Dynamic Whole Program Profiling

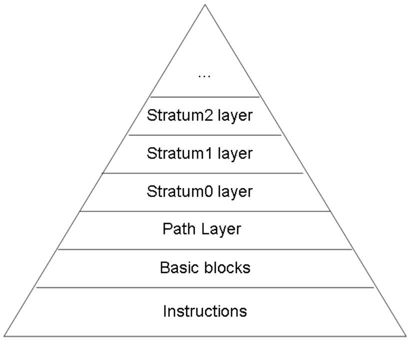
Richard Gorton
GPG Developer Tools
Advanced Micro Devices, Inc

Abstract—**Many methods used to capture and store application traces use separate phases for collection of the raw trace and subsequent compression into a more manageable form. While this is viable for short or partial traces, the storage requirements become a significant challenge when application behavior exceeds 10^12 instructions in length. A new concept, the stratum layer (*pl.* strata), is borrowed from geology and applied to program behavior. Tracking strata in an application enables significant algorithmic compression without the need for an explicit grammar. A tool utilizing dynamic strata collection and storage is described, and the whole program profiles for SPEC CPU 2006 are collected. Performance statistics about the technique are presented, as are various application statistics. The execution-time slowdown for this technique is moderate, while compression ratios are high.**

I. INTRODUCTION

COLLECTING and storing the complete execution trace of an application enables compiler writers, processor architects and researchers to better analyze application behavior. While it is desirable to attempt to keep the execution time overhead of trace collection low, the limiting factor for trace collection is likely to be the patience of the person collecting the data; when the overhead typically exceeds about 50x, the usability and utility of the collection technique is low. However, storing collected traces presents significant challenges - the amount of data which needs to be collected to represent the lossless control flow of an application can easily be multiple terabytes in size if performed without significant compression. As an example, on the x86_64 architecture, the reference dataset for the 454.calculix benchmark in SPEC CPU2006 [1] executes more than 7.3e12 instructions. At an average instruction length of more than four bytes/instruction, merely storing the bytes of each instruction would require more than 30 terabytes. And 454.calculix is not even a particularly long running application - the reference time is under 2 ½ hours. Consider the storage requirements for an application which runs for multiple days. Clearly, significant dynamic compression is needed to facilitate collection of complete execution traces. Whole Program Paths [2] provide a means by which significant compression can be achieved; the application is instrumented to collect acyclic path information, and a formal compression grammar is directly applied to the data generated by the instrumentation program.

This paper extends the path-based concepts of Whole Program Paths [2] via a layered approach – it utilizes the collection of cyclic paths without the use of an explicit grammar to perform algorithmic compression. Conceptually, this approach views application execution behavior as having multiple layers of content built upon lower level layers. Multiple instructions form Basic Blocks; cycles of Basic Blocks form a path; cycles of Repeated Paths form a Stratum Element; cycles of Repeated Stratum Elements form a Stratum Layer. Further higher level strata can be expressed as cycles of repeated lower level strata, as shown in Figure 1.



An implementation of this approach enabled the complete traces for all of the reference datasets of SPEC CPU2006 [1] to be captured and stored in less than 100 GB of storage. The tool was implemented as a PIN [3] instrumentation tool, coupled with some additional C code.

II. TOOL OVERVIEW

A Dynamic Whole Program Profiler tool is a superset of a hot path profiler – in addition to tracking which paths are the most frequently executed, a whole program profiler must track temporal information. Because of this, a side effect of such a tool is that it can report hot path information. In this implementation, multiple processes which communicate via shared memory are utilized to perform the collection and dynamic compression of the data. Doing so enables the computational load to be shared across multiple processor cores. Additionally, it improves performance by moving a significant portion of the computational load out of the binary instrumentation environment and into native code. The implementation is not tightly tied to the PIN environment, and could be ported to use other underlying binary instrumentation tools, such as ATOM [5]. Because the binary instrumentation toolkit is dynamic, precomputed graphs are not available to the instrumentation program, which effectively requires that the path construction occur lazily, and are thus cyclic.

A. Some definitions and abbreviations

Here are some of the definitions and abbreviations used for the remainder of this paper.

