CIE 6020 (Spring, 2018)

Report: Profile-Guided Source Coding in Dual-Compilation

Name: Chenhao Wu ID Number: 117010285

- This cover sheet must be signed and submitted along with the report on additional sheets
- By submitting this report with my name affixed above,
  - I acknowledge that I am aware of the University policy concerning academic misconduct (appended below),
  - I attest that the work I am submitting for this assignment is solely my own, and
  - I understand that suspiciously similar homework submitted by multiple individuals will be reported to School and Registry for investigation.
- Academic Misconduct in any form is in violation of CUHK(SZ)'s Student Disciplinary Regulations and will not be tolerated. This includes, but is not limited to: copying or sharing answers on tests or assignments, plagiarism, having someone else do your academic work or working with someone on homework when not permitted to do so by the instructor. Depending on the act, a student could receive an F grade on the test/assignment, F grade for the course, and could be suspended or expelled from the University.

Profile-Guided Source Coding in Dual-Compilation

Chenhao Wu

May, 2019

1 Introduction

Within this decade, the computing performance of mobiles and embedded systems have been significantly raised such that more computing workloads can be undertaken on mobiles and embedded systems, such as digital image processing, machine learning and deep learning applications, augmented-reality applications, and so on. In practice, one of the critical bottlenecks in mobiles and embedded systems design and optimization is the run-time memory space and power consumption. Since the complexity of program and data set grew much faster than the performance of memory capacity in the mobile and embedded system, rather than merely use expensive computing components to feed the needs of applications with high computing overhead, it is more promising to find ways to best utilize the system use of memory space. To realize this objective, we proposed a profile-guided source coding (PGSC), which will perform an adaptive compiler-guided data compression on memory space during run-time according to different applications and run-time environment.

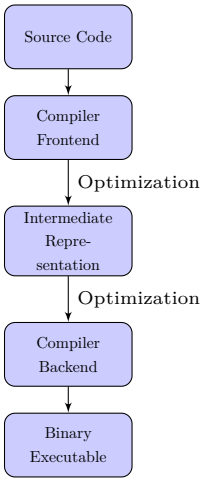

Figure 1: Traditional Compilation Procedure

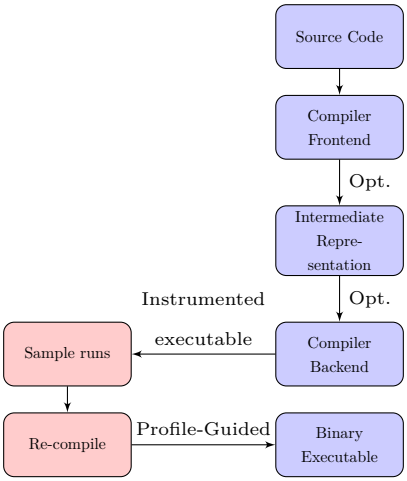

Figure 2: Compilation Procedure with PGO

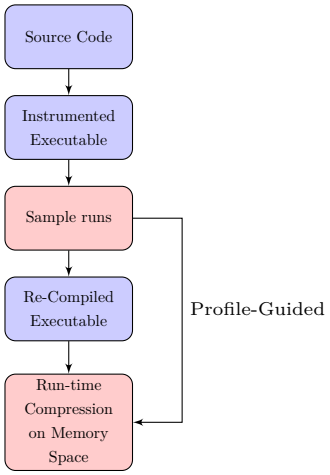

Figure 3: Profile-Guided Source Coding

The idea of this approach comes from a modern compilation technique, Profiled-Guided Optimization (PGO), that as shown in Figure 1 and Figure 2, differed from the traditional approach [1], the compiler

will perform a recompilation based on initial executable and sample runs. A profile, for instance, breaks the source code into blocks and devises a table of the frequency each block is executed via sample runs, and during recompilation, code blocks will be reordered with respect to the frequency table. For example, blocks with higher frequency will be allocated in memory area with lower memory address and blocks with lower frequency will be allocated in higher address so that the additional I/O overhead consumed in function callbacks will be reduced. Similar to previous compiler optimization methods, like the deletion of unreachable code, the constant folding optimization and propagation optimization, the objective of PGO is to further reduce the redundancy of unnecessary and/or unimportant logic branches in the source program.

