

LoadSpy Client Tool for DrCCTProf

Cassandra Chen, Apsara Fite, David Gantman, Mashaallah Moradi

University of California - Merced

CSE 165: Intro to Object Oriented Programming

Dr. Pengfei Su

Fall 2022

I. Introduction

Complex programs often have issues with efficiency, affecting performance and prolonging execution time. Inefficiencies in abstract programs are notably more difficult to detect and eliminate. Thus, while compilers can optimize code to a degree, it is better to modify the code at the implementation level to improve performance. DrCCTProf is a fine-grained call path profiling framework for analyzing metrics of programs at runtime. It monitors instructions by instrumenting binaries and provides diagnostic data for developers. It is built atop DynamoRio (DR), an interface for manipulating binaries at runtime. We present our client tool LoadSpy for the DrCCTProf library, designed to specifically identify redundant loading operations in target applications.

Motivation

Throughout our careers as software engineers we are constantly plagued with programs limited by memory and time complexity. If we can check for redundant variables in said programs, we will have a better understanding on why they are not as efficient as they should be. We decided to work on the Loadspy project because after building the program we have a tool that makes our job easier, instead of reading the program line by line to detect redundant variables we can check how redundant our program is by using loadspy to see the exact instructions that are being called redundantly, so that we know which part of our program needs optimization and which parts we can leave alone. Additionally if we use loadspy and there aren't any redundancies we will know that our program's performance is not being hindered by redundant variables and we can focus on other areas of our code to improve performance.

Redundant Loading

In C and C++, arrays are stored in memory as pointers to the element at its zeroth index. Subsequent elements of the array are stored at uniform offsets of the zeroth element's

address (indexed addressing). To illustrate, `*myArr` or `*(myArr + 0)` will return the zeroth element of array `myArr`. The line `*(myArr + 4)` will return the fourth element. Redundant loading occurs when an indexed address is loaded from memory more than once. While retrieving data from memory isn't particularly slow, loading from cache is even faster. Retrieving data from the stack happens significantly faster than retrieving data from memory.

Loading values in C/C++ means reading the value at a certain location so the CPU can perform some type of calculation. When a value is read from an array, the program must reach into memory to find the value. This is inherent to C/C++ because arrays in these languages are simply data elements of the same type stored in sequential memory. While reading from memory is fast, reading from something on the CPU is much faster. For example, if a value is stored on the stack, it will be kept on the CPU (in a register/cache) which is significantly faster to read from than reading all the way from memory. Named variables, also known as local variables, are stored on the stack where they can be retrieved significantly quicker than values stored in memory. A redundant load is a load in which subsequent reads from the same memory location yield the same value, thus making the additional reads redundant as there was no reason to fetch the value all the way from memory. To resolve this, if the value is stored on the stack using a named variable instead, the subsequent reads are much faster. The reason this isn't the default action with arrays is because they aren't necessarily small and can easily over-consume space in the cache. The best solution for efficiency is to store the relevant elements from the array in a local variable for the duration of the local scope's execution.

Design of LoadSpy

The goal of the LoadSpy tool in the DrCCTProf framework is to analyze programs specifically for redundant loading instructions. Understanding that redundant loading is

loading from an address more than once, LoadSpy must have a way to keep track of seen addresses. This is where shadow memory comes in.

Shadow memory is a broad computer science concept, but in the scope of memory redundancy profiling, shadow memory is used to track the most recent values held at certain addresses. Whenever an address is accessed for the first time, its shadow base address is added to shadow memory along with the value at that address. Whenever the address is read from again, LoadSpy checks what the value at that address was the last time it was accessed. This allows LoadSpy to determine if the same value was read from the same location multiple times.

Shadow memory is used in the existing client tool StoreSpy, which analyzes programs for redundant write operations. Redundant writes occur when the same value is written to an address more than once. In StoreSpy, each store/write instruction is instrumented to record the destination address and value to shadow memory before the instruction executes. Then, after the instruction executes, StoreSpy loads the shadow value of the address for comparison with the new value. If the value for the shadow address is equal, then it is a redundant write operation.

LoadSpy, however, does not need to compare values before and after an instruction is executed. It simply needs to check if the address has been accessed before. As LoadSpy goes through the instruction list, it will store each read address into shadow memory. But first, it will check to see if the address is already in shadow memory, meaning that the operation is a redundant read. LoadSpy will then log the redundant instruction's call path and byte data for diagnostics.

II. Implementation

To best explain our client tool, we will start from our main function, *dr_client_main*, in a top-down approach. The main function calls two other functions of importance:

ClientInit and *InstrumentInsCallBack*. *ClientInit* creates a log file to output call paths and memory data for redundant reads detected in the analysis. Then, *drcctlb_init* initializes the DR system with the callee function *InstrumentInsCallBack* and the parameter *CustomFilter*. *CustomFilter* sets DR to only include read instructions with the statement *instr_reads_memory*. This instruction information is referred to as *instrument_msg* within the *InstrumentInsCallBack* function. Summarily, the main function initializes a log file for diagnostic data and then passes only the read instructions of the target application to *InstrumentInsCallBack*.