Bb – Basic Block

Author footnote left blank for submission purposes

Path – a cycle of Bbs which are tracked by their starting addresses. For example, if the Bb addresses are “AABAABAAB”, this breaks down into the following paths: Path$_0$ is A; Path$_1$ is BA; Path$_2$ is AB; Path$_3$ is B. This evaluates to $(A^2)(BA)(AB)(A^2)(B)$, while the optimal representation of the example paths evaluates to $(A^2B)^3$.

Repeated Path (RP) – consecutive executions of a Path. In the above example, Path$_0$ has an initial count of 2, which is written as Path$_0^2$.

Stratum Element – a cycle of repeated paths. For example, if the sequence of repeated paths is “P_0^7, P_1^{12}, P_0^5, P_1^{12}”, the Strata are $S_0 = P_0^7, P_1^{12}, P_0^5$, and $S_1 = P_1^{12}$.

Repeated Stratum Element – consecutive executions of a Stratum Element. If Stratum Element S_0 was executed 94 times, this would be written as S_0^{94}.

Stratum Layer – A cycle of Repeated Stratum Elements. Stratum Layer$_N$ is comprised of Stratum Elements from Stratum Layer$_{N-1}$. For the initial case of Stratum Layer$_0$, the elements are repeated paths.

DWPP – Dynamic Whole Program Profile

III. THE IMPLEMENTATION

A. Path Construction

As every Basic Block (Bb) is executed, save the disassembly of the Bb once to a file, annotated with the starting address to facilitate reporting. Each Bb is instrumented to provide the Bb starting address and number of instructions to the ExtendPath() routine. The algorithm for this is roughly:

```
ExtendPath(void *BbAddr, uint32_t numBbInsns) {
   // newpath – the path currently under construction
   // prevpath – the previously constructed path
   Increment relevant statistics
   If (BbAddr not already seen in path being constructed) {
      Extend newpath with BbAddr
      newpath.length++;
   } else {// a cycle seen, means the path is complete
      If (newpath == prevpath) {
         prevpath.tripcount++;     // a repeated path
         reset newpath;
      } else {    // not an immediately repeated path
         ProcessPath (prevpath);
         prevpath = newpath;
         reset newpath;
      }
   }
}
```

There is some additional code to collect statistics and check for extremely long paths (an implementation limit of 2048 was used). While such long paths might exist, none have been encountered to date.

The ProcessPath routine stores the path in a hash table, and converts the path data into a unique (uint64_t) path identifier for use elsewhere. The algorithm for the ProcessPath routine is:

```
ProcessPath(struct PATH_QUEUE_ELT *RP) {
   Increment relevant statistics
   Compute-hash-value (RP)
   pathID = Search-the-hash-table-and-add (RP)
   PassRPtoStrataServer(pathID, RP.tripcount, isFinalPath);
}
```

For performance considerations, the communication with the strata analysis code is done via shared memory, with the strata analysis server running as a separate process. When the shared memory buffer is filled with (pathID, tripcount) pairs, the strata server will process them.

B. Strata construction

Instead of constructing cycles of Bbs, the strata construction uses [pathID, tripcount] pairs to construct cycles of Repeated Paths. The basic algorithm for extending a stratum layer in the ExtendStrataLayer() routine is very similar to the ExtendPath() algorithm, except that the parameters are passed as a union:

```
ExtendStratumLayer([pathID, tripcount]) { // int,int
   Uint64_t value64 = [pathID, tripcount];
      // convert the pair into a single uint64_t
   Increment relevant statistics
   If (value64 not already seen in cycle being constructed) {
      Extend Stratum$_0$ Element(value64)
      Stratum$_0$ Element.length++;
   } else {// a cycle seen, completes a Stratum$_0$ Element
      // newStratum is the newly constructed cycle
      // prevStratum is the immediately previously
      //     constructed cycle
      If (newStratum == prevStratum) {
         prevStratum.tripcount++;
         reset newStratum;
      } else {
         ProcessStratum(prevStratum);
         prevStratum = newStratum;
         reset prevStratum;
      }
   }
}
```

