# PEBIL: Binary Instrumentation for Practical Data-Intensive Program Analysis

Michael A. Laurenzano[†] Joshua Peraza[§] Laura Carrington[†] Ananta Tiwari[†] William A. Ward, Jr.[‡] Roy Campbell[‡]

[†] Performance Modeling and Characterization Laboratory
San Diego Supercomputer Center
University of California, San Diego
michaell@sdsc.edu, lcarring@sdsc.edu, tiwari@sdsc.edu

[§] Dept. of Computer Science and Engineering
University of California, San Diego
jperaza@cse.ucsd.edu

[‡] High Performance Computing Modernization Program
United States Department of Defense
william.ward@hpc.mil, roy.campbell@hpc.mil

***Abstract*—In order to achieve a high level of performance, data intensive programs such as the real-time processing of surveillance feeds from unmanned aerial vehicles, genomics sequence comparison or large graph traversal require the strategic application of multi/many-core processors and co-processors using a hybrid of inter-process message passing (e.g. MPI and SHMEM) and intra-process threading (e.g. pthreads and OpenMP). To facilitate program and system design decisions, program runtime behavior gathered through binary instrumentation is useful because it enables inspection of the low-level interactions between a data intensive program and a multi-core processor or many-core co-processor. This work details two novel mechanisms in the PEBIL binary instrumentation platform that make it well-suited for analyzing data-intensive programs by providing (1) support for fast lookup of instrumentation thread-local storage (ITLS) and (2) support for the fast enabling and disabling of instrumentation at runtime as a methodology for supporting sampling. These features are compared to two other popular binary instrumentation platforms, Pin and Dyninst, in both analytical and empirical terms for programs implemented using the popular but disparate parallelization models MPI and OpenMP. Empirical comparisons are made for two binary instrumentation applications that are critical to the analysis of data-intensive programs, basic block counting and memory address trace collection. These empirical results show that PEBIL is unrivaled in terms of overhead for basic block counting, introducing an average of 18% extra runtime for MPI programs and 116% for OpenMP programs as opposed to 60% (MPI) and 232% (OpenMP) for Pin and 20% (MPI) and 14743% (OpenMP) for Dyninst. For memory address trace collection that makes use of the conventional optimization of sampling 10% of the memory addresses of a program to reduce processing time, PEBIL also introduces the lowest overheads of 144% (MPI) and 222% (OpenMP) compared to 313% (MPI) and 360% (OpenMP) with Pin and 1113% (MPI) and 89075% (OpenMP) with Dyninst.**

## I. Introduction

Understanding the behavior of data-intensive programs is critical for effectively utilizing existing computing resources and for understanding and architecting future systems. Programs that rely on processing large amounts of data abound – processing surveillance feeds, genomics, social data, meteorological and complex physics simulations to name a few. Meanwhile, data-intensive computing increasingly relies upon complex models of parallelization in order to achieve higher levels of throughput. A key technology that enables software and hardware designers to understand the behavior of data-intensive programs is *binary instrumentation* – rewriting or translating a compiled executable in order to insert extra code into the program, usually to collect information about the behavior of the program as it is running. This allows important behaviors within data-intensive programs, such as execution patterns or memory behavior, to be examined at very fine levels of detail.

Binary instrumentation also has applications in other areas of computer science. It has been used historically and in recent years to support open source and commercial development/debugging support tools as well as a range of commercial and academic research projects in the areas of branch prediction analysis [1], cache behavior analysis [2], speculative execution emulation [3], call graph analysis [4], undefined value and memory leak detection [5], taint [6] and other data flow analysis [7], performance optimization [8] and modeling [9], energy optimization [10] and modeling [11], thread race detection [12], program porting [13], code coverage [14], program fault tolerance [15], and to develop security attacks [16] and defenses [17].

