Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure

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

Abstract

Profiling monitors a program's execution flow via the insertion of counters at key points in the program. Profiling information can then be used by a compiler's optimization passes to increase the performance of frequently executed sections of code. This document describes the implementation of edge profiling, path profiling and a method with which to combine profiles in the Low Level Virtual Machine (LLVM) compiler infrastructure.

1 Introduction

Profiling is a method of feedback directed optimization (FDO) that inserts additional instructions into a program to monitor its execution. Using gathered profiling information from program execution, optimizing compilers can perform code transformations which aim to enhance the performance of frequently executed (hot) code sections. Profiling can be separated into two categories: online and offline. *Online profiling* provides feedback while a program is running, allowing the program to change based on the current hot code sections. *Offline profiling* generates runtime information which is intended to be used, once a program's execution is complete, for optimization during a second (non-profiled) compilation. In both situations, profiling incurs time and memory overhead to a program's runtime and compilation.

This document discusses the implementation and usage of the following types of offline profiling of a program's control flow graph (CFG):

- *Naïve edge profiling* (NEP): A frequency measurement of each *edge* between *basic blocks* (BBs) – sequences of instructions without incoming or outgoing branches – of a program's CFG. Counters are positioned in either the source or target BB. If an edge's source node has multiple successors and it's target node has multiple predecessors, it is deemed a *critical edge* – there is no safe place to insert instrumentation exclusive to that edge. Thus, the edge is *split* by inserting a new BB between its source and target, where profiling instrumentation may be safely placed.

- *Optimized edge profiling* (OEP): A frequency measurement of a program's CFG edges. It is an optimization of the naïve method, where instrumented edges are based on a maximum spanning tree – all edges do not require instrumentation [1]. The instrumented edges provide enough information to calculate all edge frequencies at a later time, due to conservation of flow.

- *Path profiling* (PP): A frequency measurement of executed paths in a program's CFG. If the CFG contains back-edges, the number of potential paths is unbounded. Thus, path numbering (PN) is based on the algorithm proposed by Ball and Larus [2], which transforms a function's CFG into a directed acyclic graph (DAG) with a bounded number of potential paths, assigning each path a unique number.

The goals of this project were to implement PP into the LLVM compiler and to incorporate a method of combining the accumulated profiles of NEP and PP. LLVM only supports offline profiling. Currently, it contains instrumentation passes for NEP and OEP, and an interface to the compiler that loads the accumulated profiling information.

Traditionally, profiling information is gathered from a

*This development of profiling for the LLVM compiler infrastructure 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.

```
$ cd $HOME
$ mkdir llvm llvm/llvm llvm/llvm-gcc llvm/src llvm/build \
   llvm/build/llvm llvm/build/llvm-gcc
$ svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm/src/llvm
$ svn co http://llvm.org/svn/llvm-project/llvm-gcc-4.2/trunk llvm/src/llvm-gcc
$ cd llvm/src/llvm
$ patch -p0 -i $HOME/llvmprof.patch
$ cd ../../build/llvm
$ ../../src/llvm/configure --prefix=$HOME/llvm/llvm \
   --enable-optimized --disable-assertions
$ make; make install
$ cd ../llvm-gcc
$ ../../src/llvm-gcc/configure --prefix=$HOME/llvm/llvm-gcc \
   --program-prefix=llvm- --enable-languages=c,c++,fortran \
   --enable-llvm=$HOME/llvm/llvm --enable-optimized  --disable-assertions
$ make; make install
```

Figure 1: Example commands which install llvm and llvm-gcc into a linux user's home directory. It is assumed that the LLVM profiling patch is saved in the home directory as well.

single training run (usually a smaller input); optimizations are performed based on the profiling results of that run. However, edge frequency might be highly input dependent.

We implemented a new methodology called *combined profiling* (CP) that enables a variation-preserving combination of profiles from multiple training runs. When using CP, an optimizing compiler can not only examine an average of event frequencies across the different inputs, but it may also perform statistical queries about the distribution of these events across multiple inputs. For instance, some inputs might execute parts of a program's code with extremely high frequency while others do not execute those parts at all. In such situations, one would observe high variance in the profiling results that might influence code transformations more heavily than only examining averages.

The profiling for LLVM was developed under the supervision of Professor Jose Nelson Amaral and 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, reimplementing the PP interface and creating a toolset to manage both *combined edge profiling* (CEP) and *combined path profiling* (CPP). Kaiven Zhou has created a set of scripts to benchmark the varying types of profiling in LLVM.

2 Obtaining Patched Compiler

The LLVM compiler is language independent. A separate front end compiles code to LLVM's intermediate representation, LLVM bitcode, that has equivalent binary and textual representations. Profling related transformations and instrumentation all act exclusively on the bitcode in the back end, which can be compiled to native machine code or executed with `lli`, LLVM's just-in-time (JIT) compiler.

The most recent release of the LLVM compiler may be downloaded via svn from `http://llvm.org/svn/llvm-project/llvm/trunk`. The profiling patch should then be applied in the svn directory. Figure 1 shows sample commands on a linux machine which download, patch and compile LLVM and llvm-gcc – one of the most commonly used front ends for c, c++ and fortran. One should set up the `PATH` environment variable to include the compiler executables, located in `insall-dir/bin`, for both llvm and llvm-gcc.

3 Profiling Implementation

Profiling in LLVM can be separated into three (mostly independent) phases: instrumentation, execution and analysis. The instrumentation phase performs a code transformation pass where profiling instructions are inserted into a given program. At execution, profiled programs must be linked to a lightweight dynamic library: `profile_rt`, that contains functions responsible for storing profiling in-

uint32	ArgumentInfo
uint32	Argument String Length
uint8[]	Argument String
uint8[]	Zero Padding to 32bits
	... Profiling Data ...
uint32	ArgumentInfo
uint32	Argument String Length
uint8[]	Argument String
uint8[]	Zero Padding to 32bits
	... Profiling Data ...
	...

Figure 2: The profiling header file storage format, present at the beginning of each profiling run. `ArgumentInfo` is an identification number used by the compiler.

uint32	EdgeInfo/ OptEdgeInfo
uint32	Number of Counters
uint32	Counter 0
uint32	Counter 1
uint32	Counter 2
uint32	Counter 3
	...

Figure 3: NEP and OEP data storage formats – each field is an unsigned 32 bit integer. `EdgeInfo` and `OptEdgeInfo` are identification numbers used by the compiler.

formation and writing it to disk. If a profiling output file already exists from a previous run, `profile_rt` appends the new profiling data (this is equivalent to `cat`). Once profiling information has been obtained, it can be loaded into the LLVM optimizer `opt` and analyzed to optimize the original, uninstrumented program. This paper's primary focus is the first two phases of profiling which gather varying types of CFG profiles.

3.1 Profiling Header

A header is associated with each profiling trial. It is appended to the profiling output file ahead of any other profiling information for the current run. A profiling file that has accumulated over several runs will have a separate header for each run. The header contains the command line arguments associated with program execution, to aid in differentiating among profiling trials. Figure 2 outlines the profiling header storage format.