The usage of this approach, however, can be extended into attempts in reducing the memory space, that after sample runs, not only the frequency of code blocks can be collected, but also the preliminary investigation of frequency that each block of memory is accessed can be gathered as well. As shown in Figure 3, the procedure of PGSC consists of a dual-stage compilation, that the compiler will initially compile the source code into an instrumented executable, and based on a profile generated after sample runs, the compiler will re-compile the source code such that during the run-time, the program will compress specific areas of memory space in order to reduce the redundancy of memory use, and will instantaneously decompress the compressed contents whenever the program needs.

In the tide that more and more applications on mobile and embedded systems behave to buffer long arrays of audio, images and video frames into the memory space during run-time, that in a low frequency the program access to them but they still need to be remained in the memory space, for usages like intermediate training matrices, sample sets of images and so on, PGSC can regularly compress them to reduce the buffer size. The compression is compiler-guided, thus it can be performed no matter which optimization has been applied onto the program and what third-party libraries the program has imported.

In this report we will briefly introduce the control flow of PGSC, the methodologies we applied to perform the compression over memory space, and the implementation of a compiler with PGSC.

2 Related Work

One solid work to use data compression for run-time memory system utilization had been practiced by Ozcan Ozturk, Mahmut Kandemir and Mary Jane Irwin [2], in which they designed a computational algorithms to compress the run-time data. Compiler-guided compression are also considered in compiler optimization related work [3][4][5]. In software engineering, a profile-directed feedback was firstly applied in IBM kernel compilation [6], and also adopted in the build of Firefox and Chromium [7][8]. Previous work using profile-guided compiler optimization had proved that this method of optimization is able to reduce the average memory occupancy from 15% to 50%.

3 Implementation

3.1 Structural RAM Assignment for Run-time Compression

In a run-time environment with the stack-allocation strategy, a consecutive memory area will be divided into segments, and allocated by activation records and run-time objects. The order of allocation is arbitrary,

and therefore in a while after the program being run, the allocated areas and free areas are interlaced. The system manages and looks up an allocation table to free up allocated space and reuse freed space.

As shown in Figure 5, an activity record consists of a sequence of run-time data and pointers towards the control & access links.

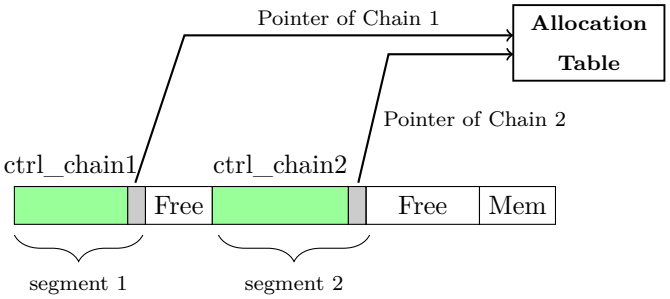

Figure 4: Memory Allocation

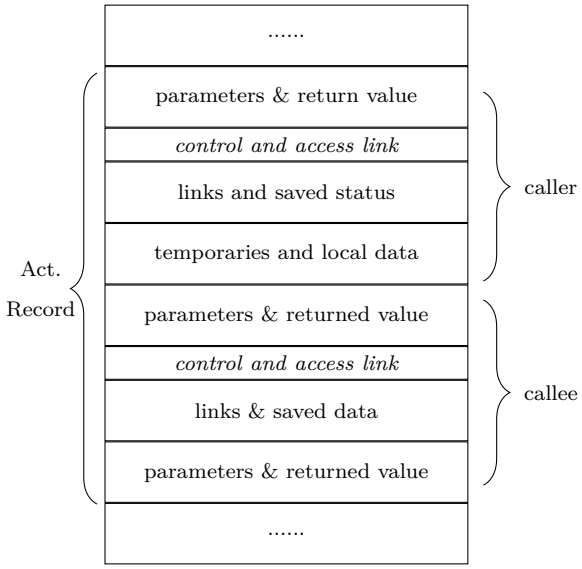

Figure 5: Task Stack

Our scheme to achieve data compression on RAM requires a sufficient analysis on run-time memory access records, and then we will separate the entire memory space into two elementary areas, run-time area and compressed area. Kindly note that the separation of memory space is not an actual separation on memory sequence, but applying two different tables to manage respective memory space.