This work details two novel features of the PEBIL [18] binary instrumentation platform that position it as a practical and low-overhead vehicle for the analysis of data-intensive programs: (1) its mechanism for providing instrumentation thread-local storage (ITLS) as a means of delivering thread-specific program analysis in an efficient manner and (2) its mechanism for enabling and disabling instrumentation at runtime in order to maximize the effectiveness of sampling as an approach for relieving data collection and processing overheads. To highlight these features and to compare PEBIL's performance to two other state of the art binary instrumentation platforms, Pin [19] and Dyninst [20], this work presents experiments centered around two binary instrumentation applications that are crucial for understanding data-intensive program behavior – basic block counting and memory address trace collection. Basic block counting is useful for learning about the execution patterns and hot spots of a program, particularly when combined with static analysis that allows features of the static executable such as operation counts/types/sizes, data flow/dependence and control flow relationships to be viewed in terms of how they fit into dynamic program behavior. In the context of a data-intensive program, understanding a program's behavior is often largely a matter of understanding how data moves through a system's memory hierarchy. Binary instrumentation allows a memory address stream to be captured and analyzed in any number of ways in order to extract properties of interest such as cache hit rates, reuse distances, or spatial/temporal locality information.

Through a series of experiments given in Section IV, the overhead of PEBIL for basic block counting is shown to be 3.3x smaller than Pin for MPI programs, 2x smaller than Pin for OpenMP programs, about the same as Dyninst for MPI programs and 127.1x smaller than Dyninst for OpenMP programs. For sampling a memory address trace, in Section V PEBIL is shown to introduce overheads that generally decrease as a linear function of the amount of the address stream that is discarded due to sampling. At the conventional sampling rate of 10% [21][22], PEBIL introduces overheads that are 2.2x smaller than Pin for MPI programs, 1.6x smaller than Pin for OpenMP programs, 7.7x smaller than Dyninst for MPI programs and 401.2x smaller than Dyninst for OpenMP programs.

The rest of this paper is organized as follows: Section II discusses related work in binary instrumentation and background information on the binary instrumentation platforms that are discussed throughout the paper: PEBIL, Pin and Dyninst. Section III provides a description of the experimental system, software and benchmark programs used to achieve the empirical results given throughout the paper. Section IV provides a description of PEBIL's mechanisms for providing instrumentation thread-local storage along with a series of experiments designed to examine the runtime overhead of that approach and how that overhead compares to the approaches of Pin and Dyninst. Section V details PEBIL's mechanism for enabling/disabling instrumentation at runtime along with a series of experiments designed to demonstrate its effectiveness as a mechanism for providing sampling. Finally, Section VI concludes.

## II. Background and Related Work

Binary instrumentation is a technique that has been utilized for over twenty years to collect information from running programs. Early efforts to this effect were dedicated profilers that performed specific limited functions on a binary. For example, Pixie [23] was a binary instrumentation tool for MIPS that instrumented an executable so that it maintained basic block and conditional branch counters. A few years later, the qpt [24] tool was written for the MIPS and SPARC platforms, which was capable of counting basic block executions as well as collecting the memory address trace of a running program. Later efforts in binary instrumentation focused on delivering binary instrumentation capabilities as a *platform* or *toolkit*, on top of which programmers could write their own customized program analysis tools. The earliest of these toolkits were the Executable Editing Library (EEL) [25] (generalized from the qpt tools) and ATOM [26], a binary instrumentation toolkit written for the Alpha AXP platform. Binary instrumentation continues to enjoy popularity among researchers on modern computing platforms like PowerPC [27] and the x86 family, which is the platform targeted by PEBIL. As a platform that has remained popular for several decades, x86 invariably has been the target of many binary instrumentation toolkits. Of these, the Valgrind [28] and DynamoRIO [29] projects deserve mention, though are not examined in detail in this work because they distinguish themselves in terms of functionality at the cost of runtime overhead. The remainder of this section describes several of those projects: PEBIL, Pin and Dyninst, which are the representatives of three different approaches to binary instrumentation that, based on previously published results, are most likely to introduce low runtime overheads for tools like basic block counting and memory address tracing.