3.2 Edge Profiling

The NEP and OEP instrumentation passes share similar implementation: each creates a global array of zero-initialized unsigned 32 bit profile counters (the counters will overflow if the integer width becomes saturated). Edges of a procedure's CFG grow linearly with the number of BBs, making an array a suitable intermediate profiling storage form. Once program execution is complete, the `profile_rt` dynamic library is responsible for writing the gathered profile to disk. LLVM contains an interface that exposes profiling information to profile-guided optimization passes. The table in Figure 3 outlines the format for NEP and OEP.

3.3 Path Profiling

The instrumentation and analysis phases of PP share the same code for PN logic (a set of classes to represent nodes, edges and DAGs). The path identification and numbering algorithms assign 32 bit unsigned integer values to paths. They convert a function's CFG into a DAG, thus eliminating the possibility of infinite paths in a procedure [2]. 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.

3.3.1 Path Profiling Instrumentation

The PP instrumentation pass uses a new set of classes that are derived from those used for PN, incorporating additional procedures specific to 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 [4]. 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 might not be a perfect match – instrumented code might have additional BBs due to edge splitting.

uint32	PathInfo	Number of Functions	uint32
uint32	Function 1	Number of Path Entries	uint32
uint32	Path Number	Path Counter	uint32
uint32	Path Number	Path Counter	uint32
uint32	Path Number	Path Counter	uint32
	...	...	
uint32	Function 2	Number of Path Entries	uint32
uint32	Path Number	Path Counter	uint32
uint32	Path Number	Path Counter	uint32
uint32	Path Number	Path Counter	uint32
	...	...	
	...	...	

Figure 4: Path profiling data storage format – each field is an unsigned 32 bit integer. `PathInfo` is an identification number used by the compiler.

3.3.2 Path Profiling Runtime

At runtime, the profiling external library is responsible for intermediate PP storage and, upon program termination, for writing the path profiles to disk. For maximum processing and memory efficiency, executed path counts are stored in arrays for those functions with a small number of potential paths. If the potential path count reaches 100 000, path counters are stored in a hash table. In the event of arithmetic overflow, path counts saturate 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 table in Figure 4 outlines the file storage format for path profiles.

3.3.3 Path Profile Analysis

The path profile loader (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 edge profiling (EP). A verification pass has been created, which demonstrates that EP can be precisely derived from PP. LLVM currently has no optimization passes that use PP, thus it is difficult to determine the kind of queries that 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.

During 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 [2].

3.4 Early Procedure Termination

Early or unexpected termination of functions further complicates the implementation of profiling, which might result in inaccurate profiling counter values. If a function does not return, an increment instruction might never be executed if it resides after the function call. NEP can avoid this problem by careful placement of edge counters; the issue is increasingly complex for OEP and PP. Currently, OEP does not support early termination (ET).

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 (over the entire function call chain) will not be lost. The introduction of these new potential paths incurs an execution runtime overhead. Before each function call, there must be a path increment in the event that said function does not return. If it returns, the path counter is decremented and normal execution continues.

3.5 Combined Profiling

CP has been implemented as a tool in the LLVM compiler, which has been designed to support CEP and CPP. The tool will merge any number of profiling files or CP files for both CEP and CPP. Combined profiles consist of a calculated mean, variance and histogram to measure the average frequency and distribution of each of their stored profiling counters. CEPs files contain histograms for executed edges; CPPs files contain histograms for the executed paths in the program.

uint64	Sum
float64	Sum of Squared Deviations
float64	Bin Width
float64	Lower Bound
float64	Upper Bound
uint32	Counter ID
uint32	Non-Zero Trial Count
uint8	Bins Used
uint8	Bin Index
float64	Bin Value
uint8	Bin Index
float64	Bin Value
uint8	Bin Index
float64	Bin Value
	...

Figure 5: The CP histogram file storage format.

3.5.1 Histogram Storage

The CP histogram is a set of bins that covers a continuous range, holding a total weight equal to the number of profiling trials. The bin count is specified by the user via the command-line option `-bc=<bin count>`, before histogram generation. There exists an epsilon, currently set to 0.01, which maintains a mimimum bin width to avoid point distributions. In addition, the mimimum range is never allowed to drop below zero. Figure 5 shows the data structure of a CP histogram, along with additional fields used to calculate the mean and variance explained in section 3.5.2.

3.5.2 Incremental Computation of the Mean and Variance

In addition to histogram representation, CP reports the mean and variance of a distribution. CP histograms may be built from a combination of single-input profiles and CPs; hence, incremental computation of these statistical values is crucial. For n trials $x_1 \ldots x_n$, the formulae for calculating the sum and sum S of squared deviations SS of single-input histogram values are as follows:

$$S = \sum_{i=1}^{n} x_i \tag{1}$$

$$SS = \sum_{i=1}^{n} (x_i - \frac{S}{n})^2 \tag{2}$$

Parallel calculations for histograms A and B are computed using the following formulae [5]:

$$S = S_A + S_B \tag{3}$$

```
$ opt -insert-edge-profiling \
  -dot-edge-numbers \
  -o example.bc example.ep.bc
$ llvm-ld -lprofile_rt -native \
   -o example example.ep.bc
$ ./example -llvmprof-out \
  example.llvmprof.out
```

Figure 6: Sample commands to obtain a NEP file and produce CFG graphs of the program's edge numbers.

$$SS = SS_A + SS_B + \frac{n_A n_B}{n_A + n_B} \left(\frac{S_A}{n_A} - \frac{S_B}{n_B} \right)^2 \tag{4}$$

Mean $\bar{x}$ and variance σ^2 are then easily calculated as follows:

$$\bar{x} = \frac{S}{n} \tag{5}$$

$$\sigma^2 = \frac{SS}{n} \tag{6}$$

3.5.3 Frequency Storage

CPs cannot be stored as raw frequencies, otherwise there would be no context in which to compare the gathered profiles over many different inputs. Consider a compiler – instrumented with profiling – that compiles first, a small source file and then a very large one. The counters from the small input might be insignificant, completely overshadowed by those of the large one. To maintain continuity, CEP edge frequencies are stored as fractions with respect to their most immediate dominating edges; CPP path frequencies are saved as fractions relative to the number of times their containing function is executed.

4 Profiling User Guide

Profiling instrumentation is accomplished through optimization passes, using LLVM's 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 passes perform checks to ensure that they do not insert profiling instructions into modules without an entry point (i.e. a `main()` function). Due to critical edge splitting, it is unwise to instrument a program with more than one type of profiling.

Once an instrumented program has been obtained, it must be executed to generate the profiling information. The output file name containing profiling information may be specified with `llvmprof-out <filename>` as a first command-line argument to the instrumented

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

Figure 7: Sample commands to obtain a path profile.