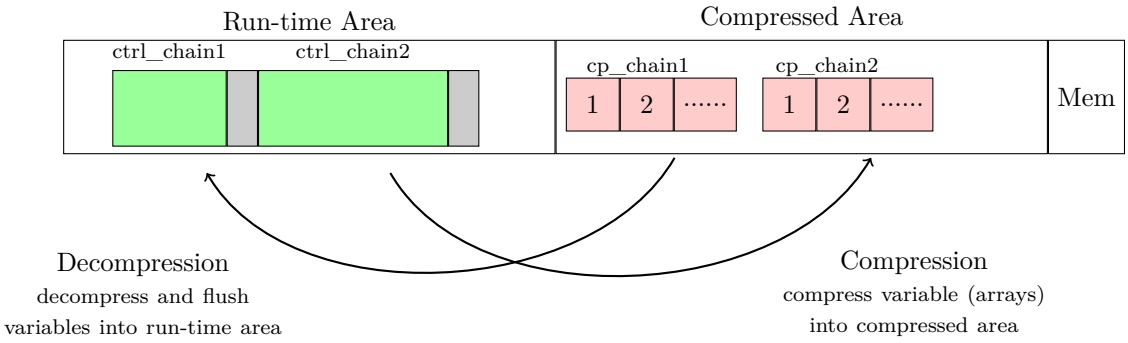

Figure 6: Run-time Compression & Decompression

In the compilation, the compiler specifies which areas of data need to be compressed during idle time, and also generate codeword of each segment, so as during run-time the program would automatically compress them from the run-time area into the compressed area. Whenever the compressed data are needed or being called, the program will instantaneously decompress the data from the compressed area, and flush them into the run-time buffer. The configuration of the compression is guided by the profile generated by sample runs,

and also an initial compiling parameter *-DEG_COMP*, which represents the degree of compression. For instance, if the constraint of the system memory is not strict, the degree of compression can then be tuned down in order to reduce the compression area and increase computing efficiency.

3.2 Implementation of Run-time Compression Compiler

In this project, we applied a program profiler *valgrind* and a widely used compiler framework *LLVM* to implement the run-time compression compiler. For the first tool we used in the project, *valgrind*, it has several internal tools to examine and track the record of allocation/free instructions called by the program on heap during run-time, including the information of how much heap memory the programs used and statistics of how much time the program access to each segment of allocated space. Also, it can also generate readable profiles after analysis, describing the behavior of the program and passing to consecutive process.