### A. Background on PEBIL

PEBIL is a platform for developing tools to instrument ELF/Linux executables on the x86 and x86_64 platforms. PEBIL functions as a static binary rewriter, making modifications to the program before it executes. PEBIL operates by reading the binary file, generating and inserting instrumentation, then writing a modified binary to disk that can be run at any time. The location and behavior of the inserted instrumentation code is defined by user-supplied *instrumentation tools* that are written using PEBIL's API. Given this set of user-defined instrumentation points, PEBIL utilizes a technique called code patching to integrate the instrumentation code into the program's text/data, generating instrumentation code and data in a way that ensures that the instrumentation is transparent to the program. That is, PEBIL inserts instrumentation in such a way that the running instrumented program preserves the exact behavior of the original program while running the inserted instrumentation and collecting information about the program as a side-effect. More details about preserving transparency and PEBIL's instrumentation mechanisms can be found in [18].

PEBIL was designed primarily with the goal of producing efficient instrumented code, which collects information about the running program while imposing as little runtime overhead as possible. Efficiency is paramount because one of the goals of PEBIL is to allow for data collection on large and long-running programs, which may run for hours or days and utilize hundreds or thousands of CPUs. On such resource-intensive programs, performing program analysis with as little extra overhead as possible introduces fewer resource constraints on existing and future systems and reduces its turnaround time. This drive for efficiency continues to motivate PEBIL's development, often resulting in innovations within its design. This work details two such innovations: how PEBIL provides access to instrumentation thread-local storage (ITLS) and PEBIL's mechanism for enabling and disabling instrumentation at runtime as a means of efficiently providing sampling.

ITLS is provided to a running multithreaded program by PEBIL in the form of a hash table that is made available to each process. Each entry in this hash table is a small pool of memory that is made available to a particular thread and can contain that thread's local data structures or references to its data structures. A thread's memory pool can be accessed quickly by a running thread because PEBIL utilizes a very fast hash function, allowing the pool's location to be found from scratch in as few as three x86_64 instructions. PEBIL also uses data flow analysis to attempt to find a register in which to cache this location, allowing it to be reused between instrumentation points under certain circumstances. Details on this scheme can be found in earlier work [30] and in Section IV.

To support sampling PEBIL also provides the means to quickly activate and deactivate any instrumentation point at any time during a program run. More information about this mechanism is given in Section V, though it accomplishes activation and deactivation by embedding information about instrumentation, such as its location and size, into the instrumented binary executable, allowing a runtime support library to quickly swap `nop` instructions for instrumentation code through a single short and shallow function call. While this mechanism is more limited in functionality than what is provided by Pin and Dyninst, it allows instrumentation to be transitioned from one state to the other very quickly. These quick transitions are advantageous when instrumentation needs only the active/inactive distinction, which can be the case for instrumentation involved in producing information that will be sampled.

## B. Background on Dyninst

Dyninst [20] is a dynamic binary instrumentation (DBI) platform and static rewriter that supports a variety of platforms, including x86 and x86_64. Like PEBIL, Dyninst uses code patching to introduce instrumentation into a program. Dyninst provides no specific support for providing ITLS to a running thread, though an ad hoc mechanism can be built using its facilities for writing instrumentation snippets. Instrumentation snippets are customized hand-written sequences of low-level statements designed to quickly accomplish routine instrumentation tasks. Within Dyninst, instrumentation snippets can be written to utilize the identifier of the executing thread in order to index a data structure that holds the location of every thread's ITLS. This support for utilizing the thread identifier in hand-coded instrumentation is somewhat similar to PEBIL's in concept, though while PEBIL uses a single instruction to get the thread identifier into a register Dyninst uses a much more elaborate mechanism involving at least two function calls plus all of the extra code this entails (e.g., including code to protect registers that are defined by those functions). Dyninst also lacks a facility for caching the thread identifier or other thread-related information that might allow instrumentation code to reuse the location of thread-local data once computed. As shown in Section IV, these factors combine to introduce very large overheads when utilizing ITLS within instrumentation tools developed with Dyninst. Dyninst as a DBI is able to arbitrarily remove and insert any instrumentation at any time during the program run. Compared to PEBIL's simple mechanism of simply turning instrumentation into either an active or inactive state, this provides far higher degrees of control over how instrumentation is done. However, this flexibility comes at a cost of runtime overhead, making it inefficient as the basis for performing sampling because sampling requires reusing the same instrumentation in the same locations and must be inserted and removed frequently.

