Path Profiling in LLVM

Adam Preuss*
Dept. of Computing Sciences
University of Alberta
apreuss@ualberta.ca

August 22, 2010

1 Introduction

The objective of this submission is to incorporate an efficient implementation of *path profiling* (PP) into the LLVM compiler. PP is a method of *feedback directed optimization* (FDO) that monitors the execution sequence of a function's basic blocks with respect to a *control flow graph* (CFG). A compiler optimizer can perform better transformations with the benefit of path execution frequency.

PP incurs a performance overhead because the use of profiling information is a multi–step process. First, an instrumentation pass must analyze the CFG of a program, associating numbers with potential paths. It must then insert additional instructions to monitor each possible path. The program must be executed to gather the frequency of each path and, finally, the original code must go through an optimization pass based on the information gathered by the PP instructions.

Currently, LLVM contains instrumentation passes for *block profiling* (BP) and *edge profiling* (EP), as well as an interface that allows the optimization passes to access the profiling information. The PP instrumentation pass builds on some of the existing profiling utilities. However, to avoid the additional overhead of unnecessary computational methods in an existing interface, the new path profiler defines a new interface for FDO passes. This interface accesses the accumulated PP data. *Path numbering* (PN) is based on the algorithm proposed by Ball and Larus [1], which transforms a function's CFG into a *directed acyclic graph* (DAG) and assigns each path a unique number.

The PP for LLVM was developed under the supervision of Professor José Nelson Amaral and under the advisement of Paul Berube. Slobodan Pejic began work on this project in the summer of 2009, designing a layout and writing the PP instrumentation pass. Adam Preuss has completed and tested the project, making modifications to the existing instrumentation pass and reimplementing the path-profile interface.

2 User Information

Each aspect of PP is made available through LLVM's optimization tool `opt`. Before running any passes, all object files that are part of the program must be linked into a single LLVM bitcode file. As a precaution, the instrumentation pass performs a check to ensure that it does not insert PP instructions into modules without an entry point (i.e. a *main* function).

By default, the path profiler does not insert instrumentation to handle incomplete paths in those methods with early or unexpected termination (for instance, a call to `exit()`). In some cases, PP may not produce accurate edge profiles. Specifying the command line option `process-early-termination` in

*This development of path profiling for LLVM was funded in part by the Natural Science and Engineering Research Council of Canada through two Undergraduate Student Research Awards (URSAs). The first was granted in 2009 to Slobodan Pejic and the second in 2010 to Adam Preuss.

```
opt -insert-path-profiling -dot-pathdag -o example.bc example.pp.bc
llvm-ld -native -o example example.pp.bc
./example -llvmprof-out example.llvmprof.out
```

Figure 1: Sample commands to obtain a path profile

```
opt example.bc --o /dev/null -path-profile-loader -path-profile-verifier \
        -path-profile-loader-file example.llvmprof.out \
        -path-profile-verifier-output example.edgefrompath.llvmprof.out
cmp example.edgefrompath.out example.edge.llvmprof.out
```

Figure 2: Sample commands to load and verify a path profile

both the instrumentation and verification phases modifies the path numbering scheme to handle early termination, subsequently allowing path profiles to produce exact edge frequencies. This may incur a significant overhead depending on the position and frequency of method calls in a program. The implementation of early termination processing is discussed in section 3. All other command–line flags specific to PP are explained in the following subsections:

2.1 Obtaining Path Profiles

The PP instrumentation pass is invoked with the command–line option `insert-path-profiling`. Should a user wish to view the derived DAGs of each function in the instrumented program, the option `dot-pathdag` may be specified. The optimizer outputs a DAG of each graph in a `.dot` file [2] named `pathdag.<function-name>.dot`. Once an instrumented program has been obtained, it must be executed to generate the PP information. The output file name containing PP information may be specified with `llvmprof-out <filename>`; otherwise, it defaults to `llvmprof.out`. An example of the instrumentation/execution phase for the program `example.bc` is shown in figure 1.

2.2 Loading Path Profiles

PP information must be readily available to future FDO passes. A *path–profiling loader* (PPL) pass must exist to satisfy its potential dependents. The PPL defines an interface, such that other passes may have access to specific PP information. Invocation of the PPL is accomplished with the command–line option `path-profile-loader`, with an optional argument, `path-profile-loader-file <filename>`, specifying the file with PP information. By default, the PP filename is `llvmprof.out`.

2.3 Verification

A verification pass was created to help instill confidence in LLVM developers that the information generated by PP is accurate. The *path–profiling verifier* (PPV) analyzes information provided by the PPL and derives an edge profile, creating a file that can be compared with the output of LLVM's edge profiler. A byte-per-byte comparison showing that the two files are identical should build confidence that PP is producing information that is consistent with the independently developed edge profiler. Although the PPV runs through LLVM's optimizer, it does not perform any code transformations.

