Address Stream Profiling

Charles Drews (csd305@nyu.edu, N11539474)

May 4, 2014

Project Phase 3: Final Report

Virtual Machines: Concepts and Applications, CSCI-GA.3033-015, Spring 2014

Problem Definition:

A program's performance on a virtual machine may be much worse than its native performance due to the additional overhead of emulation and binary translation. However, if the VM performs dynamic binary optimization, it may be possible to reduce this performance loss, or even provide better performance than the native environment if the source code was not particularly well optimized during compilation. In the latter situation, it might be beneficial to use an aggressively optimizing VM even when the source ISA and host ISA are the same. Therefore optimization is very important to VM performance.

In order to optimize, we must have some information about the code. One way to obtain such information is via static code analysis. Such static analysis can reveal static basic blocks, but cannot determine how often each block is executed or the likelihood of each path of a conditional branch being taken. This means that static analysis cannot reveal which blocks are heavily used and worth optimizing more, as opposed to lightly used blocks where the overhead of optimization may not be worthwhile. Similarly, static analysis cannot provide sufficient data to help decide how to improve code locality in translated code blocks, since the likelihood of each block transferring control to each other block is not known. The problem, therefore, is that static analysis does provide sufficient information to perform effective optimization.

Problem Importance:

The general solution to this problem is to use profiling to gather statistics about the program as it executes. The problem and this solution are important because of the many ways profiling data can be used. Profiling can reveal dynamic basic blocks, along with how often each dynamic basic block transfers control to each other dynamic basic block. From this we can infer a control flow graph, determine how heavily or lightly used each block is, and determine the likelihood of each path of a conditional branch being followed. Such information allows for a variety of optimizations to be performed.

Knowing how often each block is executed (the block profile), allows the VM to determine which blocks are worth aggressively optimizing and which blocks need only minimal optimization (or none). Knowing how often each block transfers control to each other block (the edge profile, from which the block profile can be inferred) allows for prediction of which path from a block-ending branch instruction will be followed during execution. Looking at the edge profile in aggregate can reveal the most common paths taken through all the blocks of code during execution. Knowing these paths allows for the formation of superblocks, consisting of a sequence of dynamic basic blocks that are likely to be executed together in a particular order. These superblocks then become good units for storing translated code in the VMs code cache, since when part of the translated superblock is needed, it is likely the entire superblock will be needed. This is therefore a means of improving the locality of the translated code stored in the code cache.

Challenges:

Solving this problem by collecting profiling data poses some small challenges and some more significant challenges. The first small challenge is to distinguish between instruction addresses and data addresses in a program's address stream, which is simple enough using a tool (e.g. Valgrind) that can label each address in the stream with its type. The next small challenge is identifying the dynamic basic blocks, but that should also be relatively simple based on whether consecutive instructions from the address stream are represented by contiguous addresses, taking into account the length of each instruction and the corresponding offset from one instruction's starting address to the next.

A more difficult challenge will be to recognize conditional branching instructions even when the branch is not taken and the control flow instead falls through to the next contiguous instruction. If at some point later in the address stream that branch is taken and it becomes clear that instruction is a jump, then instances of that same instruction from earlier in the stream that simply fell through must now also be updated and handled correctly as branches. A branch instruction must end a static or dynamic basic block, even if the jump is not taken and instead falls through. Therefore, relying only on whether two addresses are contiguous is not enough; I must also take into account information learned about an instruction from its other appearances in the stream.

Another challenge is distinguishing between conditional branching instructions and indirect jump instructions. Each will of course appear at the end of a dynamic basic block, but the conditional branch should be followed by one of a relatively small number of possible destination instructions, while an indirect jump (such as a return from a procedure call) might be followed by any one of a large number of possible destination instructions. Profiling a conditional branch is important since the likelihood of each of the branch's paths is key to forming super blocks. Profiling indirect jump may not be very useful for optimization purposes since none of the possible paths may appear very often. Also, the indirect jumps may significantly increase the number of edges appearing in the control flow graph; it may be helpful to determine a threshold for minimum number of edge traversals below which an edge may be excluded from exhibits displaying the graph.

One more challenge I see is determining an efficient way to present the profiling data. The number of blocks/nodes in the control flow graph might be very high for some programs, it will be important to store and present this data in a good way, whether for the purpose of providing it to optimization software, or providing it to a user for inspection.

Solutions:

My profiling tool implements the following solutions to the challenges described above. The first challenge is differentiating between instruction and data addresses. This is easily solved by obtaining the address stream using Valgrind's Lackey tool, then parsing each line of the stream and ignoring those that don't begin with "I" as the first character after the preceding newline. Next, each instruction must be categorized as one of the following: the beginning of a dynamic basic block (DBB), inside a DBB (neither first nor last instruction of a DBB), or the last instruction of a DBB, which must be a jump of some kind (call, return, conditional branch, unconditional branch, etc.). Also the same beginning/inside/end categorization for static basic blocks (SBBs) will need to be made.

