Code Optimization and Transformation Course Project on:

List Scheduling

The Code Optimization and Transformation course exam is composed by two parts. One is an oral test, the other is an homework, to be terminated before course last call. To pass the whole exam, you must get a pass grade from both the test and the homework. The homework must be taken in pairs.

During the lab classes, the LLVM [7] compiler has been introduced. The homework must use the 3.0 release version of LLVM. A sample project – COT passes [8] – is available on GitHub [4]. It must be used as a starting point for the homework. LLVM testing framework [3] must be used to validate the implementation.

Sources must versioned using Git [9]. A good tutorial can be found here [2]. Sources must be published on GitHub [4].

Assignment

The goal of the project is to implement *List Scheduling* of basic-blocks. The algorithm is described in [1].

Your implementation must work on the LLVM Intermediate Representation. Since the original algorithm works on a representation that is very close to machine code, the following restriction must be considered:

Machine Info: the list scheduling algorithm requires knowing the latencies of each instruction and the number of functional units of the target machine. LLVM IR is not so close to the target machine, so these information are missing. You are free to define such parameters, using a MIPS [5] architecture as starting point – e.g MIPS R10000 [6].

Advices

Machine information can be defined in a configuration file. The LLVM framework provides the llvm::MemoryBuffer class to read a whole file in memory. It can be used to load the configuration file and parse it like a string.

You can implement an analysis pass to compute these information – e.g. loading them from the configuration file. Once loaded, they never changes. Such kind of analysis are special. They are called *Immutable*. The llvm::ImmutablePass must be used as the superclass for that kind of passes.

The list scheduling is obviously an optimization pass. It can be implemented using a llvm::BasicBlockPass. It requires knowing information about dependencies between instructions in a basic-block – e.g. *Read-After-Write*, *Write-After-Write*, ... Algorithm reported in [1] uses a *Data-dependency Graph* to represents such dependencies. You are free to chose whether implement an analysis pass building the DDG of a basic-block, or check dependencies exploiting available LLVM passes – e.g. dominance information, def-use chains, memory dependence information, ...

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*, chapter Instruction Level Parallelism. Pearson.

[2] Scott Chacon. Pro Git. URL `http://git-scm.com/book`.

[3] John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner. LLVM Testing Infrastructure Guide. URL `http://llvm.org/releases/3.0/docs/TestingGuide.html`.

[4] GitHub Inc. GitHub. URL `http://github.com`.

[5] MIPS Inc. MIPS Technologies, . URL `http://www.mips.com`.

[6] MIPS Inc. MIPS R10000 Microprocessor User's Manual, . URL `http://techpubs.sgi.com/library/manuals/2000/007-2490-001/pdf/007-2490-001.pdf`.

[7] University of Illinois at Urbana-Champaign. Low Level Virtual Machine. URL `http://www.llvm.org`.

[8] Ettore Speziale. Compiler Optimization and Transformation Passes. URL `https://github.com/speziale-ettore/COTPasses`.

[9] Linus Torvalds. Git. URL `http://git-scm.com`.