If the source of an instruction is a memory address, that indicates that there is an object being read from. *InstrumentInsCallBack* iterates through the read instructions of the target application by using *instr_num_src* to get the number of sources in the instruction list. Each of the sources in the list are checked using *instr_get_src* and *opnd_is_memory_reference* to see if the source is an address. If the source is verified to be addresses, and thus is an object, it is stored into memory using *InstrumentMem*. After all source addresses have been stored, the core analysis function *InsertCleanCall* is finally called through *dr_insert_clean_call*.

In the DR library, *dr_insert_clean_call* is a function that inserts into the instruction list and stores the application state information on the DR stack, where it can be accessed from the callee function using *dr_get_current_drcontext*. We do so in our callee function *InsertCleanCall*, storing the state/context information as *drcontext* and the current thread information as the *per_thread_t* pointer *pt*. The list of objects being read for the current thread, *cur_buf_list* in *pt*, is then iterated through so that *InstrumentValueOfRead* can be called for each one.

The *InstrumentValueOfRead* function's primary purpose is to address the different data types being analyzed. The size of the current object being read, *resize*, and information

from *is_float* is used to determine the data type. For example, if *refSize* is 4 bytes, it means that the object is a float; 8 bytes corresponds to a double; 16 bytes is a float when *is_float* is true, and a double if *is_float* is false; and so on.

CheckNByteValueOfRead uses the information on data type and size from *InstrumentValueOfRead* to properly copy the address into shadow memory. First and foremost, it adds the current size of the object to the total read bytes (Line 243: *pt->bytesRead += AccessLen*). Then, it either finds or creates a shadow address for the object using *GetOrCreateShadowBaseAddress* and stores the result in *shadowAddr*. If the shadow address existed prior to this instruction, then *shadowAddr* will be an address. If this is the first access of this object, meaning there is no corresponding address in shadow memory, then *shadowAddr* will be some garbage value. The boolean *isRedundantRead* is true if *shadowAddr* is equal to *addr*, meaning that the object address already exists in shadow memory and, therefore, has been read before. Otherwise, it will be false since an address cannot be equal to a garbage value.

Once *isRedundantRead* has been determined, accounting for the different possible data types and sizes, we call *RedundantReadHandler*. This function records data on redundant operations. If *isRedundantRead* is true, then the context and *threadID*, or the call path, will be added to the redundancy table. Finally, the context is updated for the current thread.

There is also the function *RecordValueForLargeRead* for handling special cases where the size of the object is an integer larger than 8 bytes. It essentially does the same processes as *CheckNByteValueOfRead* and *RedundantReadHandler*.

III. Results

To test the LoadSpy client tool, we need to compare the diagnostic data from a redundant version and an optimized version of a program. The redundant program involved a for loop calling the same address multiple times to add a value to *sum*. It was designed to be

very redundant and easily detectable by LoadSpy. Below is the code of the first test and the result given by running LoadSpy.

```

1  const int length = 100000;
2  int myArray[length];
3  int sum;
4  void update() {
5      for (int i = 0; i < length; i++){
6          sum += myArray[i] / 11;
7          sum -= myArray[i] / 3;
8          sum += myArray[i] / 5;
9          sum += myArray[i] / 7;
10         sum -= myArray[i] / 9;
11         sum += myArray[i] / 13;
12         sum += myArray[i] / 17;
13         sum -= myArray[i] / 19;
14     }
15 }

```

```

1  ClientInit
2  ***** Dump Data from Thread 0 *****
3
4  Total read bytes = 12218903
5
6  Total redundant bytes = 7713309
7
8  Total redundant bytes ratio= 63.126035 %
9

```

LoadSpy detected that the unoptimized program had a redundant byte ratio of around 63.1%.

To optimize this program, we should reduce the number of times that it loads that address from memory. We can do so by loading the value at the address into a variable in cache. Using the variable, instead of calling the address every time we change *sum*, reduces the number of redundant loads.

