  o Read the machine code from source platform
  o Transform machine code to target platform
  o Optimize the code to run efficiently in target platform

• Applications:
  o Java Virtual Machine.
  o QEMU - Quick Emulator which enables emulation.
  o VirtualBox
  o Valgrind
Dynamic Recompilation

**Copy one byte at once instruction**

beginning:
  mov A,[first string pointer]

  mov B,[second string pointer]

loop:
  mov C,[A]
  mov [B],C
  inc A

  inc B

  cmp C,#0
  jnz loop

end:

**Copy using movs**

beginning:
  mov A,[first string pointer]

  mov B,[second string pointer]

loop:
  movs [B],[A]

  jnz loop

end:
Implement Static Binary Translator

- Concept of Peephole:
  - Identify the set of instruction.
  - Perform the pattern matching.
  - Evaluate the truthiness

- Example for Peephole:
  - ld [r2]; addi 1; st [r2] ⇒ inc [er3] {r2 = er3}

- Concept of superoptimization:
  - Enumerate the possible rules
  - Pruning the search space are very important
Implement Static Binary Translator contd...

The table showing mapping of the target and source architectures

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 -&gt; eax</td>
<td>Register mapped to another register</td>
</tr>
<tr>
<td>r1 -&gt; M1</td>
<td>Register mapped to memory location</td>
</tr>
<tr>
<td>Ms-&gt; eax</td>
<td>Mapping memory location in source code to register in target architecture</td>
</tr>
<tr>
<td>r1-&gt;eax</td>
<td>Invalid combination or mapping of registers.</td>
</tr>
<tr>
<td>r2 -&gt;eax</td>
<td></td>
</tr>
<tr>
<td>Ms-&gt;Mt</td>
<td>Memory memory mapping</td>
</tr>
</tbody>
</table>
Implement Static Binary Translator contd…

- **Register Map Selection**

<table>
<thead>
<tr>
<th>Source Inst</th>
<th>Registers</th>
<th>Mapping</th>
<th>Target Inst</th>
</tr>
</thead>
<tbody>
<tr>
<td>mr r1,r2</td>
<td>r1, r2</td>
<td>r1-&gt;eax</td>
<td>mov1 M1, eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2-&gt;M1</td>
<td></td>
</tr>
<tr>
<td>subfc r1,r2,r1</td>
<td>r1,r2,r3</td>
<td>r1-&gt;eax</td>
<td>sub1 ecx,eax</td>
</tr>
<tr>
<td>addc r1,r1,r3</td>
<td></td>
<td>r2-&gt;ecx</td>
<td>add1 edx,eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r3-&gt;edx</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Cost}(M) = \text{cost}(T) + \min(\text{cost}(P_i) + \text{switch}(P_i, M)) \]
Static Binary Translator

- Binary Translation using Peephole Superoptimizers
  - Harvester: which collects target instruction sequence.
  - Enumerator: which enumerates the possible instruction sequences.
  - Table: which acts as lookup for the matched sequence.
  - Replace the instructions keeping in mind the cost of the instructions (Number of instructions, number of registers).
Controlling Program Execution using Binary Instrumentation

- Binary Instrumentation
  - It refers to the ability to debug, monitor performance and code tracing.
  - It uses Binary Translation.
  - It inserts extra code to help in the process of analysis.

- Controlling Program Execution
  - Save the context of the current running program.
  - Perform the operations.
  - Come back to previous context.
Program in PIN

if (INS Address(ins) == 0x400000)
{
    INS InsertCall(ins, IPO INT 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() { ... }
Applying the binary translation

- Applications – Intel Pin, Virtual Machines VM Ware, Valgrind, QEMU
- At the Operating Systems level – VM Ware
- A programmer/debugging prospective application of Binary translation – Valgrind
VMWare and Binary Translation

- Virtualization's backbone is the Binary translation.
- Binary translation and direct execution together enable the virtualization.
- Need of binary translation in VM – Ring0, Ring1, Ring2, Ring3 and the Host Computer the system hardware.
- Virtual Machine layer isolates the Guest OS and enables the non-virtualizable instructions execution at the system hardware level.
- At the Ring0 – Binary translation
VMWare and Binary Translation

- User applications
- Guest OS
- VMM
- Hardware, the host computer

Ring 3

Ring 2

Ring 1

Binary Translation of OS Requests

Ring 0

Direct execution
Valgrind and Binary Translation

- Dynamic Binary Translation (DBT) and Dynamic Binary Instrumentation (DBI) used interchangeably but are not the same
- From DBI – Dynamic Binary Analysis (DBA)
- DBA – a tool to shadow (almost reflection) the resources a program accesses into another variable that describes it
- Shadow value like memcheck
- The memcheck – track bit values and indicate potential threats of using undefined bits which later build an entire variable
- Shadow values imply shadow registers (guest and host registers, ThreadState), shadow memory, read/write operations, system calls, allocation and deallocation etc.
- Valgrind tool = combination of Valgrind core (DBT and others) [plus] the tool plug-in (DBI's Instrumentation part)
• Program to be analyzed – client program
• Valgrind process – analysis transaction
• Loads the client program into the Valgrind's process. The executable is used.
• Core disassembles code – intermediate representation IR
• Process the IR to obtain the shadow values
• Converts the code back into the machine code.
Valgrind contd...