A sample profile used in our project is like following

Listing 1: (Sample Profile)

```
$ cat ./sample_prof
desc: --time-unit=B --detailed-freq=1
cmd: ./sample1
time_unit: B
#-----------
snapshot=0
#-----------
time=0
mem_heap_B=0
mem_heap_extra_B=0
mem_stacks_B=0
heap_tree=detailed
n0: 0 (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
#-----------
snapshot=1
#-----------
time=1016
mem_heap_B=1000
mem_heap_extra_B=16
mem_stacks_B=0
heap_tree=detailed
n1: 1000 (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
n0: 1000 0x108730: main (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
...
#-----------
snapshot=24
#-----------
time=30344
mem_heap_B=10000
mem_heap_extra_B=24
mem_stacks_B=0
heap_tree=detailed
n3: 10000 (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
n2: 8000 0x1086FC: g (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
```

```
n1: 4000 0x108712: f (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
n0: 4000 0x10874C: main (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
n0: 4000 0x108751: main (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
n1: 2000 0x10870D: f (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
n0: 2000 0x10874C: main (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_massif/sample1)
n0: 0 in 1 place, below massif's threshold (1.00%)

==8260== ======== SUMMARY STATISTICS ========
==8260==
==8260== guest_insns:  2,268,254
==8260==
==8260== max_live:     73,104 in 2 blocks
==8260==
==8260== tot_alloc:    73,104 in 2 blocks
==8260==
==8260== insns per allocated byte: 31
==8260==
==8260==
==8260== ======== ORDERED BY decreasing "max-bytes-live": top 10 allocators ========
==8260==
==8260== -------------------- 1 of 10 --------------------
==8260== max-live:    72,704 in 1 blocks
==8260== tot-alloc:   72,704 in 1 blocks (avg size 72704.00)
==8260== deaths:      1, at avg age 118,908 (5.24% of prog lifetime)
==8260== acc-ratios:  0.00 rd, 0.00 wr  (0 b-read, 16 b-written)
==8260==    at 0x4C29F9F: malloc (vg_replace_malloc.c:299)
==8260==    by 0x4EBA9BF: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.22)
==8260==    by 0x400F799: call_init.part.0 (dl-init.c:72)
==8260==    by 0x400F8AA: _dl_init (dl-init.c:30)
==8260==    by 0x4000C59: ??? (in /lib/x86_64-linux-gnu/ld-2.24.so)
==8260==
==8260== -------------------- 2 of 10 --------------------
==8260== max-live:    400 in 1 blocks
==8260== tot-alloc:   400 in 1 blocks (avg size 400.00)
==8260== deaths:      none (none of these blocks were freed)
==8260== acc-ratios:  0.50 rd, 1.54 wr  (200 b-read, 616 b-written)
==8260==    at 0x4C2AD2F: operator new[](unsigned long) (vg_replace_malloc.c:423)
==8260==    by 0x108884: main (in /home/vitowu/Github/CIE6020-CourseProj/test/valgrind_dhat/sample2/prog)
==8260==
==8260== Aggregated access counts by offset:
==8260==
==8260== [   0]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [  16]  4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
==8260== [  32]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [  48]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [  64]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [  80]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [  96]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [ 112]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [ 128]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [ 144]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [ 160]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
==8260== [ 176]  3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
==8260== [ 192]  3 3 3 3 3 3 3 3 1 1 1 1 1 1 1 1
==8260== [ 208]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
==8260== [ 224]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
==8260==
==8260== ================================================================
```

from which we can obtain the analysis of allocate/free and read/write instructions on RAM.

The second tool, *LLVM*, stands for a flexible compiler framework with great extensibility, allowing researchers in the area of compiler optimization to easily load their own customized optimization modules into compilation process and control the ordering of compilation. In this project, we implement three *LLVM opt* module, which are ***ProfileManager.so***, the agent to parse profiles; ***Compression.so***, the module to emit compression/decompression functions into original program; and ***LZO.so***, a library to provide compression and decompression methods to the optimizer.

The primary compression method we applied in the project is Lempel–Ziv–Oberhumer compression, During the second compilation, *LLVM* will substitute part of reading/writing instructions with function calls of LZO compressing/decompressing according to the profile. Eventually, for writing instructions of some data structure being allocated in run-time, the finalized program does not directly write the raw data onto the RAM, and instead, the program compress them and write the compressed data into compressed area. And also, the reading instructions are being substituted by decompression functions in a similar approach.

3.3 Work Flow

An simplified work-flow of Run-time Compression Compiler is as following:

1. *LLVM* guides clang/clang++ (generic raw compilation) to initially compile the source code into binary
2. Insert *valgrind* into the binary program and run for an arbitrary long period, profiling the program on heap usage
3. Passing the profile to *LLVM* and substitute read/write instructions according to profile

4 Experiments

4.1 Testing Programs

To examine the performance of run-time compiler, we apply several testing programs to make comparison between ordinary compiler and compiler with profiled-guided source coding.

Testing program	Functionality
LU-Decomposition	Solve a set of linear equations using LU decomposition
Facial Recognition	Capture and learn new faces from camera
LZ Compression	Tool to compress literature text using LZO
Huffman Coding	Tool to compress literature text using Huffman coding on word

4.2 Testing Environment

- **Operating System**: Debian 9 Virtual Machine
- **Architecture**: x86-64
- **RAM**: 512 MB
- **Number of Processors**: 1
- **Clock Frequency** 2.8 GHz
- **Generic Compiler**: clang/clang++

The number of processors and RAM size were intended to be limited in order to simulate a similar environment as on mobile phones and embedded systems, in which CPU and RAM overhead are much more significant than personal computers.