Comparing the result found from the second test run, the change is significant. The redundancy of the optimized code is approximately 50.0%, which is 13.1% less than the unoptimized code. Below is the code of the optimized test and the result given by running LoadSpy.

```

1  const int length = 100000;
2  int myArray[length];
3  int sum;
4  void update() {
5      for (int i = 0; i < length; i++){
6          int scalar = myArray[i];
7          sum += scalar / 11;
8          sum -= scalar / 3;
9          sum += scalar / 5;
10         sum += scalar / 7;
11         sum -= scalar / 9;
12         sum += scalar / 13;
13         sum += scalar / 17;
14         sum -= scalar / 19;
15     }
16 }

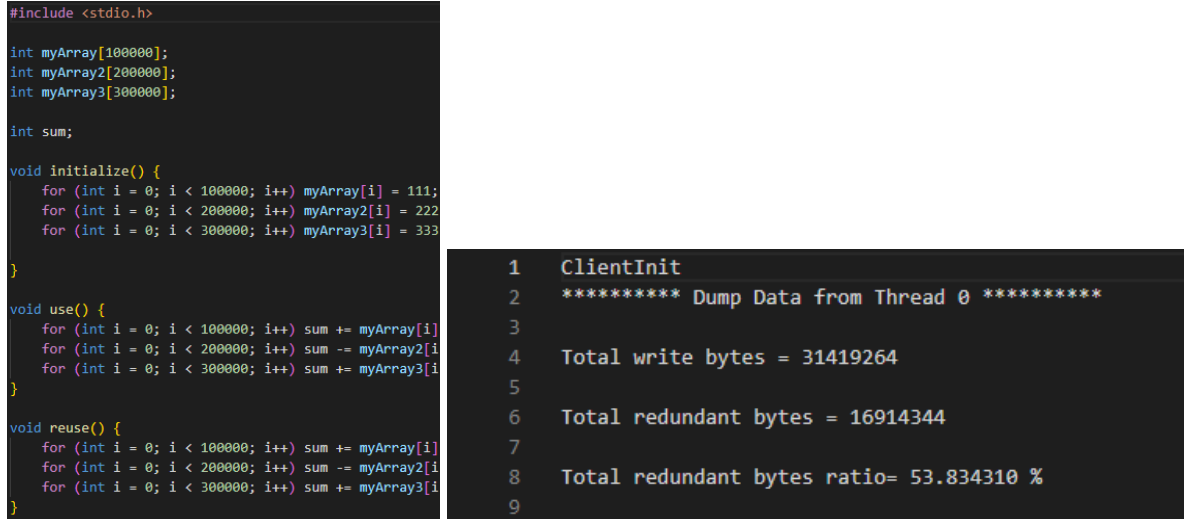
```

```

1  ClientInit
2  ***** Dump Data from Thread 0 *****
3
4  Total read bytes = 9818903
5
6  Total redundant bytes = 4913309
7
8  Total redundant bytes ratio= 50.039286 %
9

```

Finally, we verified that our client tool works as expected by using *redLoadInt* provided by our TA, Yuanzhou Yang. We successfully replicated the expected redundant byte ratio of 53.8%, verifying that our LoadSpy produces accurate results.



```
#include <stdio.h>

int myArray[100000];
int myArray2[200000];
int myArray3[300000];

int sum;

void initialize() {
    for (int i = 0; i < 100000; i++) myArray[i] = 111;
    for (int i = 0; i < 200000; i++) myArray2[i] = 222;
    for (int i = 0; i < 300000; i++) myArray3[i] = 333;
}

void use() {
    for (int i = 0; i < 100000; i++) sum += myArray[i];
    for (int i = 0; i < 200000; i++) sum -= myArray2[i];
    for (int i = 0; i < 300000; i++) sum += myArray3[i];
}

void reuse() {
    for (int i = 0; i < 100000; i++) sum += myArray[i];
    for (int i = 0; i < 200000; i++) sum -= myArray2[i];
    for (int i = 0; i < 300000; i++) sum += myArray3[i];
}

1 ClientInit
2 ***** Dump Data from Thread 0 *****
3
4 Total write bytes = 31419264
5
6 Total redundant bytes = 16914344
7
8 Total redundant bytes ratio= 53.834310 %
9
```

IV. Conclusion

The LoadSpy client tool detects loading redundancies within a program and gives us information regarding amount, and ratio of redundant bytes. Using the concept of shadow memory we access an instruction and its address will be added to the shadow address, then we access the next instruction and the address of the current instruction will be checked with the previous loaded shadow address. If two instructions are read from the same memory location we flag them as redundant reads and increment our redundant count by the amount of bytes each redundant instruction was. Each time we read a new instruction we compare it to all the previous shadow memory addresses(that were initially unique) if none of them matches we add the instruction into the shadow memory otherwise if the shadow address matches one of the previous addresses we label that instruction as redundant.

References

DrCCTProf Tutorial at CGO'22. XPerfLab. (2022, April 2). Retrieved November 27, 2022, from <https://www.xperflab.org/drcctprof/tutorial>

Liu, X. (2020). *DrCCTProf: A Fine-grained Profiler for ARM Eco-system*. Xu Liu's blog. Retrieved November 27, 2022, from <https://xl10.github.io/blog/drcctprof.html>

Liu, X. (2021). *DrCCTProf*. Read the Docs. Retrieved November 27, 2022, from <https://drcctprof.readthedocs.io/en/latest/>