## C. Background on Pin

Pin [19] is a dynamic binary instrumentation platform developed and maintained by Intel for use on Intel x86, x86_64, Xscale (no longer supported) and Itanium (no longer supported) platforms. Pin uses a technique called just in time (JIT) compilation in order to instrument a running program, meaning that Pin generates and introduces instrumentation code into the program just prior to executing that code for the first time. Pin introduces a number of optimizations into its basic JIT model in order to improve performance, most notably utilizing the powerful post-link optimizer Ispike [31] in order to optimize the sequences of instrumented code that are generated. Pin provides ITLS to a running thread by storing its location *at all times* in some general purpose register that is stolen from the program. This technique is very effective at minimizing the overhead of accessing that thread-local data, particularly in code that has lower levels of register pressure. Pin's register stealing approach is possible because of its use of Ispike, which allows the program to simply be reorganized around the stolen register where necessary. Pin provides functionally similar support to Dyninst in terms of removing and reinserting instrumentation during an instrumented program run, allowing for all instrumentation to be cleared from the program using `PIN_RemoveInstrumentation` and for arbitrary instrumentation to then be generated and inserted back into the program. Like Dyninst, Pin's mechanism is far more rich than what PEBIL provides but comes at a cost of extra runtime overhead in order to employ it.

## III. Experimental Setup

The remainder of this paper presents a number of empirical experiments designed to measure the overhead related to several mechanisms within the binary instrumentation platforms PEBIL, Pin and Dyninst. These experiments are driven largely by two binary instrumentation applications that are critical as vehicles for understanding the behavior of data-intensive programs, basic block counting and memory address trace collection, and which are important enough that they comprise the motivation for some of the earliest instances of binary instrumentation [23][24]. For all instrumentation tools written using PEBIL, Pin and Dyninst, aggressively optimized implementations of each are implemented using PEBIL version 2.0.0, Pin 2.12-53271 and Dyninst 7.0.1 respectively. Each result presented here is the mean of three independent runs of that particular experiment. For the sake of simplicity in making comparisons, tools for all three platforms are written so that they perform no instrumentation for code residing in shared libraries. Though it also functions as a static rewriter, all experiments utilize Dyninst's dynamic binary instrumentation (DBI) functionality. To avoid unfairly penalizing Dyninst for this choice, the timers that collect the runtime during all Dyninst-related experiments are started after the program has been fully instrumented; the overhead of initially instrumenting the program is ignored.

All experiments are performed on a dual-socket, 8-core Intel Xeon X3450. Each core of the X3450 has a 32KB dedicated L1 cache and 256KB of L2 cache. All four cores in a socket share 8MB of L3 cache and both sockets on the board share 16GB of main memory. All of the experiments are either pure OpenMP or pure MPI; no hybrids are used.

That is, all OpenMP experiments are run using a single process with some number of threads executing within that process and all MPI experiments are run with some number of processes, each of which contains a single thread. For OpenMP and MPI respectively, 8-way runs are executed so that each thread or process is pinned to each of the 8 available cores throughout the entire run of the experiment; 4-way runs are similarly pinned to logical processors 0 and 2 of both processors throughout the life of the experiment.

