LABORATORY TUTORIAL #4

PROFILE-GUIDED OPTIMIZATIONS, VALUE PROFILING, AND CODE SPECIALIZATION

V1.0, Nov. 2023

Course: "Efficient Heterogeneous Computing"
Master Program in Informatics and Computing Engineering (M.EIC)
Faculty of Engineering of the University of Porto (FEUP), Porto, Portugal

GOALS:

- Acquire acquaintance with profile-guided optimizations (PGO), code specialization (including partial evaluation), and value profiling

REQUIRES:

- Experiments in a desktop or laptop with C programs

- Execution time measurements

PLANNED TIME:

1:30

EXPERIMENTS

PROFILING GUIDED OPTIMIZATION (PGO)

Profile Guided Optimizations (PGO) [6][5][7][8] refers to the application of optimizations taking advantage of profiling information.

Let's use the PGO support of *gcc* to apply PGO. Herein we use two options to control optimizations [11], *-fprofile-use* and *-fprofile-generate*:

1. Compile the code generating the binaries to collect profile information:
   `gcc -fprofile-generate=<profile_dir> progname.c`
   or
   `gcc -fprofile-generate progname.c`
2. Execute the program binary generated in 1. with the input data (data shall be representative of the most real-life scenarios)
3. Compile the program with the PGO option to consider the profiling data collected in execution 2.:
   `gcc -fprofile-use=<profile_dir> progname.c`
   or
   `gcc -fprofile-use progname.c`
4. Execute the new program binary and verify if the execution time has improved.

(1) Could you provide two code examples to which you think that PGO can have a noticeable impact in terms of execution time? Justify.
(2) Compile the project code with PGO options and verify the impact on execution time.

CODE SPECIALIZATION AND PARTIAL EVALUATION

Code/program specialization is an important optimization that can significantly impact performance and/or power dissipation and energy consumption, especially when the target architecture provides resources to naturally implement certain specializations and execution models that may benefit from such specializations. In the context of compiler optimizations, Partial Evaluation[1] (see, e.g., [4]) can be seen as a particular case of code/program specialization and sometimes refers to the automatic application of optimizations when program variables have fixed values at runtime.

1. Analyze the project code and identify possible code specialization and partial evaluation opportunities.
2. Select the opportunities you think may have more impact in terms of execution time and apply them to the project code.
3. Check the impact by measuring the execution time before and after applying code specialization to the project.

VALUE PROFILING

Value profiling [1][2] is a profiling technique that, at the program level, is focused on acquiring information regarding the values stored in variables. This information can then be efficient in applying code/program specialization [3] or even deciding about multiversioning[2].

1. Figure 1 illustrates a simple example using the MVP library [13] to monitor the values of program variables. Based on the values used in the program, the MVP output shows two specific values for the exponent used in the *pow* function. The values collected might result in the use of two possible specialized versions. Present a new version of this simple example considering code specialization.
2. Analyze the project code and use the MVP library to collect values of selected variables you think are important as code specialization opportunities.
3. Modify the code of the program using code specialization if you identify more opportunities than the ones previously identified. Compare the execution times of the possible project versions you may have.

[1] From [12]: "Partial evaluation, which is also known as "currying" [15], is a technique where the computation is evaluated with respect to a given operand's value. That is, rather than treating an operand as a variable, its value becomes "fixed." Given a specific value of an argument, the code is specialized by replacing the occurrences of the variable with the corresponding value."
[2] I.e., the generation and inclusion of multiple variants of a region of code.

```c
#include <stdio.h>
#include <math.h>

#define N 10

int a[N] =
   {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int b[N] =
   {2, 2, 2, 3, 3, 3, 3, 2, 2, 2};
int c[N];

int main() {

  …

  for(int i=0; i<N; i++) {
    c[i] = pow(a[i], b[i]);
  }

  …
}
```

(a)

```c
#include <stdio.h>
#include <math.h>

#include "monitor_v2/valuecountermonitor.h"

#define N 10

int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int b[N] = {2, 2, 2, 3, 3, 3, 3, 2, 2, 2};
int c[N];

int main() {
    VCM* vcm_a = vcm_init("test VCM a", 100, 0, LFU, 0, 200, 1);
    VCM* vcm_b = vcm_init("test VCM b", 100, 0, LFU, 0, 200, 1);
    VCM* vcm_c = vcm_init("test VCM c", 100, 0, LFU, 0, 200, 1);

    for(int i=0; i<N; i++) {
        vcm_inc(vcm_a, a[i]);
        vcm_inc(vcm_b, b[i]);
        c[i] = pow(a[i], b[i]);
        vcm_inc(vcm_c, c[i]);
    }

    vcm_print(vcm_a);
    vcm_print(vcm_b);
    vcm_print(vcm_c);

    vcm_a = vcm_destroy(vcm_a);
    vcm_b = vcm_destroy(vcm_b);
    vcm_c = vcm_destroy(vcm_c);
}
```

(b)

Figure 1. Code example: (a) original code; (b) code modified for monitoring the values for each variable using the VMP library [13].

REFERENCES

[1] B. Calder, P. Feller and A. Eustace, "Value profiling," Proceedings of 30th Annual International Symposium on Microarchitecture, Research Triangle Park, NC, USA, 1997, pp. 259-269, https://doi.org/10.1109/MICRO.1997.645816.

[2] Brad Calder, Peter Feller, Alan Eustace, "Value Profiling and Optimization," Journal Instr. Level Parallelism, 1999, 1-6. https://jilp.org/vol1/v1paper2.pdf

[3] Eui-Young Chung, B. Luca, G. DeMicheli, G. Luculli and M. Carilli, "Value-sensitive automatic code specialization for embedded software," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, no. 9, pp. 1051-1067, Sept. 2002, https://doi.org/10.1109/TCAD.2002.801096.

[4] Neil D. Jones. 1996. An introduction to partial evaluation. ACM Comput. Surv. 28, 3 (Sept. 1996), 480–503. https://doi.org/10.1145/243439.243447

[5] Profile Guided Optimization (PGO), https://en.wikipedia.org/wiki/Profile-guided_optimization

[6] Carleton, Gary, Knud Kirkegaard, and David Sehr. "Profile-guided optimizations." Dr. Dobb's journal 23.5 (1998). https://www.drdobbs.com/profile-guided-optimizations/184410561

[7] Nitin Kumar, "Profile-guided optimization (PGO) using GCC on IBM AIX: Applying PGO to Facebook zstd," Published January 17, 2019. https://developer.ibm.com/articles/gcc-profile-guided-optimization-to-accelerate-aix-applications/

[8] "Profile-Guided Optimization," Intel® C++ Compiler Classic Developer Guide and Reference, 7/13/2023. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/profile-guided-optimization.html

[9] GNU gcc, “Auto-vectorization in GCC,” https://gcc.gnu.org/projects/tree-ssa/vectorization.html
[10] GNU gcc, “3.1 Option Summary,” https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
[11] GNU gcc “3.11 Options That Control Optimization,” https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[12] João M.P. Cardoso, José Gabriel F. Coutinho and Pedro C. Diniz, Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, Morgan Kauphman, 2017. ISBN: 978-0-12-804189-5 . https://doi.org/10.1016/C2015-0-00283-0. Online resources: https://web.fe.up.pt/~jmpc/books/hpec/

TOOLS AND LIBRARIES TO BE USED

[13] Vitória Correia, and Pedro Pinto, “VMP: Value Counter Monitor,” version 2, FEUP, SPeCS Research Group, June 2023.