The PPV pass may be invoked with the command-line option `path-profile-verifier`. Additionally, an EP output file name may be specified with the argument `path-profile-verifier-output <filename>`. By default, this file name is `edgefrompath.llvmprof.out`. A combined example of the PPL and PPV passes is shown in figure 2. This example assumes that the unin-

strumented bitcode file is named `example.bc`, and that PP and EP information are available in files named `example.path.llvmprof.out` and `example.edge.llvmprof.out`, respectively.

For `cmp` to detect an exact match, the program binaries for PP and EP must be identically named because command-line arguments at runtime are stored in the profiling files.

The verification pass was tested on a multitude of different programs and inputs, including many in the SPEC CPU2006 suite; all native edge profiles matched the path-derived profiles. Note that when performing verifications, if path or edge counters overflow, it is likely that the edge profile will not be a perfect match.

3 Implementation

The PP addition to LLVM has been implemented in three (mostly independent) modules: instrumentation, runtime and analysis. All three modules share the same code for PN logic - a set of classes to represent nodes, edges and DAGs - but each module can be run independently. The path identification and numbering algorithms are intended to assign unsigned integer values to paths and to convert a function's CFG into a DAG, thus eliminating the possibility of infinite paths in a procedure [1]. If the number of potential paths becomes very large, the PN logic splits DAGs to produce shorter paths (Currently, the splitting threshhold is set to 100,000,000). Otherwise, for certain DAGs, the total number of potential paths can quickly exceed the integer width.

3.1 Unexpected Procedure Termination

Early or unexpected termination of functions further complicate the implementation of PP, which can result in the loss of path execution counts. If a function does not return, the path counters of all functions in the call chain may be lost. The path profiler can optionally assign additional unique paths from function entry points to each function call present in the CFG. Thus, if a function does not return, path counting information will not be lost.

The introduction of these new potential paths incurs a runtime overhead. Before each function call, there must be a path increment in the event that said function does not return. If it returns, then the previous incrementation is corrected and normal execution continues.

3.2 Instrumentation

This phase can be considered a code–transformation pass. It derives a new set of classes from those used for PN, incorporating additional specific procedures for the instrumentation process. Once a DAG has been derived from the CFG, the placement of path counters in a bitcode file is reorganized to produce the lowest runtime overhead. The reorganization is accomplished by minimizing the number of additional instructions along any particular path [3]. Critical edges are split to accommodate the requirements of instrumentation, ensuring a proper PN scheme. One must take care to use the original uninstrumented bitcode file when loading PP information, because the instrumented CFG may not be a perfect match.

3.3 Runtime

At runtime, the profiling external library is responsible for intermediate PP storage and, upon program termination, for writing the path profiles to file. For maximum processing and memory efficiency, executed path counts are stored in arrays for those functions with small potential path counts, and hash tables for large ones. In the event of arithmetic overflow, path counts are capped at the maximum integer width. If a path count exceeds the integer width, the profiler has provided enough information to deem this a hot path.

The following table outlines the file format for path profiles:

PathInfo	Number of Functions
Function Number	Number of Path Entries
Path Number	Path Counter
Path Number	Path Counter
...	...
PathInfo	Number of Functions
Function Number	Number of Path Entries
Path Number	Path Counter
Path Number	Path Counter
...	...
...	...

3.4 Analysis

The PPL interface is implemented as an analysis pass and is designed separately from the generic profile-loader interface to reduce both processing and memory overhead. Further, PP information supersedes EP and BP. The verification pass demonstrates that EP can be precisely derived from PP. Though not implemented, the same can be said for BP. LLVM currently has no optimization passes that use PP, thus it is difficult to conclude what kind of queries will be made to the PP interface. As the use of PP in LLVM evolves, the PP interface can be changed to attend to new needs of the code generator.

For analysis, path numbers and their respective frequencies are loaded into memory. Should an optimizer pass require PP information, any path can be generated "on the fly", by traversing the DAG of the function of interest, because the path number is known [1].

4 Future Work

The following is a list of features not yet supported by this implementation of PP. If a program makes use of such features, instrumenting code with PP could produce undesired results.

- c++ exceptions
- `longjmp()`, `setjmp()`, and other similar functions
- signals
- multithreaded processes whose threads share a common address space (this would be easy to implement by moving path counter increment functions into critical sections)
- programs without a `main` function

5 Conclusion

The objective of this submission is to incorporate an efficient implementation of PP into the LLVM compiler. The PP instrumentation and interface is the first step toward using path profiles for FDO in the LLVM compiler. As optimization passes begin to use PP information in code transformations, it will become clear what specific path-related queries are required; the PP interface can easily be extended to meet these requirements. This submission to LLVM contains all of the components required to instrument a program with PP instructions, gather path profiles derived from program execution, and load profiles back into the optimizer for use in future passes. A verification pass is included to show that the PP results are consistent with the results obtained by the independently developed edge profiler.

References

[1] Thomas Ball, James R. Larus. Efficient path profiling. *International Symposium on Microarchitecture: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, Paris, France, pages 46 – 57, 1996.

[2] http://www.graphviz.org/

[3] Thomas Ball. Efficiently counting program events with support for on-line queries. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5): 1399-1410, September 1994.