TABLE I: Descriptions of Experimental Programs

| Name | Description | Input Set | Thread/Process Count |
|---|---|---|---|
| BT (NPB) | block tri-diagonal solver | B | 4 |
| CG (NPB) | conjugate gradient | B | 8 |
| DC[*](NPB) | data cube | W | 8 |
| EP (NPB) | embarrassingly parallel | B | 8 |
| FT (NPB) | 3D fast Fourier Transform | B | 8 |
| IS (NPB) | integer sort | C | 8 |
| LU (NPB) | lower-upper Gauss-Seidel | B | 8 |
| MG (NPB) | multi-grid on mesh sequence | B | 8 |
| SP (NPB) | scalar penta-diagonal solver | B | 4 |
| HMMER | bio. sequence database search | glob4/uniprot | 8 |

[*] OpenMP only.

The benchmarks used through these experiments are the OpenMP and MPI implementations of version 3.3 of the NAS Parallel Benchmarks (NPBs) [32], as well as version 3.0 of the bioinformatics code HMMER [33]. All benchmarks are compiled with gcc version 4.4.3. Table I provides brief descriptions of these programs, their data sets and the process/thread counts used, which are kept consistent between OpenMP and MPI for the experiments performed on each program. In all cases, data collected during the instrumented program run (basic block counts or memory addresses) is tracked per thread and per MPI rank.

## IV. Instrumentation Thread-local Storage

Usefully supporting multithreaded programs within a binary instrumentation platform involves providing a mechanism that allows running threads to access instrumentation thread-local storage (ITLS). As opposed to simply providing thread-safe access to shared data, providing ITLS allows for the possibility for the information gathered by the instrumentation tool to be thread-specific, which is useful for discovering potential imbalances between threads. Beyond this, since each thread often runs on its own hardware context, per-thread results can be useful for characterizing how each thread utilizes its private context (e.g., a branch predictor or L1 cache) without the involvement of unrelated threads running on different contexts. The remainder of this section details the PEBIL model for providing ITLS then employs a series of experiments based on two binary instrumentation applications, basic block counting (Section IV-A) and memory address tracing (Section IV-B), designed to uncover how this threading model compares empirically to those of both Pin and Dyninst. See Section II for background on how ITLS is provided within Pin and Dyninst.

PEBIL provides thread-local data structures to an instrumented multithreaded program by providing a hook to thread creation that allows a copy of thread-local data structures to be generated at the time of thread creation. The data structures created therein are made accessible though a single table, shared by all threads within a process, which contains a small pool of memory for each thread. This memory pool can contain anything of interest to the thread, but is currently only 32 bytes. In practice, therefore, this limits the pool to holding simply the addresses of other interesting thread-local data structures for all but the simplest instrumentation applications. When collecting memory address traces, for example, this pool can contain the address of a buffer that holds unprocessed memory addresses that have been collected from the program. The remainder of this section discusses how a thread accesses its private memory pool at runtime as well as an optimization implemented on top of that access mechanism that allows the location of the memory pool to be cached for short periods of time within a running program.

*1) Accessing Instrumentation Thread-Local Storage at Runtime:* Each thread has access to a small pool of private memory through a shared table that it is provided to each process in a PEBIL-instrumented multithreaded program. If $TID$ is the unique identifier for a thread, then the formula for deriving a thread's index into this table is given by the expression $(TID >> m)\&(2^n - 1)$, where $m$ and $n$ are parameters that can be customized. This formula yields $IDX \in [0, 2^n{-}1]$, which is simply bits $m$ through $m{+}n{-}1$ of $TID$. From the standpoint of efficiency, this method is perfect since it can generate $IDX$ from scratch in as few as three x86_64 instructions[1], however it will generate identical values for any threads whose $TID$ is identical in bits $m$ through $m + n - 1$.