Once unique Repeated Stratum Elements are discovered, they are handed off to the ProcessStratum routine. Much like to the ProcessPath routine, ProcessStratum stores the strata data in a hash table, assigns a unique identifier to to it, and passes a [stratumID, tripcount] to yet another server. It could be an additional stratum layer server, but in the simple instantiation, this happens to be a final data compression server process, which is also communicated with via shared memory:

```
ProcessStratum(struct STRATUM_ELT* aLayer) {
   Increment relevant statistics
   Compute-hash-value (aLayer)
   stratumID = Search-the-hash-table-and-add (aLayer);
   tripcount = aLayer->tripcount;
   PassStratumtoCompresionServer (stratumID, tripcount);
}
```

C. The compression server

The compression server is a simple general purpose data compressor (using the system libz.so) to further compress the data (in this case, [stratumID, tripcount] pairs) prior to writing it to a file.

D.Reported results & data files

A set of 6 data files are written to during the collection: a pair of files comprising the temporal strata information, a file containing the unique strata$_0$ details, a file containing the unique path details such as the disassembly of pertinent Bbs, a log containing path statistics, and a log containing strata statistics. The path log also contains a report of the hottest N paths by trip count * instruction length, in disassembled form.

IV.IMPLEMENTATION ENVIRONMENT

The system used to capture the complete traces for all of SPEC CPU2006 [1] had the following relevant characteristics: x86_64 Linux® (Fedora 7), which uses gcc 4.1.2; 8GB memory, 160 GB of disk; 2.6Ghz dual core processor. The binary instrumentation toolkit used was PIN [3] version 2.2-15113. The SPEC CPU2006 benchmarks were compiled at an optimization level of –O2, and run in "base" mode. The code for the tool is available at **http://www.[URL]**. PureDB [7] was used as a simple database to store content, and libz.so was used to provide a final level of general compression of content in some of the data files.

V.RESULTS

A.Execution time performance

For the machine used, a run of SPEC CPU2006 [1] using the reference datasets was initially performed without any instrumentation, followed eventually by a run (albeit a single iteration) using the DWPP tool. Full results are in Table 1.

Benchmark	Ratios: no tool	Ratios: DWPP	DWPP overhead
400.perlbench	12.8	0.416	30.76
401.bzip2	10	0.11	90.9
403.gcc	9.54	0.734	12.99
429.mcf	7.53	0.297	25.35
445.gobmk	15.4	0.332	46.38
456.hmmer	10.3	0.0115	895.65
458.sjeng	12.5	0.0655	190.83
462.libquantum	17.2	1.85	9.29
464.h264ref	16.1	0.384	41.92
471.omnetpp	9.02	0.643	14.02
473.astar	8.22	0.373	22.03
483.xalancbmk	6.16	0.562	10.96
410.bwaves	5.88	0.992	5.92
416.gamess	14.3	0.963	14.84
433.milc	11.9	2.95	4.03
434.zeusmp	9.28	3.28	2.82
435.gromacs	8.27	1.42	5.82
436.cactusADM	6.9	2.38	2.89
437.leslie3d	6.31	1.82	3.46
444.namd	11.5	0.158	72.78
447.dealII	14.8	0.414	35.74
450.soplex	11.1	0.158	70.25
453.povray	14.9	0.703	21.19
454.calculix	4.36	0.315	13.84
459.GemsFDTD	7.66	3.00	2.55
465.tonto	6.71	0.382	17.56
470.lbm	13.3	6.22	2.13
481.wrf	7.73	0.862	8.96
482.sphinx3	14.7	0.943	15.58

The slowdowns ranged from 2.1x on 470.lbm, to 895x on 456.hmmer. The geometric mean for all of the SPEC CPU2006 reference datasets is 16.9, with only 2 having slowdowns of over 100x.

B.Compression

The compression ratio is calculated using an average instruction length of just over 4.2 bytes. This was computed from a static disassembly of libc and the emacs executable on the test system. The dynamic average instruction length was not computed, but could be tracked by further modification of the DWPP tool. Thus, the number of dynamically executed instructions * 4.2 is computed to be the "uncompressed" size. The size of all files written by the DWPP tool is the "compressed" size. The adjusted compression ratios are given in Table 2.