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

Figure 8: Sample commands to load and verify a path profile.

program; otherwise, the profiling filename defaults to `llvmprof.out`. The following sections present many examples of instrumenting and executing programs for varying types of profiling:

4.1 Obtaining Edge Profiles

The NEP and OEP instrumentation passes are invoked with the respective command–line options `-insert-edge-profiling` and `-insert-optimized-edge-profiling`. There exists an additional pass `-dot-edge-numbers`, generating `.dot` graphs with associated edge numbers for each function's CFG in the program [3] . An example of the instrumentation/execution phases for NEP is shown in Figure 6.

4.2 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` must be specified. The optimizer outputs a DAG of each graph in a `.dot` file named `pathdag.<function-name>.dot`. An example of the instrumentation/execution phase for the program `example.bc` is shown in Figure 7.

By default, the path profiler does not insert instrumentation to handle incomplete paths in those methods with ET – for instance, a call to `exit()`. In some cases, PP might not produce accurate profiles. Specifying the command line option `process-early-termination` in both the instrumentation and analysis phases modifies the path numbering scheme to handle ET, subsequently allowing path profiles to produce exact edge frequencies. ET handling might incur a significant overhead depending on the position and frequency of method calls in a program.

4.2.1 Loading Path Profiles

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

4.2.2 Verification

In this project, 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 is 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 8. This example assumes that the uninstrumented 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 NEP must be identically named because

```
$ ./example.nep -llvmprof-output o1.nep input1
$ ./example.nep -llvmprof-output o2.nep input2
$ ./example.nep -llvmprof-output o3.nep input3
$ llvm-cprof example.bc o1.nep o2.nep o3.nep -eo edge1.cp -bc=100
$ ./example.nep -llvmprof-output o4.nep input4
$ ./example.nep -llvmprof-output o5.nep input5
$ ./example.nep -llvmprof-output o6.nep input6
$ llvm-cprof example.bc o4.nep o5.nep o6.nep edge1.cp -eo edge2.cp -bc=15
```

Figure 9: Example commands which gather and combine naïve edge profiles.

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 NEPs 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.

4.3 Building Combined Profiles

A new tool `llvm-cprof` has been created for LLVM which is responsible for building combined profiles. It must take the filename of the profiled program's uninstrumented bitcode file as the first command-line argument. It can take any number of NEP, PP, CEP and CPP filenames as additional command-line arguments, which are used to build the associated CEP and CPP files. The flag `-eo=<filename>` specifies the name of the CEP output file (`edge.cp` by default); `-po=<filename>` specifies the name of the CPP output file (`path.cp` by default); `-bc=<integer>` specifies the number of bins in the output CP histograms (15 by default). Figure 9 shows example commands which gather and combine naïve edge profiles; the same procedure applies to CPP. There also exists a perl script `print-cp.pl` which can be used to view combined profiles

5 Experimental Results

The verification pass was tested on a multitude of different programs and inputs, including all `c` benchmarks in the SPEC CPU2006 suite; all NEP files matched the PP derived EP files. Verification tests were performed on a slightly modified version of the compiler in which path counters were allowed to overflow. Had counter values been restricted to the maximum integer width, it is likely that the verification process would fail due to common edges among executed paths.

Compilation time and runtime experiments were conducted on a series of two processor machines with the following specifications:

- Red Hat 4.1.2-42
- Linux Version 2.6.18-92.1.18.el5 x86_64
- 2x Quad-Core AMD Opteron™ Processor 2350 2000Mhz, 128KB L1, 512KB L2, 2MB L3 shared.
- 2x 4GB Memory

Times were measured for NEP, PP and PP with ET support. Results show that EP, PP and PP with ET compilation time overheads are 49%, 26% and 41% with respect to the uninstrumented compilation time. Runtimes overheads are 28%, 80% and 154%, respectively. OEP is a very recent addition to the LLVM compiler and was not included in the benchmarking experiments because its accuracy has not yet been evalutated.

Table 1 shows the compile time overhead measurements; the associated graph is shown in Figure 10. Table 2 shows the run time overhead measurements. Each benchmark was executed and compiled 3 times; the average time and 95% confidence interval values are recorded in the tables.

All associated runtime graphs for individual benchmark results are included at the end of the document.

6 Future Work

The following is a list of features that are either incomplete or not yet supported by the existing profiling infrastructure in LLVM:

- **Unconventional Jumps in Path Profiling:** Currently, the path profiler does not support many more complicated jumping or branching instructions used by c++ exceptions, `longjmp()`/ `setjmp()` functions or signals.

- **Combined Optimized Edge Profiling:** `llvm-cprof` could be extended to allow the combination of OEP files. A new edge-number identification system would be necessary for users to identify the appropriate edges in OEP.

- **Incremental Combined Profiling:** A future extension of combined profiling would allow `profile_rt` to write directly to CP files. Unfortunately, this is a tedious procedure because `profile_rt` does not have any code in common with the compiler. In addition, a program at runtime has no knowledge of its own CFG. An optimization pass `-generate-edge-dominance` was created which outputs edge dominance information to a file (by default `edgedom.out`, specified with the argument `-edge-dominance-file=filename`) useful to CEP.

- **Larger Profiling Event Counters:** Profiling counters are stored as 32 bit unsigned integers and thus, are prone to arithmetic overflow. The path profiler caps the counters at maximum integer width; the other profiling methods do not test for counter overflow.

- **Multithreaded Programs:** Currently, none of the profiling inplementations in the LLVM compiler supports multithreaded programs; this is not a significant problem for NEP and OEP because they both store their path counters in arrays. However, for large functions, PP makes use of a hash to store profiling counters, creating an issue if two threads are modifying a hash bin at the same time. This problem could potentially be solved by placing counter increment instructions in critical sections, though there could be a drastic performance decrease in those programs with more than one thread.

7 Conclusion

Profiling information is important to optimizing compilers, which perform code transformations in an effort to produce more efficient code. Ideally, multiple training inputs of a single program will guide a compiler to make better code transformations. The compiler will be able to examine an entire statistical distribution, as opposed to information from a single trial (which might not be representative of the entire population). The LLVM compiler currently contains instrumentation passes and a profile loader for NEP and OEP. In this project, PP instrumentation and analysis passes were incorporated into the compiler, along with a new tool to combine NEP and PP files. A perl script was written to examine the CP files; future optimization passes in LLVM should make use of the CP information. This project provides the necessary foundation to future FDO passes in the LLVM compiler infrastructure.

References

[1] Thomas Ball, James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4): 1319 - 1360, July 1994.

[2] 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.

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

[4] 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.

[5] Tony F. Chan, Gene H. Golub, Randall J. LeVeque. Updating Formulae and a Pairwise Algorithm for Computing Sample Variances. *Technical Report STAN-CS-79-773*, Department of Computer Science, Stanford University, 1979.