In principle there is no guarantee of uniqueness for these bits. In practice, however, conflicts of this sort have never been encountered with PEBIL's current default values of $m = 12$ and $n = 16$ when running up to 16 threads per process. To detect conflicts, PEBIL intercepts all thread create calls, verifying for each new thread that the $IDX$ of the new thread does not conflict with the $IDX$ for any other existing thread. If a conflict is detected, PEBIL generates a runtime error rather than falling back to a slower mode that can resolve conflicts or guarantee that they will not occur. In such cases execution can be retried with different values of $m$ and $n$, potentially using a different or larger set of bits of $TID$ to produce $IDX$ in order to reduce the likelihood of $IDX$ conflicts between threads. It should be noted that each additional bit used to produce $IDX$ (that is, each increment of $n$) doubles the size of the table of each process's memory pool, which puts practical limits on the size of $n$.

*2) Caching the Location of a Thread's Memory Pool:* Even though the sequence of instructions that computes the location of a thread's private memory pool is short, that sequence may need to be executed very frequently. Detailed instrumentation applications typically require frequent access to the memory pool, every basic block for basic block counting and every memory instruction for memory address tracing. Instead of requiring the location of the memory pool to be recomputed every time a thread needs to access thread-local data, PEBIL attempts to cache the computed location in a dead register so that it need not be recomputed by every

[1]In x86_64 a running thread's unique identifier is stored in `%fs:0x10`. $IDX$ can therefore be generated using a sequence which has a `mov`, a `shr`, then an `and` instruction.

subsequent instrumentation point. To do this, PEBIL currently examines code at the function level to try to identify whether any single register is dead throughout the function's execution. If no such register is found, PEBIL must generate code that recomputes the location of the memory pool every time it is required. However, if such a dead register is available within a function PEBIL inserts code to compute the location of the memory pool only at the entry and reentry (that is, immediately following a call to another function) points of the function.

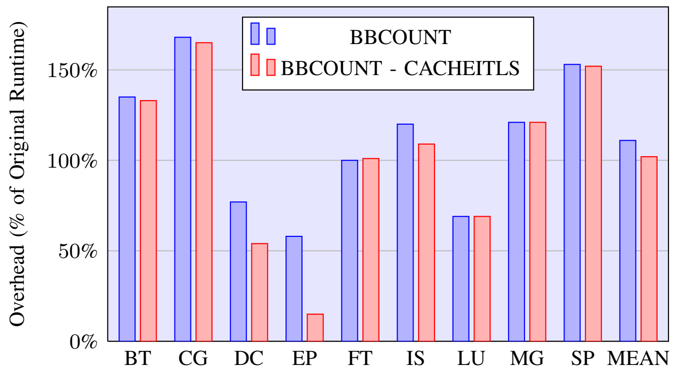


Fig. 1: Overhead for running programs instrumented by PEBIL for basic block counting, with and without caching the location of instrumentation thread-local storage (ITLS) in a dead register.

Figure 1 shows the runtime overhead of a basic block counter on the NAS parallel benchmarks both with and without this optimization. For most benchmarks the optimization leads to little or no improvement but for DC and EP it is highly effective, reducing the overhead from 77% to 54% for DC and from 58% to 15% for EP. Further inspection into this phenomena reveals that for DC and EP, PEBIL is able to employ this optimization inside functions that are particularly important, thereby having a relatively large impact on overhead, whereas it is only employed in relatively unimportant functions in the other benchmarks. These results suggest that the caching optimization is effective where applicable, though somewhat limited in impact primarily due to the fact that a single register must be found to be dead everywhere in a function. In order to increase its efficacy, future work within PEBIL should include extending this optimization, perhaps to relax the requirement that the same register be dead everywhere within a function or to involve smaller code structures like loops or basic blocks.