The next challenge is to determine which instructions in the address stream represent jumps, since they are not tagged as such in the Valgrind/Lackey output. My approach here is to check whether or not the each instruction's address refers to a memory location that is contiguous with the location of the previous instruction. This means taking the previous instruction's location, adding the instruction length (provided in the Valgrind/Lackey output), and comparing to the current instruction's location. For example, if the previous address is 040016b0 with a length of 3 bytes, then if the current instruction address is anything other than 040016b3, I will know the previous instruction represented a jump, and the current instruction represents the start of a new DBB and SBB.

When this happens, and the current instruction is the start of a new block, I first check if I've already seen a block starting with that address or if it is new. Since each block (even DBBs) can be uniquely identified by

its first instruction, when I encounter a previously unseen block-starter I enter it into a hash table containing a struct for each SBB or DBB which counts how often it passes control to which other blocks. I also update the block struct with the block's last instruction once the next jump is encountered.

Conversely, if the previous and current instruction locations are indeed contiguous then I do not have evidence of a jump. However, I cannot with certainty say the previous instruction was not a conditional branch instruction; it's possible that it was, but the condition was not satisfied and rather than branch, control instead "fell through" to the next contiguous instruction. This means I need to look up each instruction in my hash table of block-starters; if I get a match then the previous instruction was a conditional branch that was not taken. There is one problem though: what if the previous instruction was indeed a conditional branch not taken, but this is the first time I'm seeing the current instruction, so it has not yet been identified as starting a new block?

There are two ways I could handle this challenge. First, after completing my scan of the address stream, I could scan it a second time. During the second pass, I'd have the full benefit of the hash table of blocks built during the first pass, and could identify conditional branches even if the branch is not taken on their first execution. For the sake of efficiency though, I chose not to make two passes through the stream. In an initial test of the Valgrind/Lackey tool, the address stream for a call to "ls -l" contained over 1.1 million entries, and that's after excluding the non-instruction addresses from the stream.

To avoid scanning the stream twice, whenever I encounter a block-ending instruction (a jump/branch that is taken), I check it against all previously identified blocks to see if it happens to fall in the middle of any other block. If it does, then that previously identified block contains a conditional branch that must not have been taken when that block was identified, and that block must now be split in two. This solution does of course involve back tracking (checking through each existing block every time a new block is found) but I believe it results in substantially less overhead than scanning through the whole address stream a second time.

A related challenge is also identifying all the branch/jump targets that must be the start of a SBB. Its possible that the first time one of these jump targets is encountered, it will be executed following the contiguously previous instruction, and there will be no indication it is a target (a.k.a. SBB leader). To handle this, whenever an SBB-starting jump target is identified, I look back at all the previously identified SBBs to see if it falls in the middle of one. If so, that SBB will be split in two, like I described above for splitting SBBs or DBBs based on newly identified branches.

Once each SBB and DBB-ending jump has been identified, the next challenge is to distinguish the conditional branches from the indirect jumps. The former are very important to the overall control flow graph (CFG) of the address stream; one of the primary reasons for creating the CFG in the first place is to see how often the conditional branches are taken and not taken. The indirect jumps, however, are not of as much interest, since they may jump to a large number of targets, none of which may be targeted with a high enough frequency to tell us anything useful about the program's control flow.

To make this necessary distinction, for each jump I consider whether control ever falls through to the next contiguous block, and to how many other non-contiguous blocks is control passed. This yields the following categorization of each block-ending instruction: NB for Not a Branch (control always falls through to the next contiguous block), UB for Unconditional Branch (control always passes to the same non-contiguous block), CB for Conditional Branch (control either falls through or passes to one non-contiguous block), or IJ for Indirect Jump (control may fall through or pass to multiple other blocks).

NB mostly occurs when an SBB ends with a non-jump instruction that immediately precedes a leader, or target for some other jump. However, an NB can also show up when profiling with DBBs, though the reason why is not obvious. Each DBB starts with a unique instruction, but multiple DBBs can end with the same jump/branch instruction. If, for example, instruction X ends both DBB 1 and DBB 2 then it is possible that every time X is executed at the end of DBB 1 it will fall through to the next contiguous block, while its executions at the end of DBB 2 sometimes yield a jump. Therefore, X is properly identified as a jump, but the profile for DBB 1 shows X as NB since it always falls through. In this manner, X actually has two sets of edge statistics in my profiler's output, one set for when it is executed at the end of DBB 1, and one

for when it is executed at the end of DBB 2. Fortunately, the other branch types (UB, CB, IJ) are more straight-forward, and as expected appear in both SBB profiling and DBB profiling.