CPU2006 Benchmark	Non-Profiled Raw Time (s)	Percentage Overhead		
		Edge Profiling	Path Profiling	PP w/ Early Termination
400.perlbench	48.88 ± 0.05	93 ± 3	31 ± 2	55 ± 2
401.bzip2	2.28 ± 0.01	68 ± 2	28 ± 2	42 ± 2
403.gcc	188.8 ± 0.5	48 ± 2	21 ± 2	35 ± 2
429.mcf	0.84± 0.01	32 ± 2	20 ± 2	28 ± 2
433.milc	4.66 ± 0.01	37 ± 2	18± 2	38 ± 2
445.gobmk	110 ± 9	45 ± 2	39 ± 2	50 ± 2
456.hmmer	8.82 ± 0.01	30 ± 2	18 ± 2	39 ± 2
462.libquantum	1.13 ± 0.01	33 ± 2	24 ± 2	42 ± 2
464.h264ref	24.29 ± 0.08	75 ± 2	18 ± 2	29 ± 2
470.lbm	0.42 ± 0.01	36 ± 2	39 ± 2	48 ± 2
482.sphinx3	6.61 ± 0.01	37 ± 2	25 ± 2	49 ± 2
Average	N/A	49 ± 12	26 ± 4	41 ± 5

Table 1: Compile time overhead numerical results with 95% confidence.

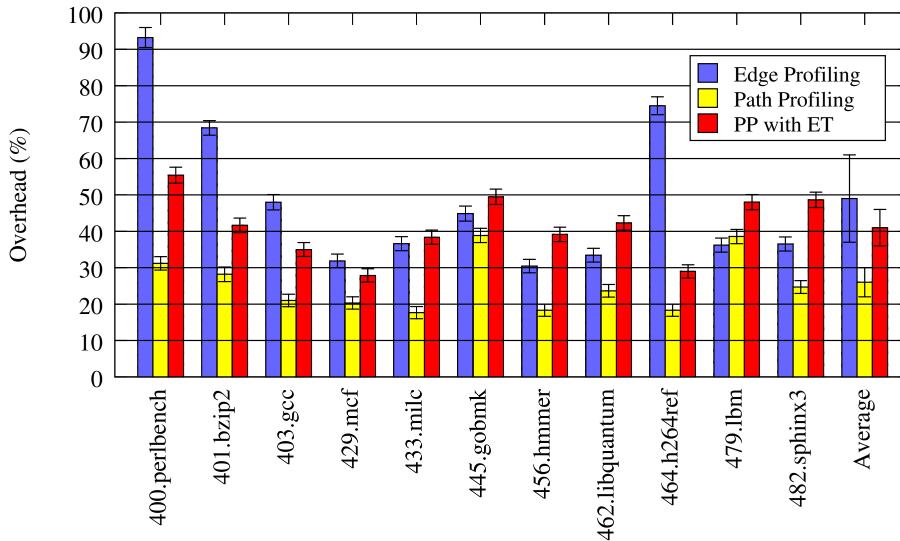


Figure 10: Compile time overhead graph with respect to non-profiled compile time

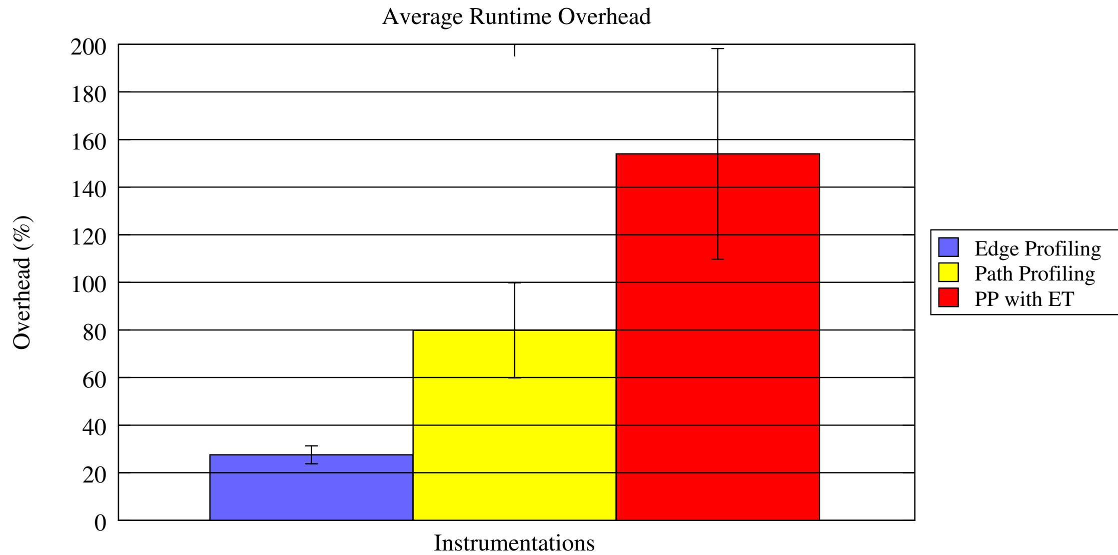
CPU2006 Benchmark	Non-Profiled Raw Time (s)	Percentage Overhead		
		Edge Profiling	Path Profiling	PP w/ Early Termination
400.perlbench – train 1	6.95 ± 0.02	41 ± 2	69 ± 2	117 ± 3
400.perlbench – train 2	10.1 ± 0.3	24 ± 2	31 ± 2	78 ± 3
400.perlbench – train 3	23.11 ± 0.06	39 ± 2	124 ± 3	209 ± 4
400.perlbench – train 4	0.94 ± 0.01	31 ± 2	49 ± 2	98 ± 3
400.perlbench – ref 1	521.1 ± 0.3	35 ± 2	158 ± 4	305 ± 4
400.perlbench – ref 2	167.0 ± 0.5	32 ± 2	73 ± 2	151 ± 4
400.perlbench – ref 3	274 ± 3	36 ± 2	119 ± 3	199 ± 4
400.perlbench – average	N/A	34 ± 4	89 ± 31	166 ± 54
401.bzip2 – train	237 ± 2	22 ± 2	26 ± 2	25 ± 2
401.bzip2 – ref	215 ± 1	28 ± 2	33 ± 2	32 ± 2
401.bzip2 – average	N/A	25 ± 4	29 ± 5	29 ± 5
403.gcc – train	2.09 ± 0.01	34 ± 2	67 ± 2	130 ± 3
403.gcc – ref 1	85.1 ± 0.2	20 ± 2	32 ± 2	53 ± 2
403.gcc – ref 2	106.8 ± 0.1	27 ± 2	55 ± 2	114 ± 3
403.gcc – ref 3	137.2 ± 0.2	24 ± 2	34 ± 2	52 ± 2
403.gcc – ref 4	92 ± 1	28 ± 2	37 ± 2	59 ± 2
403.gcc – ref 5	108 ± 1	24 ± 2	33 ± 2	48 ± 2
403.gcc – ref 6	148 ± 1	25 ± 2	35 ± 2	53 ± 2
403.gcc – ref 7	215 ± 1	17 ± 2	29 ± 2	43 ± 2
403.gcc – ref 8	177 ± 1	22 ± 2	28 ± 2	42 ± 2
403.gcc – ref 9	37 ± 0.1	33 ± 2	69 ± 2	144 ± 3
403.gcc – average	N/A	25 ± 3	42 ± 9	74 ± 23
429.mcf – train	87.1 ± 0.7	4 ± 1	-2 ± 1	6 ± 1
429.mcf – ref	1822 ± 44	10 ± 2	-1 ± 1	11 ± 2
429.mcf – average	N/A	7 ± 4	-1.4 ± 0.7	9 ± 4
433.milc – train	33.36 ± 0.08	4 ± 1	10 ± 2	17 ± 2
433.milc – ref	1022 ± 4	4 ± 1	10 ± 2	17 ± 2
433.milc – average	N/A	3.7 ± 0.3	9.9 ± 0.2	16.8 ± 0.1
445.gobmk – train 1	12.00 ± 0.01	37 ± 2	139 ± 3	290 ± 6
445.gobmk – train 2	103.50 ± 0.02	37 ± 2	221 ± 5	490 ± 8
445.gobmk – train 3	16.28 ± 0.04	37 ± 2	165 ± 4	356 ± 6
445.gobmk – train 4	3.35 ± 0.01	37 ± 2	126 ± 3	270 ± 5
445.gobmk – train 5	2.55 ± 0.01	34 ± 2	101 ± 3	205 ± 4
445.gobmk – train 6	24.76 ± 0.04	37 ± 2	148 ± 4	327 ± 6
445.gobmk – train 7	9.29 ± 0.01	39 ± 2	127 ± 3	273 ± 5
445.gobmk – train 8	22.28 ± 0.04	35 ± 2	156 ± 4	335 ± 6
445.gobmk – ref 1	143.1 ± 0.4	36 ± 2	209 ± 4	433 ± 8
445.gobmk – ref 2	364.9 ± 0.7	37 ± 2	241 ± 5	505 ± 9
445.gobmk – ref 3	197.5 ± 0.3	33 ± 2	180 ± 4	354 ± 6
445.gobmk – ref 4	141.9 ± 0.4	37 ± 2	233 ± 5	490 ± 8
445.gobmk – ref 5	186.2 ± 0.3	37 ± 2	255 ± 5	563 ± 9
445.gobmk – average	N/A	36 ± 0.7	177 ± 26	376 ± 57
456.hmmer – train	114.3 ± 0.1	39 ± 2	41 ± 2	39 ± 2
456.hmmer – ref	819 ± 5	38 ± 2	39 ± 2	36 ± 2
456.hmmer – average	N/A	38 ± 1	40 ± 1	38 ± 2