Application and run number	Number Instructions Executed	Net file sizes	Adjusted compression ratio
astar_1	4.218E+11	1.238E+09	1428
astar_2	8.532E+11	1.572E+09	2276.4
bwaves_1	3.740E+12	2.350E+09	6682.2
bzip2_1	4.275E+11	9.098E+08	1969.8
bzip2_2	1.788E+11	3.208E+08	2339.4
bzip2_3	3.090E+11	4.129E+08	3141.6
bzip2_4	5.457E+11	9.554E+08	2398.2
bzip2_5	5.992E+11	8.317E+08	3024
bzip2_6	3.416E+11	7.453E+08	1923.6
cactusADM_1	2.780E+12	5.300E+06	2202937.8
calculix_1	7.393E+12	5.489E+09	5653.2
dealII_1	1.905E+12	1.612E+09	4964.4
gamess_1	1.089E+12	3.823E+08	11965.8
gamess_2	7.889E+11	2.440E+08	13578.6
gamess_3	3.436E+12	1.005E+09	14351.4
gcc_1	8.115E+10	1.101E+08	3091.2
gcc_2	1.520E+11	2.562E+08	2490.6
gcc_3	1.470E+11	1.313E+08	4699.8
gcc_4	1.110E+11	1.250E+08	3725.4
gcc_5	1.172E+11	1.147E+08	4288.2
gcc_6	1.593E+11	1.499E+08	4464.6

gcc_7	1.840E+11	1.392E+08	5548.2
gcc_8	1.738E+11	9.877E+07	7392
gcc_9	5.945E+10	2.017E+08	1234.8
GemsFDTD_1	2.500E+12	6.407E+06	1638928.2
gobmk_1	2.313E+11	2.449E+09	394.8
gobmk_2	6.136E+11	3.913E+09	655.2
gobmk_3	3.172E+11	2.253E+09	588
gobmk_4	2.304E+11	2.863E+09	336
gobmk_5	3.287E+11	2.568E+09	537.6
gromacs_1	2.000E+12	6.088E+08	13792.8
h264ref_1	5.037E+11	3.235E+08	6539.4
h264ref_2	4.202E+11	2.486E+08	7098
h264ref_3	3.814E+12	2.093E+09	7652.4
hmmer_1	8.953E+11	4.270E+09	877.8
hmmer_2	1.898E+12	7.647E+09	1041.6
lbm_1	1.277E+12	3.242E+06	1654371.6
leslie3d_1	4.131E+12	9.413E+06	1843031.4
libquantum_1	2.264E+12	6.180E+08	15384.6
mcf_1	3.892E+11	2.048E+09	798
milc_1	1.169E+12	1.513E+08	32470.2
namd_1	2.307E+12	1.404E+09	6900.6
omnetpp_1	5.940E+11	5.546E+08	4498.2
perlbench_1	1.097E+12	7.943E+08	5796
perlbench_2	3.805E+11	9.319E+07	17144.4
perlbench_3	6.992E+11	1.936E+08	15166.2
povray_1	9.997E+11	1.260E+08	33314.4
sjeng_1	2.306E+12	4.328E+09	2234.4
soplex_1	3.761E+11	2.066E+09	764.4
soplex_1	3.870E+11	8.206E+08	1978.2
sphinx_1	3.128E+12	2.409E+09	5451.6
tonto_1	3.739E+12	1.526E+09	10285.8
wrf_1	3.894E+12	9.027E+07	181167
Xalan_1	1.202E+12	2.856E+08	17677.8
zeusmp_1	2.039E+12	3.154E+06	2714703.6

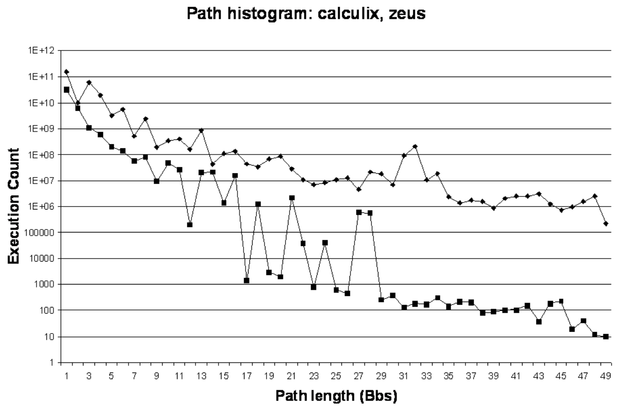
These range from a low of 336 (4th dataset in 445.gobmk) to 2.71 Million for 434.zeusmp. The mean of the compression ratios was almost 192,000, while the geometric mean was 7069.