5 Result

In the experiment section, we compiled the testing programs by two different approaches. The first approach is using *clang* to compile the source code without adding any optimization flag. The second approach is using Run-time Compression Compiler to compile the testing programs, which is implemented in this project. We analyzed the maximum memory occupancy, the average memory occupancy, and extra time in compilation respectively, and summarize to a comparison between this two approaches as following

Testing program	MMO(clang)	AMO(clang)	MMO(RCC)	AMO(RCC)	Extra Time(RCC)
LU-Decomposition	214.3 Mb	165.7 Mb	153.7 Mb	53.9 Mb	+47%
Facial Recognition	470.0 Mb	365.2 Mb	190.7 Mb	80.3 Mb	+239%
LZ Compression	253.7 Mb	103.8 Mb	217.8 Mb	93.7 Mb	+65%
Huffman Coding	324.1 Mb	214.7 Mb	293.7 Mb	196.5 Mb	+53%

where MMO stands for Maximum Memory Occupancy and AMO stands for Average Memory Occupancy.

From the result of experiments, we can find that by using run-time compression compiler, both the maximum memory occupancy and the average memory occupancy decreased. For LU-Decomposition and Facial Recognition programs, the decrease rate is than larger than the rest two programs, since during run-time, the former two programs would store lots of intermediate matrix and intermediate pictures array, which has much more redundancy than the latter two programs and then contribute to a higher compression rate.

Also, even though the extra compression time increased after applied run-time compression compiler, it is still within an acceptable range. Hence, for the most of applications in ordinary scenarios, applying run-time compression on them is useful and considerable.

6 Conclusion and Future Work

In this report, we introduced our approach, as a compiler optimization, to delimit the memory occupancy. In our approach, we propose to use a profile-guided strategy and to apply source coding on RAM during run-time. Also, we implemented a run-time compression compiler and designed experiments to test and validate our approach. From the result we can conclude that, our approach is feasible in utilizing the memory occupancy with a considerable optimizing rate. One shortcoming occurred in the experiment is that the programs using run-time compression run slower than ordinary programs, since CPU has to distribute some computing resources to compress and decompress the data instantly. We believe that this problem is significant due to our approach is left in software-level optimization, and it would be reduced or even eliminated by using hardware-based compression. In the future, we would like to apply the technique illustrated in the report on hardware and compare the performance of them.

References

[1] J.D. Ullman A. Aho R. Sethi. *Compiler: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Mary Jan Irwin Ozcan Ozturk Mahmut Kandemir. "Using Data Compression for Increasing Memory System Utilization". In: (2009), pp. 901–914.

[3] M S Ali. "Compiled Code Compression for Embedded Systems using Evolutionary Computing". In: (2008), pp. 1173–1174.

[4] Takeshi Ohkawa Takashi Yokota Kohta Shigenobu Kanemitsu Ootsu. "A Translation Method of ARM Machine Code to LLVM-IR for Binary Code Parallelization and Optimization". In: (2017), pp. 575–579.

[5] Sharmila Chidaravalli Manjunath T.K. "Static Analysis of Designing A Compiler Tool That Generates Machine Codes for The Predicted Execution Path: A Bypass Compiler". In: (2011), pp. 418–422.

[6] *IBM Knowledge Center Profile-directed Feedback*. `https://www.ibm.com/support/knowledgecenter/SSGH4D_13.1.0/com.ibm.xlf131.aix.doc/proguide/optimizepdf.html`. Accessed: 2019-03-15.

[7] *Mozilla Building with Profile-Guided Optimization*. `https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Building_with_Profile-Guided_Optimization`. Accessed: 2019-03-15.

[8] *Chromium Making Chrome on Windows Faster with PGO*. `https://blog.chromium.org/2016/10/making-chrome-on-windows-faster-with-pgo.html`. Accessed: 2019-03-15.

[9] C. S. Wilson S. P. Amarasinghe J. M. Anderson S. W. K. Tjiang S.-W. Liao C.-W. Tseng M. W. Hall M. S. Lam J. L. Hennessy R. P. Wilson R. S. French. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers". In: (1994), pp. 31–37.