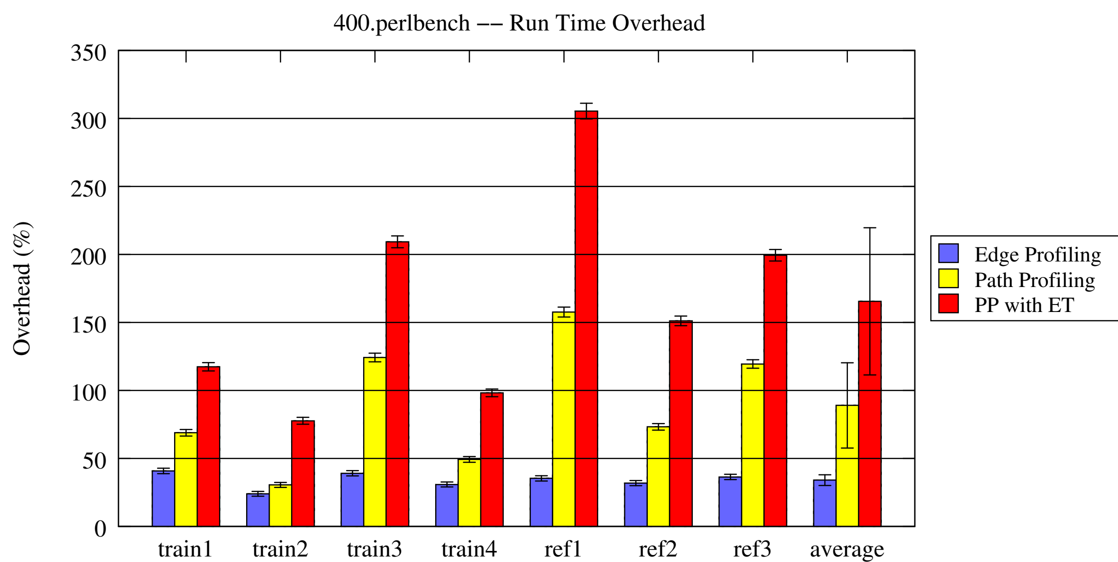
Continued on next page . . .

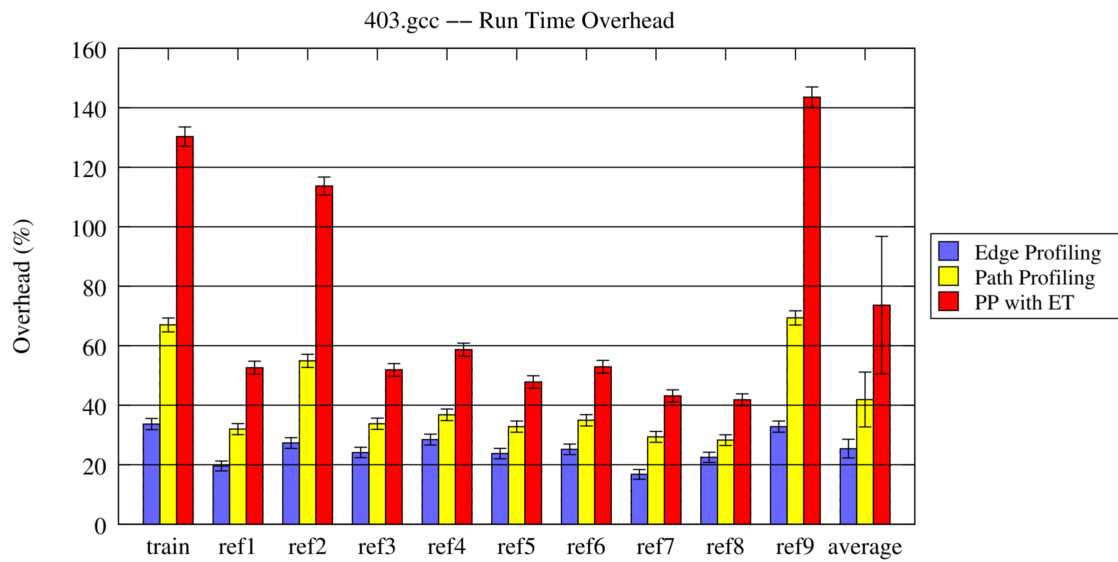
Table 2: Runtime overhead numerical results with 95% confidence.