All edges from each branch/jump are tracked, and my tool can produce a CFG will all edges included by passing a "-a" option (described in the user manual) when calling the tool. The default action, however, is to output the CFG ignoring the non-worthwhile edges. My approach is for jumps with only 2 (or 1) different paths, show both paths/edges even if one has a very low frequency of actually being taken. Then for jumps with >2 paths, the indirect jumps, shows all the paths/edges which have a frequency over a certain threshold (set at 10% based on my judgment of how much "clutter" is acceptable in a CFG), and do not display the paths/edges with frequency below the threshold.

The next challenge is determining a suitable output format. My tool produces two output files, one describing the profile in terms of SBBs, and one describing the profile in terms of DBBs. Both files utilize the same format (described in both the user manual and in the output files themselves) which shows a row of data for each basic block. Each row assigns a unique block ID, specifies the starting and ending instruction addresses of the block, the number of instructions in the block, and the number of times the block was executed. The row then describes the block's ending instruction in the terms described above (NB, UB, CB, or IJ) and finally provides a list of all the outgoing edges from the block. This edge list shows the block ID of each other block to which control was passed, and indicates the frequency with which control was passed. Also if one of these edges represents a fall-through to the next contiguous block, that edge is flagged with "(ft)". Therefore, this output combines a block profile (listing of blocks and how often each block was executed) with and edge profile (listing of all paths taken from each block-ending instruction and how often each path was taken) into one.

Analysis:

To test the accuracy of my profiling tool, I created a sample address stream in the same format as the Valgrind/Lackey output. I created this by hand so it would be quite short and easy to manually check, and also so that it would be sure to introduce the scenarios described above where blocks must be split based on information learned later in the scan of the stream. This sample input is in the attached file "example_trace.txt" (please see the attached user manual for an explanation of how to run my tool on this sample trace).

This stream contains the instruction with address 00000009 several times. The first time 00000009 is encountered it is contiguous with the prior and subsequent instructions, and there is no indication it is a jump. Therefore, after this first appearance 00000009 is not identified as either the start or end of a block, and is considered to be in the middle of a block. However, the next time instruction 00000009 appears it is followed by a non-contiguous instruction, indicating that 00000009 is a jump. This tests whether my profiler can correctly recognize that the prior block with 00000009 in the middle must be split into two blocks, which it does for both SBBs and DBBs. Please see the attached files "example_trace_SBB_profile.txt" and "example_trace_DBB_profile.txt" which show my tool's correct output based on this stream.

Similarly, Instruction 000000b3 appears twice. The first time it is contiguous with the prior and subsequent instructions, and is considered to be in the middle of a block. However, the second time it appears it is not contiguous with the prior instruction, meaning it is the target of a jump. This tests whether my profiler will correctly split apart the prior SBB with 000000b3 in the middle, while NOT splitting apart the prior DBB with 000000b3 in the middle. This is because all instructions must be unique in SBBs, so a jump target cannot appear in the middle of an SBB. However, only the first instruction needs to be unique with DBBs, so having 000000b3 in the middle of a DBB is no problem, even after we learn that 000000b3 is a jump target.

The output profiles based on "example_trace.txt" demonstrate that my tool can correctly process an address stream, and can successfully produce an SBB profile where each instruction only appears in a single SBB, and a DBB profile where an instruction may appear in multiple DBBs as long as each DBB has a unique starting instruction. I also ran my tool on several other input traces created by the Valgrind/Lackey tool. The output for these traces were more difficult to manually check for accuracy, simply because the traces themselves were so long.

For example the attached "test0.c" represents a simple "hello, world" program. The address stream produced by Valgrind/Lackey for this program includes over 100,000 addresses ("test0_trace.txt") even though the assembly code for this program is only 26 lines long ("test0.s" - created using "`gcc -S test0.c`"). Because the address stream was so long, the output of my tool ("test0_trace_SBB_profile.txt" and "text0_trace_DBB_profile.txt") identifies over a thousand basic blocks, which I was not able to manually match up with the assembly code which appears to only contain a couple of blocks.

I spot checked many of the blocks in my output by comparing the block and edge statistics against what I see manually in the address stream, but due to the sheer volume was not able to manually check them all. I created other address streams as well for other small programs, but every address stream Valgrind produced was similarly long, even when the corresponding assembly was very short.

However, based on my spot checks of those programs and my detailed checks of the profiles of "example_trace.txt" described above, I am confident that my tool is producing correct profiles based on the input address streams.