C.Path statistics

Data about mean path lengths and the number of unique paths for a particular application/dataset combination are also reported by the tool. These are shown in Table 3.

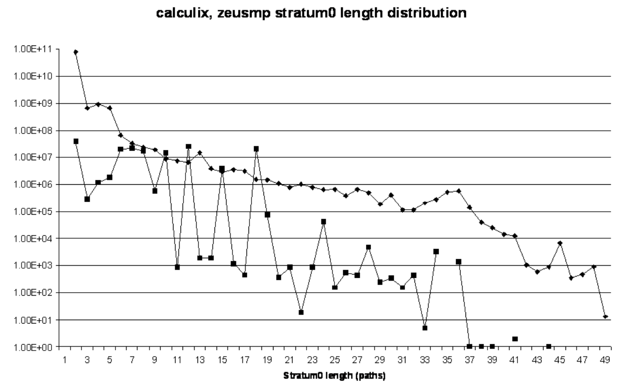
Application	Number of Paths	Mean Path Length in Bbs	Unique Paths
astar_1	10548297506	6	11289
astar_2	16697033084	8	19380
bwaves_1	1.80494E+11	1	1324
bzip2_1	15239062264	4	37049
bzip2_2	12561468469	2	12454
bzip2_3	11183249209	4	7766
bzip2_4	32750358253	2	31969
bzip2_5	15845715380	6	14653
bzip2_6	13486555907	3	37091
cactusADM_1	5761332355	10	5777
calculix_1	2.56404E+11	2	13594
dealII_1	1.06533E+11	3	28421
gamess_1	19351628019	5	18592
gamess_2	15340767343	4	18548
gamess_3	68983774581	4	20841
gcc_1	5541329594	3	149617
gcc_2	6389527821	5	281720
gcc_3	13195331219	2	169939
gcc_4	9587888749	2	163620
gcc_5	11060764022	2	152740
gcc_6	15127658645	2	185512
gcc_7	17063849928	2	149302
gcc_8	15170012455	2	113852
gcc_9	2235594999	6	265215
GemsFDTD_1	30541153872	2	4481
gobmk_1	6627190679	6	3457064
gobmk_2	15978829737	7	5154542
gobmk_3	15471200501	4	2874788
gobmk_4	6449098436	6	4102959
gobmk_5	8841556002	7	3577811
gromacs_1	15139716159	5	5405
h264ref_1	19572134739	2	9093
h264ref_2	21768823736	1	149100
h264ref_3	1.9502E+11	1	199877
hmmer_1	27159450117	4	9649
hmmer_2	57506091271	4	3270
lbm_1	3970413618	7	507
leslie3d_1	1.13395E+11	1	1886
libquantum_1	2.26541E+11	1	1146
mcf_1	16715313437	4	2441
milc_1	21465737881	2	1902
namd_1	46065088739	2	5180
omnetpp_1	12726589529	11	24448
perlbench_1	18476153860	12	132088
perlbench_2	3739943883	21	25236
perlbench_3	10361809465	13	51770
povray_1	10081996720	15	33263
sjeng_1	78029410069	6	2006699
soplex_1	26022072112	2	23619
soplex_1	26510019728	2	16634
sphinx_1	1.91775E+11	1	11244
tonto_1	1.12739E+11	3	54473
wrf_1	1.56834E+11	1	15435
Xalan_1	82228019093	3	22942
zeusmp_1	40390252069	1	2648

Because the DWPP tool is a superset of a hot path profiler, it is trivial to derive statistical data about paths from a run. For example, plots of path length (in Bbs) vs. path trip count in Figure 2.



D.Strata statistics

Similarly, plots of strata length (in numbers of paths) can be plotted from the data as show in Figure 3.