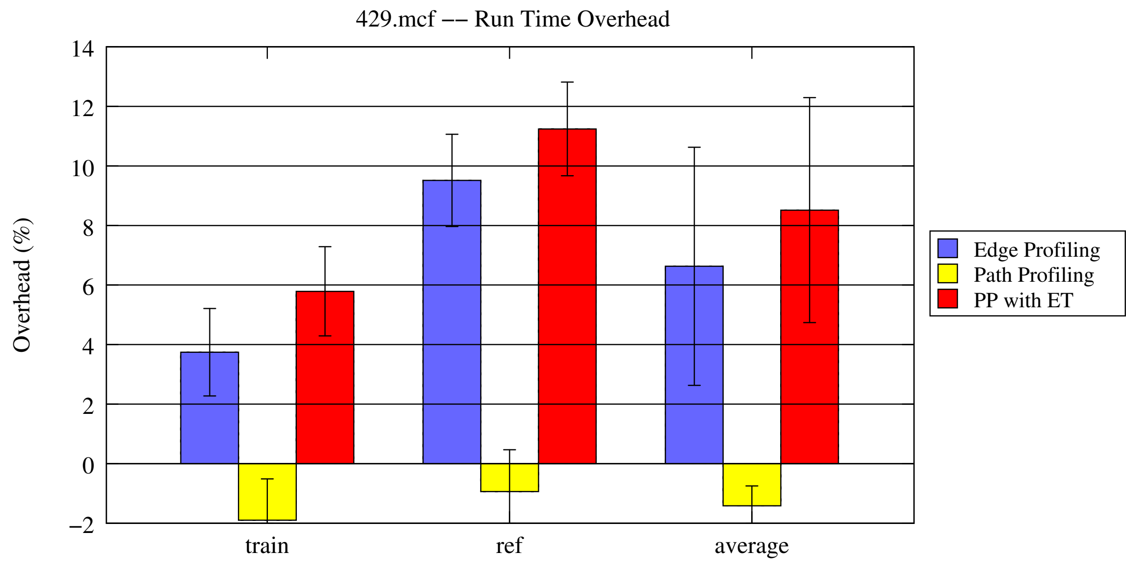
Table 2 continued from previous page.

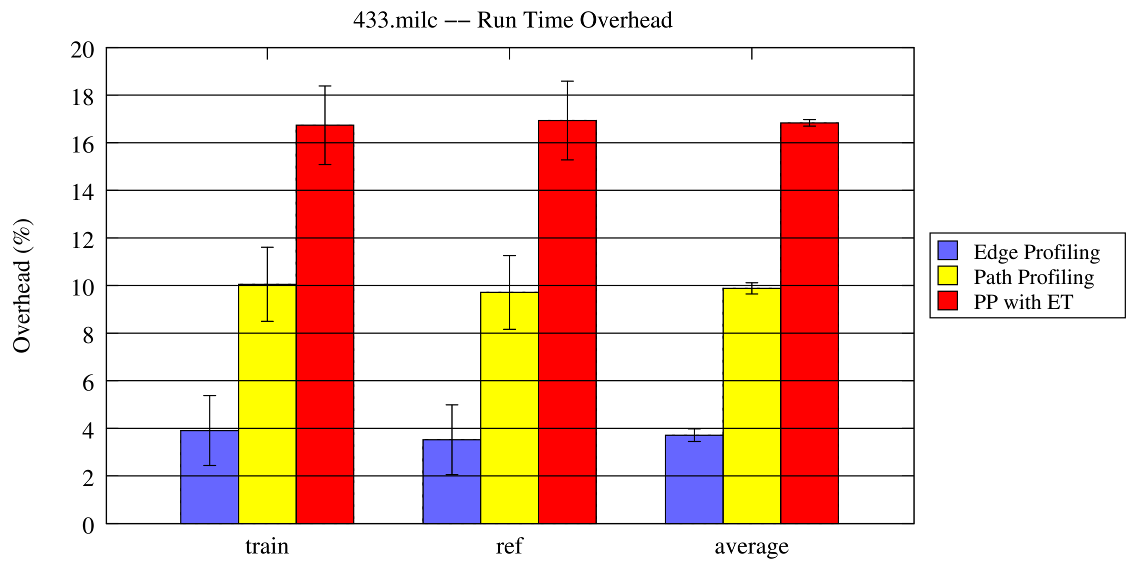
CPU2006 Benchmark	Non-Profiled Raw Time (s)	Percentage Overhead		
		Edge Profiling	Path Profiling	PP w/ Early Termination
462.libquantum – train	3.49 ± 0.03	76 ± 3	77 ± 3	82 ± 3
462.libquantum – ref	1520 ± 4	19 ± 2	29 ± 2	31 ± 2
462.libquantum – average	N/A	47 ± 40	54 ± 35	57 ± 36
464.h264ref – train	217 ± 1	26 ± 2	48 ± 2	72 ± 2
464.h264ref – ref 1	216 ± 1	26 ± 2	48 ± 2	71 ± 2
464.h264ref – ref 2	136 ± 1	21 ± 2	41 ± 2	56 ± 2
464.h264ref – ref 3	1256 ± 11	21 ± 2	41 ± 2	56 ± 2
464.h264ref – average	N/A	24 ± 2	44 ± 3	64 ± 7
470.lbm – train	94 ± 3	3 ± 1	9 ± 2	9 ± 2
470.lbm – ref	962 ± 27	3 ± 1	12 ± 2	13 ± 2
470.lbm – average	N/A	2.6 ± 0.1	10.4 ± 3	11 ± 3
482.sphinx3 – train	26.41 ± 0.02	9 ± 2	16 ± 2	17 ± 2
482.sphinx3 – ref	420.4 ± 0.1	9 ± 2	17 ± 2	17 ± 2
482.sphinx3 – average	N/A	9 ± 0.3	16 ± 1	17 ± 0.2
Average	N/A	28 ± 4	80 ± 20	154 ± 44