• Disassemble and re-synthesize technique
• Machine code to IR
• Original code effects must be clearly stated in the IR, using which the Valgrind accomplishes the entire process
Valgrind: Disassemble and re-synthesize technique

- A total of 8 phases of client program conversion, execution for DBA analysis
  - Phase 1: Disassembly, machine code into tree IR
  - Disassembler – each instruction is independently disassembled into one or more IR statements.
  - Update the guest registers in the `ThreadState` with all the information about the affects
Valgrind: Phase1 example

0x24F275: movl -16180(%ebx,%eax,4),%eax

1: ------ IMark(0x24F275, 7) ------

2: t0 = Add32(Add32(GET:I32(12),
Shl32(GET:I32(0),0x2:I8)),
0xFFFFFC 0C C :l32)

3: PUT(0) = LDle:l32(t0)
Valgrind: D&R Technique

- Phase 2: First level of Optimization, remove the redundant conversion, copy operation etc.
- Phase 3: Convert the tree IR into flat IR
- The conversion implies shadows values arrive,
- Example a memcheck makes the actual instruction look complicated
Valgrind: Phase3 Example

------ IMark(0x24F275, 7) ------
t11 = GET:I32(320)
t8 = GET:I32(0)
t14 = Shl32(t11,0x2:I8)
t7 = Shl32(t8,0x2:I8)
t18 = GET:I32(332)
t9 = GET:I32(12)
t19 = Or32(t18,t14)
t20 = Neg32(t19)
t21 = Or32(t19,t20)
t6 = Add32(t9,t7)
t24 = Neg32(t21)
t25 = Or32(t21,t24)
t5 = Add32(t6,0xFFF_F000C0C:C:I32)
t27 = CmpNEZ32(t25)
DIRTY t27 RdFX-gst(16,4) RdFX-gst(60,4)
::: helperc_value_check4_fail{0x380035f4}()
17: t29 = DIRTY 1:I1 RdFX-gst(16,4) RdFX-gst(60,4)
::: helperc_LOADV32le {0x38006504}(t5)
t10 = LDle:I32(t5)
Valgrind: D&R contd...

- Phase 4: Second optimization
- From 48 instructions to 18 instruction in the previous slide is after the second optimization
- Phase 5: From the flat IR built in Phase 4 build the tree IR
- Phase 6: Instruction selection, using a simple greedy top-down tree matching algorithm convert the IR code into instructions, using virtual registers
- Phase 7: Virtual registers to the host registers conversion
- Phase 8: Instructions are encoded and written into memory
- Finally the code is now ready for execution and analysis
Valgrind: Additional notes

- **System calls: Cannot trace into the kernel, thus,**
  - Save the stack pointer,
  - Guest registers are copied into host registers, PC is not copied
  - System call execution
  - Copies guest registers back out to memory, expect PC
  - Restore stack pointer
  - The file descriptor and memory are pre-checked by the tool and makes it unsuccessful if in case the client program tries to alter the mmap of the tool and does not consult kernel for this
Valgrind: Additional notes

- Threads: Pose a challenge to the tool with respect to the shadow values, data handling is non-atomic and each operation like load and store leads to shadow load and store and so on.
- The kernel chooses which thread to run while Valgrind dictates thread switches and thereby determines which thread should run.
- At a time only one thread which holds a lock can run and the other threads will wait.
- Locking – pipe which holds only one character and the thread running should read the pipe and start running, the thread should drop the lock before a blocking system call or if it has run for a predefined time duration.
Valgrind: Additional Notes

- Valgrind intercepts all the system calls that register for a signal handlers. Catches the signals and passes them appropriately to the client.
- Asynchronous signals are delivered b/w code blocks thus the shadow load and store are corrupted
- This is done so that the tool has control over the client program even in cases of longjump and native execution of the client program.
- Self modifying codes: Each time a block executes, a hash containing the original code from where the block was derived from is recomputed and checked, in case of a mismatch the block is discarded and the code is retranslated.
References

• http://www.eecs.berkeley.edu/~krste/papers/pin-wbia.pdf
• http://theory.stanford.edu/~sbansal/pubs/osdi08_html/index.html
• Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation
• http://en.wikipedia.org/wiki/Binary_translation
• http://en.wikipedia.org/wiki/Dynamic_recompilation