WCET Analysis-Report

M2-HECS-Lorenzo Sinitò

Brief Introduction

This TP is divided into two sections: The use of Otawa (for the estimation of WCET), and the implementation of a tool which can reconstruct the CFG of a program, starting from its disassembled code. In this case, the script has been implemented in python, and it is stored in the folder.

Part1

2.2.1 – CFG Reconstruction with Otawa

The CFG is reconstructed starting from the elf file, which is the product of the C code compilation. Depending on the specified argument, in the command line, the optimization process of the compilation is affected.

The optimization level could, without affecting the program intended behavior, change its structure. Seldom, the higher the level, the deeper the impact. The final effect can be better appreciated analyzing the generated CFG. Indeed, it could happen, to find out differences between the expected CFG and the binary-level CFG obtained. In the observed case, *bsort_return* function, the improvement can be highlighted in the use of less memory (load & store) operations.

By opening the provided CFGs, optimizations are appreciated in the reduced complexity and in the reduced number of estimated WCET. With -O1, it is reduced from 4782 to 1809. It is also worth noticing that, not always, by further increasing the optimization level, an actual improvement is encountered. In fact, with -02, the result is the same as -01.

2.2.2 – Micro-architectural Analysis

According to the content of each block, the execution time can change. It depends, in fact, from the type of instruction, from the size, from possible hazards, and from the internal architectural path.

The execution time can also vary according to the chain of instructions to be executed. For example, in case of hazard, the pipeline is stalled so that them are avoided. They are frequent when the same registers are involved in the operation chain. Seldom, it is also aggravated by memory accesses, which slow down the performance (architectural path, memory r/w delay, hit/miss with caches).

2.2.3 – Loop-bound Analysis

The structure of the .ffx file is hierarchical. It means that every layer is delimited by an identifier, which specifies its function. Every layer is opened and closed and follows the behavior of the program. However, it doesn't clearly show the possible transitions between one block to another.

Regarding the two loops, two values are indicated. One, *maxcount*, which indicates the repetition of the single loop itself. The other, *totalcount*, keeps in consideration the number of times the instruction is executed in the program. In fact, both *maxcount* is 99 (max iteration of the loop), but *totalcount* changes. For the inner loop, it is evaluated as 99x99=9801.

2.2.4 – Path Analysis

The indicated variables are related to the nodes, the edges, and the corresponding number of times. Nodes are defined as integers, while edges are modelled as sums (positive and negatives according to the verse, if entering or exiting). In this file, products are implicit. Moreover, variables can be subjected to constraints over the iteration number.

By solving the system constraints, iteration numbers are retrieved. By inserting an edge which cuts the CFG and which gives a direct path to the end, new values are obtained. In this case, of course, the iterations are drastically reduced. Files are given in the provided archive.

Part2 – CFG Reconstruction Tool

This tool is implemented in python. It reads from the file "*out.txt*", which is generated through the command

arm-eabi-none-objdump -d f.elf

"f.elf" is generated starting from "*f.c*". To use the tool for other programs, just change the input name (in this case "*out.txt*") and the function to be searched (in this case "*<find_min>:*"). As it is provided, it is configured to work with the function I have created. However, to verify its correctness, in the "*bsort*" folder, the reconstructed CFG generated with this tool can be appreciated and compared with the one generated with Otawa.

The "*file.c*" contains the code related to a simple main which recalls "*find_min()*". This function is made of a loop and an if, so that the branches behavior can be better appreciated. The script, searches for "*<find_min>:*" in the disassembled file and reads all its instructions, which are a priori stored, to understand the instruction set.

Each block terminates with a branch instruction, which in almost all cases points to the link register, which stores the return address of the calling function.

A convenient data structure, to better generate the CFG, is a graph. For this purpose, the "*NetworkX*" python module is exploited, which provides a useful set of methods to easily declare graphs and manipulate them. The parsing procedure is done as follow:

- The dissembled code is read line per line. This should be the most convenient solution, as it allows to easily distinguish the instructions, and to better comprehend the direction flow, as branch instructions can be explicitly read.
- What is necessary, to implement the parser, is:
    - Understanding how the information is provided in the disassembly format. Thanks to the use of regular expressions (imported with the "*re*" python library), each line is split. The interested columns are the one storing the memory address, the instruction, and the argument of the instruction (only for the branches, as the target is displayed).
    - The understanding of the entry point: Reckoning on the fact that the instructions are stored in memory in order, the first instruction of the block is for sure the entry point.
    - The identification of branches and of their type, as they can be conditional or unconditional. ***Considering a reduced instruction set of ARM7***, the unconditional branches are identified with the instructions *"b","bx","bxl"*, and no other instruction

starts with "*b*" but branches. In this case, only one edge is considered and properly labeled. In the other case (conditional), two edges are formed, to identify the two possible paths (taken or untaken). Furthermore, the case where the branch could have as target one internal register is not supported.

  - The identification of the target instruction (same block or outer block membership). Due to the exploding recursive complexity, if a branch points to an instruction which is outside the portion of considered code, then its CFG is not displayed, except for the node of the function itself. Simply, this node will be added and will return to the next instruction. Of course, this assumes that the called instruction is re-entrant.
  - The identification of the exit point: the branch to the *link register* is researched. If it is not present, then the last instruction is considered to be the ending one. Anyway, only the case where at most one branch to link register is considered.

The algorithm works as follows:

- First lecture, to store the function instruction lines, and for the parsing. After this procedure, all the instruction set and of the block, and the relative addresses, are known.
- Iteration through the parsed lines:
  - A node is added for every *i* instruction address as there are no repetitions and as it simplifies the research in the graph. It is also labelled with its specific instruction (will be useful for the CFG representation).
  - The *i* instruction is analyzed. Normally, if not the case of a branch, the next instruction, and so the next node, is the following line, assuming that the set isn't already ended. If this is the case, then a simple, not labelled, edge is added. In the case of a branch, it is analyzed. According to its typology and to its target address, a specific label is added to the next edge.
- At the end of the second iteration, the direct graph is formed, and it can be used for the graphical reconstruction. To complete this task, the dot format is preferred. The CFG is written into CFG.dot, that is then converted through the command "*dot -Tps CFG.dot -o CFG.pdf*", which produces the PDF file. This command is recalled using the sys python module.

Please, note that, even if the contiguous instructions should be represented as a unique block, in my representation, due to the verbose label description, there is no risk of misunderstanding. Simply, a null edge label, means that there is no other possible path. On the contrary, in case of branches, the typology is always explained.

Moreover, *NetworkX* works so that, when adding an edge, the two nodes which form it are added if missing from the graph. Anyway, for how the problem is formulated, when adding again the node with the label information, it is updated. There is no risk of information loss. Further info can be found on https://networkx.github.io/documentation/stable/