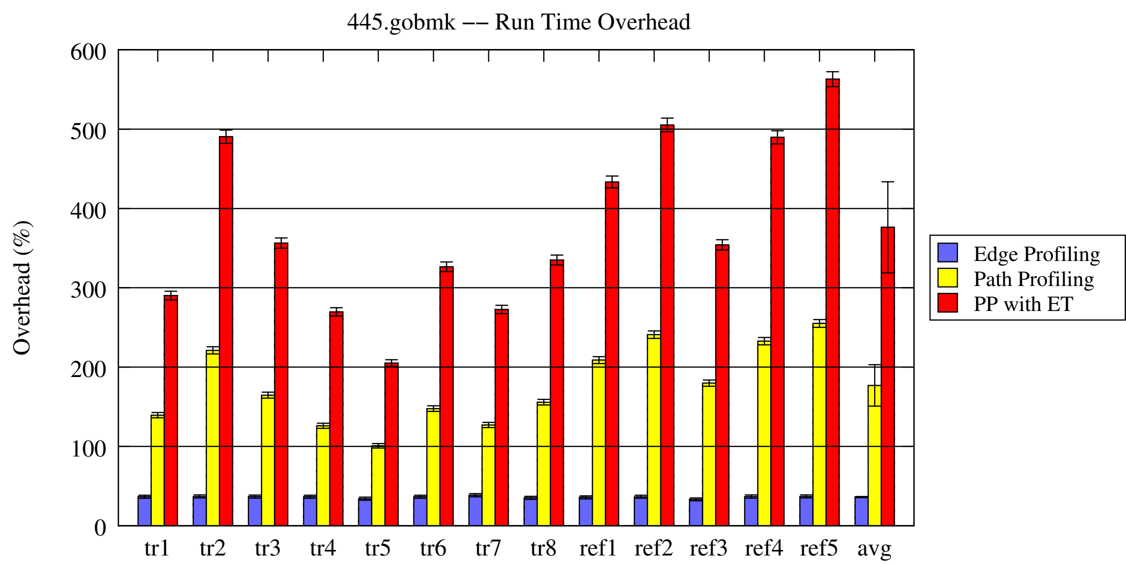


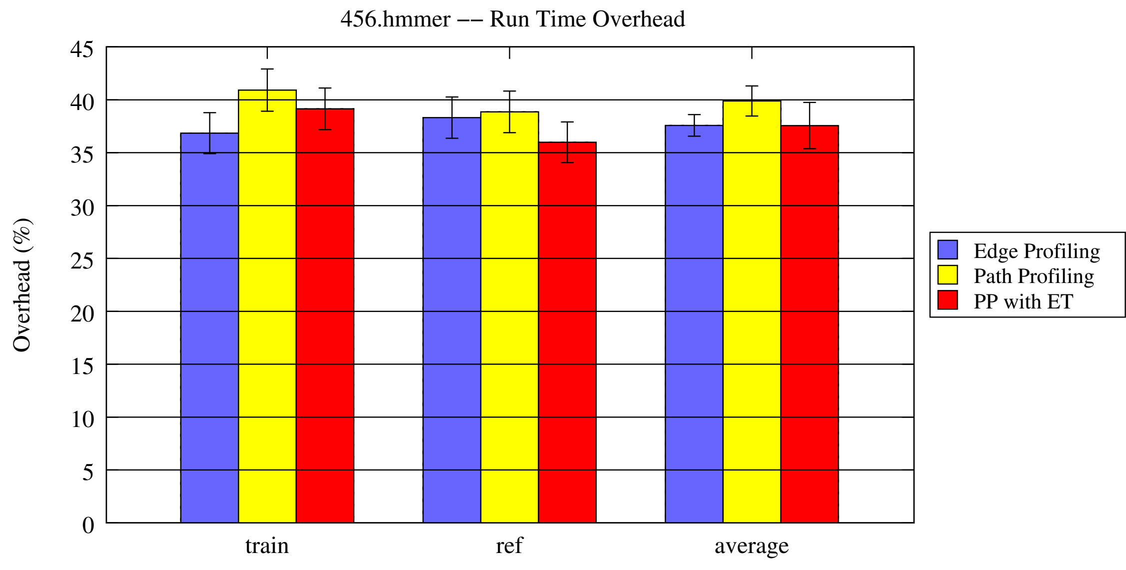

400.perlbench -- Run Time Overhead
Overhead (%)
0
50
100
150
200
250
300
350
train1
train2
train3
train4
ref1
ref2
ref3
average
Edge Profiling
Path Profiling
PP with ET

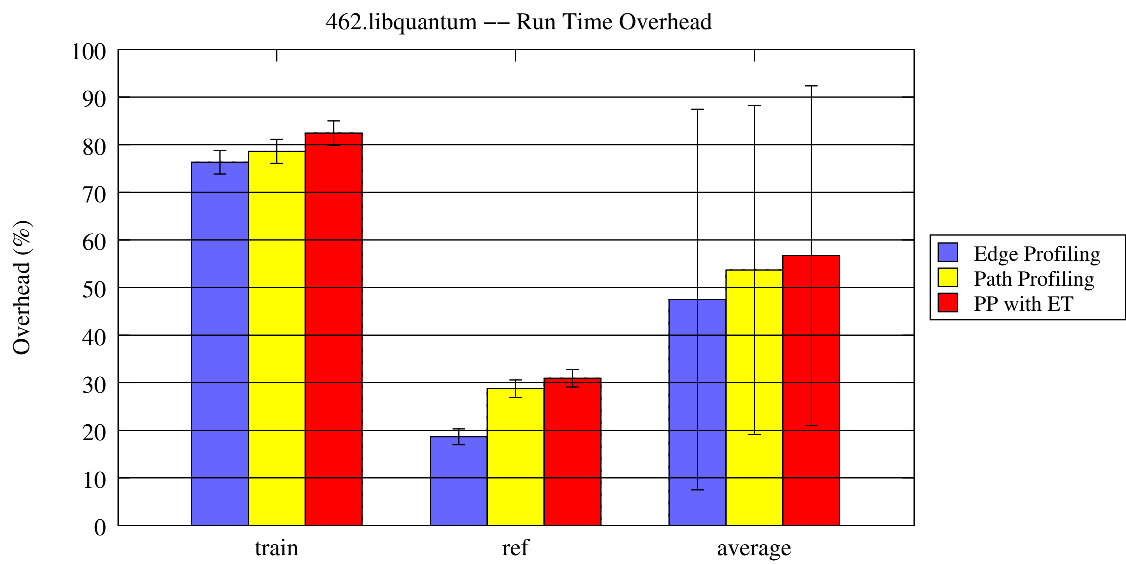

403.gcc -- Run Time Overhead
Overhead (%)
0
20
40
60
80
100
120
140
160
train
ref1
ref2
ref3
ref4
ref5
ref6
ref7
ref8
ref9
average
Edge Profiling
Path Profiling
PP with ET