Strata centric data are shown in Table 4. Some interesting characteristics appear – for some of the application/dataset combinations, there are only a handful of unique strata, while in others there are a very large number of unique strata. The number of immediately consecutive executions of any given strata is variable, and is clearly dependent upon the dataset used.

Application and run	Unique strata	Total Strata	Consecutive executions
astar_1	216725	1.514E+09	49855727
astar_2	174433	2.278E+09	29862256
bwaves_1	589	1.678E+10	7886622569
bzip2_1	2952183	636494417	64910223
bzip2_2	328460	253512210	6814624
bzip2_3	249687	479882052	115277700
bzip2_4	1676991	878405761	148328837
bzip2_5	1471941	636991494	66470614
bzip2_6	2532973	499311315	46659535
cactusADM_1	921	1.667E+09	1638347722
calculix_1	292937	8.179E+10	2.721E+10
dealII_1	637944	9.638E+09	3702997850
gamess_1	71226	2.115E+09	845557941
gamess_2	94822	1.706E+09	973215692
gamess_3	105316	9.423E+09	3091842046
gcc_1	121652	125954760	9570987
gcc_2	448882	257486208	37621706
gcc_3	196276	135584262	15385895
gcc_4	172603	158218562	9468755
gcc_5	161262	126838013	13553830
gcc_6	213331	154307156	20188679
gcc_7	142393	239746514	26643040
gcc_8	109381	194460576	20120248
gcc_9	350422	105818453	10554208
GemsFDTD_1	1820	377977122	357852740
gobmk_1	504845	303331740	37570664
gobmk_2	884401	809819115	135773748
gobmk_3	483723	977022181	347132641
gobmk_4	567802	279714633	35362127
gobmk_5	561704	414931850	62979348
gromacs_1	404537	1.12E+09	504504822
h264ref_1	190506	1.33E+09	605894273
h264ref_2	190331	747986989	321559454
h264ref_3	673614	8.264E+09	4422079684
hmmer_1	1549022	2.216E+09	35505202
hmmer_2	1543836	4.397E+09	41474736
lbm_1	227	42665658	32942217
leslie3d_1	610	529258683	443713452
libquantum_1	57273	1.254E+10	3009396591
mcf_1	607635	2.069E+09	690305335
milc_1	1132	6.083E+09	2257099434
namd_1	2269578	985288902	145758661
omnetpp_1	102058	732083937	23405995
perlbench_1	438635	1.961E+09	388446892
perlbench_2	32379	590322600	212547492
perlbench_3	111628	719276959	377253717
povray_1	47414	735294691	40473467
sjeng_1	2772428	4.94E+09	136589445
soplex_1	1157099	1.614E+09	246245241
soplex_1	699130	1.905E+09	858662299
sphinx_1	244483	7.651E+09	5121578643
tonto_1	55423	9.24E+09	6341782946
wrf_1	31058	4.358E+09	3738161098
Xalan_1	269080	1.597E+09	1021390036
zeusmp_1	1716	168257099	162929849

E.Some comments on 456.hmmer, 458.sjeng, 401.bzip

The three worst cases for execution-time performance were 456.hmmer, 458.sjeng, and 401.bzip. There is a common set of attributes for these: they have more unique strata elements than the width of the hash table to store the strata (which was 256K), and only a very small percentage (under 3%) of the strata elements had a consecutive trip count greater than one. This strongly implies that the performance on these would significantly increase with a wider (and thus less deep) hash table for this particular data structure.

VI. ENGINEERING CONSIDERATIONS AND LIMITATIONS

While there were no formal collection-time performance constraints or requirements, there were a lot of implementation decisions made to minimize memory consumption and maximize collection performance. The benchmark 445.gobmk has a very large number of unique paths; it required significant work to keep experimental runs from swapping due to this characteristic. At the time development started, PIN [3] did not support multi-threaded tools, and this also contributed to the use of multiple processes. Once the code had been rearranged to utilize separate processes, 456.hmmer, 458.sjeng, and 401.bzip drove algorithmic changes to boost performance.

Program phase change behavior was assumed; to compensate for this, the hash tables are periodically sorted to reduce pointer chasing. The amount of performance gain by doing this was never carefully measured, but empirically seemed to make things “go faster”.