## A. *Case Study: Basic Block Counting*

A basic block is a unit of contiguous code within a program that has a single entry point and a single exit point, making basic blocks counts a useful vehicle for performing compiler optimizations [34][35], building performance models [36][37], detecting program phases [38], and performing many other kinds of program analysis. For example, features of program execution like instruction counts and mixes require only static analysis combined with basic block execution counts. Basic block counting is also heavily utilized in previous binary instrumentation literature [18][19][26][39] as a test of the overhead of the binary instrumentation platform because it requires relatively heavy intrusion into the execution of the program (at every basic block) to perform a very small amount of work (incrementing a counter), thereby highlighting the overhead introduced by the platform.

Figures 2 and 3 show the overhead of basic block counting, expressed as a proportion of the execution time of the original uninstrumented program runtime for MPI and OpenMP programs respectively. A value of 0% therefore represents a run with no slowdown whatsoever, while larger values represent higher overheads. In Figure 2 the three series labeled PEBIL-MPI, Pin-MPI and Dyninst-MPI show the overhead of basic block counting for MPI-parallelized runs of each of the test programs with PEBIL, Pin and Dyninst respectively. These results are consistent with other literature performing similar tests [18][39]: the overheads for PEBIL and Dyninst are low, averaging 18% and 20% respectively, while the overhead for Pin is somewhat higher, averaging 60%. This extra overhead in Pin is largely due to the fact that Pin incurs extra overhead as the result of its reliance on just-in-time compilation as its method for introducing instrumentation code into the program.

Figure 3 shows the overhead of basic block counting for the OpenMP-parallelized versions of the same test programs from Figure 2 as well as for DC of the NPBs. The average overhead of basic block counting for multithreaded programs is 116% with PEBIL, 232% with Pin and 14743% with Dyninst. Furthermore, taking the difference in overhead between the OpenMP and MPI results as a rough measure of the marginal overhead required for ITLS support shows that PEBIL and Pin have significant but reasonable marginal overheads that average 99% and 178% of the runtime of the original program respectively, while the average marginal overhead of providing ITLS with Dyninst is 14725%. The large overhead with Dyninst is related to the expensive function call used by Dyninst to gain acccess to a thread's identifier, a function call that is executed at every basic block.

## B. *Case Study: Memory Trace Collection*

This section details a series of experiments that collect the full memory address traces of running programs, where the virtual addresses of 100% of all memory accesses made by the program are collected. Such collection forms the basis of many types of memory analysis applications – cache simulation [9][40], reuse distance calculation [41][42], access locality tracking [43], or compression and storage [44][45] for later analysis – and is therefore of wide interest within program analysis and hardware design. Rather than measure the effects of undertaking one of these specific memory analysis applications, all experiments that follow are used to collect and quickly discard the memory address trace of a program wherein the virtual addresses are calculated, inserted into per-process/per-thread buffer, then the buffer is immediately cleared once it is found to be full.

Figure 4 shows the overhead of collecting full memory address traces with PEBIL, Pin and Dyninst for MPI programs. The overheads found in this set of experiments are generally substantially higher than those found in the basic block counting experiments shown in Figure 2, which is not surprising given that memory address tracing must execute expensive effective address calculations at every memory access within the program as well as buffer maintenance code instead of