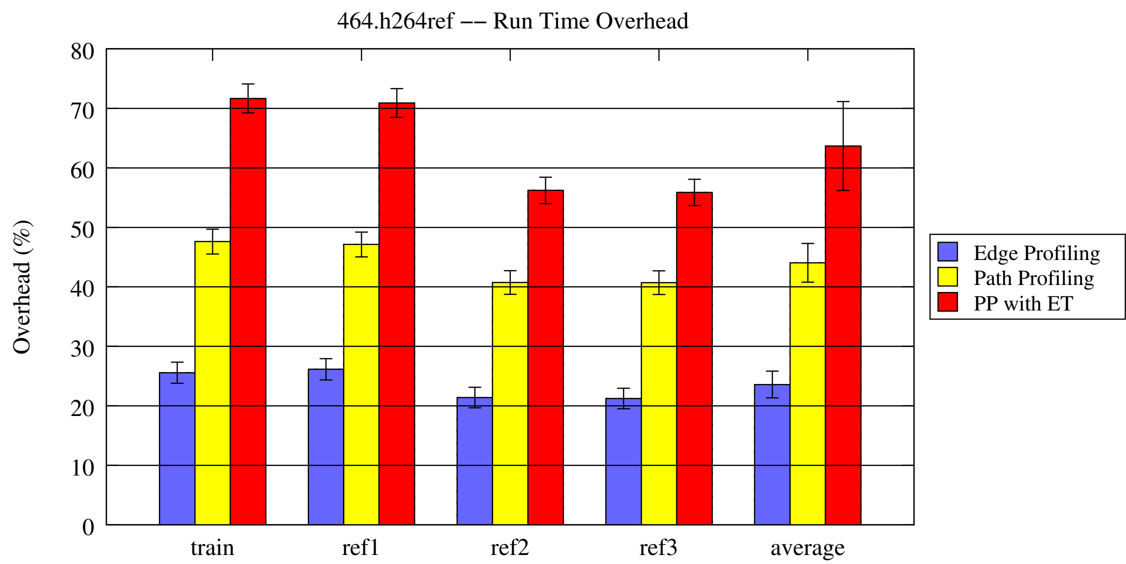

429.mcf –– Run Time Overhead
Overhead (%)
14
12
10
8
6
4
2
0
−2
train
ref
average
Edge Profiling
Path Profiling
PP with ET

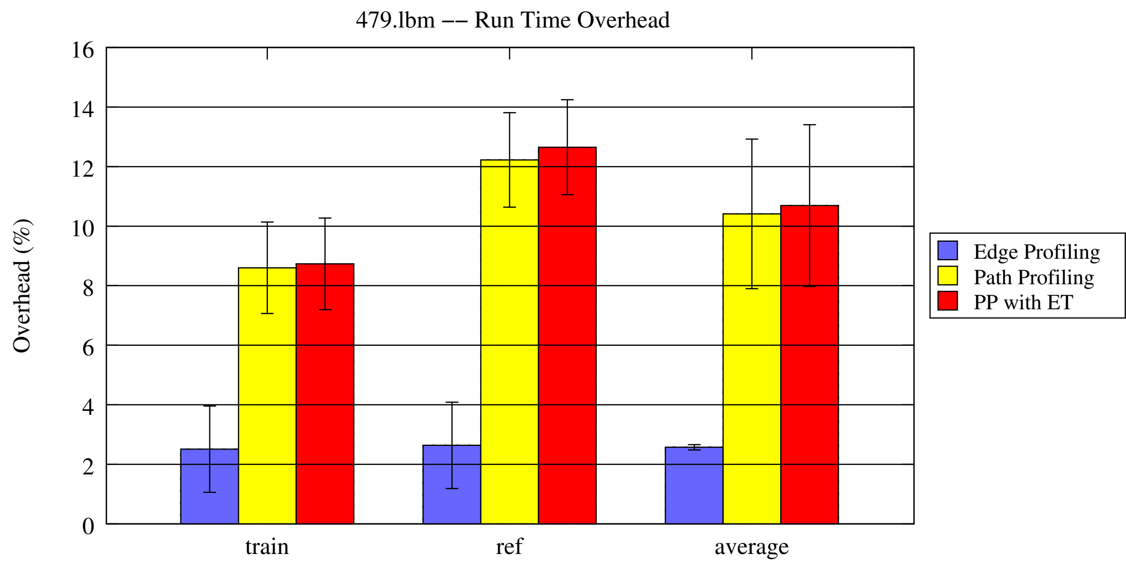

433.milc –– Run Time Overhead
Overhead (%)
20
18
16
14
12
10
8
6
4
2
0
train
ref
average
Edge Profiling
Path Profiling
PP with ET

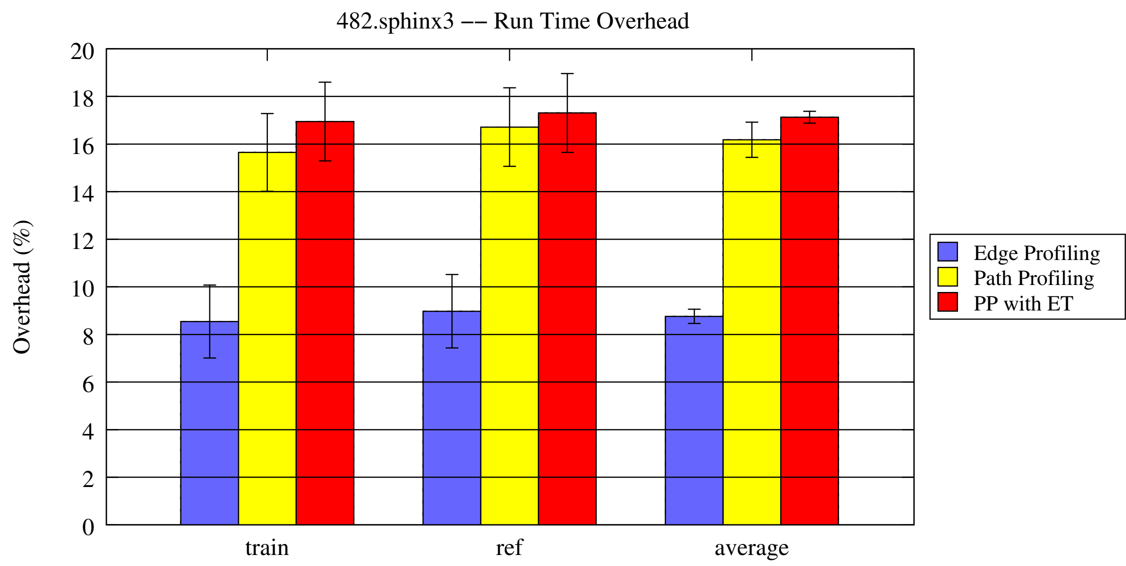

445.gobmk –– Run Time Overhead
Overhead (%)
0
100
200
300
400
500
600
tr1
tr2
tr3
tr4
tr5
tr6
tr7
tr8
ref1
ref2
ref3
ref4
ref5
avg
Edge Profiling
Path Profiling
PP with ET


456.hmmer –– Run Time Overhead
Overhead (%)
0
5
10
15
20
25
30
35
40
45
train
ref
average
Edge Profiling
Path Profiling
PP with ET


462.libquantum –– Run Time Overhead
Overhead (%)
100
90
80
70
60
50
40
30
20
10
0
train
ref
average
Edge Profiling
Path Profiling
PP with ET


464.h264ref –– Run Time Overhead
Overhead (%)
80
70
60
50
40
30
20
10
0
train
ref1
ref2
ref3
average
Edge Profiling
Path Profiling
PP with ET


479.lbm –– Run Time Overhead
Overhead (%)
16
14
12
10
8
6
4
2
0
train
ref
average
Edge Profiling
Path Profiling
PP with ET


482.sphinx3 –– Run Time Overhead
Overhead (%)
20
18
16
14
12
10
8
6
4
2
0
train
ref
average
Edge Profiling
Path Profiling
PP with ET