Attempts were made to compute and construct the Strata_1 Layer, but that very quickly was abandoned due to the amount of additional memory consumed (many runs started swapping). At present, the tool is limited to collect profiles from single-threaded applications, and only the control flow information.

VII. RELATED WORK

Whole Execution Traces [4] provides a unified format for multiple kinds of profiles which are instrumented at compile time with a statement (“Trimaran’s intermediate level statement”) [10] granularity. Compression levels averaged 41 for the datasets involved; collection performance overhead is not described except as executed on a simulator.

Whole Program Paths [2] was implemented with a static binary instrumentation tool, and the collected control-flow profiles were compressed separately from collection. Compression levels ranged from 7.3 to 392.8; little detail about execution time performance is provided – an instrumented database had a 15.6x slowdown for trace collection alone.

Nested Loop Recognition [8] analyzes data address traces and reconstructs control flow loops. The first 100e6 load instructions were traced, with compression ratios for control flow achieved over 100,000 (vs. bzip2) in some cases. No details of execution performance overhead were given.

Seekable Compressed Traces [9] utilizes multi-stage algorithms tailored to the type of trace collected, and also provides mechanisms to start playback and analysis at arbitrary points in the trace. Compression ratios exceeded 130,000 for one test case; no performance data about trace collection were given.

Of the mentioned works, only Whole Program Paths [2] focused solely upon collection of control flow traces, making direct comparison difficult. Curiously, there is little data about absolute trace collection overhead performance, possibly because the scale of trace collection of any kind to date has exceeded the capacity of commonly available hardware, or a need to use a simulator to collect some of the data.

VIII. CONCLUSIONS & FUTURE DIRECTIONS

Numerous further investigations suggest themselves, but none are currently underway. Some of these include:

- Investigation of isometric cyclical paths, esp. rotational isomers (ABCD and BCDA are rotational isomers, while ACBD and ABCD are simple isomers)
- Can phase behavior be easily detected from the collected data?
- Higher Strata Layer compression beyond Layer_0 - which would require systems with much more memory. Some initial experiments hinted that a combinatorial explosion of data happens above Layer_0, which means that the pyramid diagram of [Figure 1] may be more correctly shaped as an hourglass.
- Perform direct execution performance and compression ratio comparisons of SPEC CPU2006 using the techniques described in [2,4,8,9] as applied to complete control-flow traces.

Using a DWPP tool, it is possible to collect and store complete program instruction traces on a reasonably configured computer system with moderate overhead. This may enable broader investigations into path and strata characteristics on a wide range of arbitrary applications. Such a tool also provides a means of non-statistical (lossless) analysis for tool chain developers and processor architects when coupled with a playback mechanism. And as applications continue to scale up in size and duration, collection of relevant traces/profiles will continue to be a challenge, and require new techniques to simply collect these data.

IX. REFERENCES

[1] SPEC CPU2006 web site: http://www.spec.org/cpu2006

[2] J.R. Larus, “Whole Program Paths”, *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 259-269, Atlanta, GA, May 1999.

[3] PIN web site: http://www.pintool.org. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005, pp. 190-200.

[4] Xiangyu Zhang, Rajiv Gupta, "Whole Execution Traces," micro, pp.105-116, 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'04), 2004”

[5] Amitabh Srivastava , Alan Eustace, ATOM: a system for building customized program analysis tools, Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, p.196-205, June 20-24, 1994, Orlando, Florida, United States

[6] Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Nicholas Nethercote and Julian Seward. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.

[7] puredb source code: http://pureftpd.sourceforge.net/puredb/

[8] Ketterlin, A. and Clauss, P. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the Sixth Annual IEEE/ACM international Symposium on Code Generation and Optimization* (Boston, MA, USA, April 05 - 09, 2008). CGO '08.

[9] T. Moseley, D. Grunwald, and R. V. Peri. Seekable compressed traces. In Proceedings of the 2007 IEEE International Symposium on Workload Characterization (IISWC), 2007.

[10] *The Trimaran Compiler Research Infrastructure*. Tutorial Notes, November 1997.