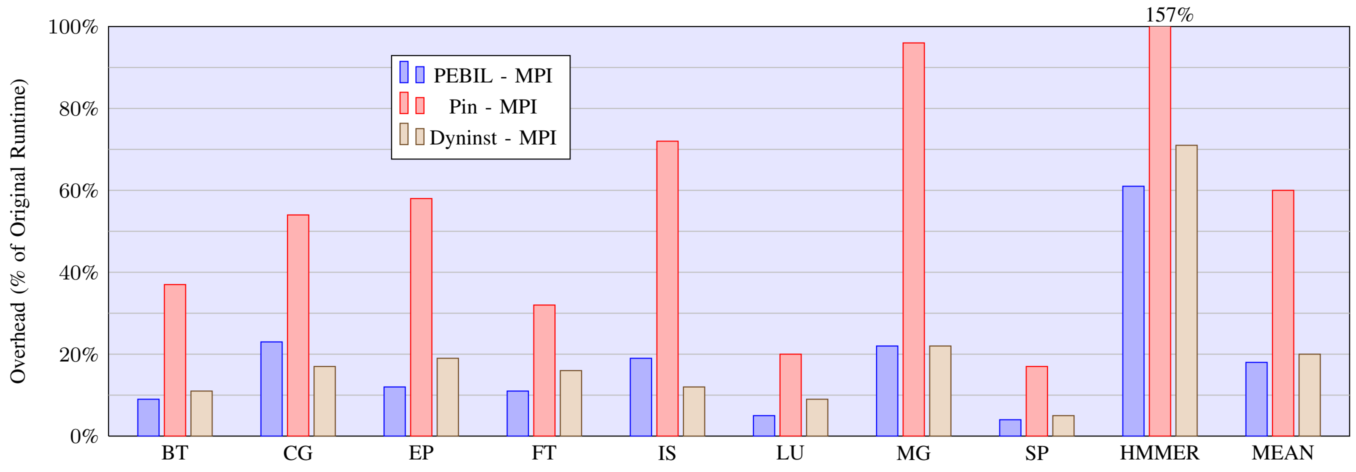


Fig. 2: Overhead of basic block counting with PEBIL, Pin and Dyninst for MPI-parallelized programs, expressed as a percentage of the runtime of the uninstrumented program.

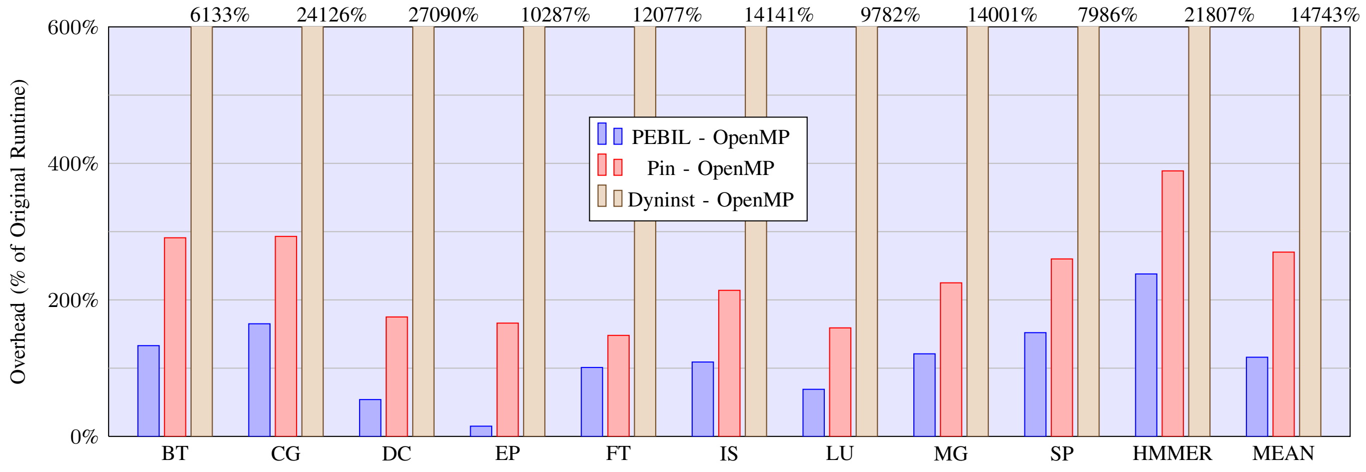


Fig. 3: Overhead of basic block counting with PEBIL, Pin and Dyninst for OpenMP-parallelized programs, expressed as a percentage of the runtime of the uninstrumented program.

simply incrementing a single counter at every basic block. For MPI programs the overheads for address trace collection with PEBIL, Pin and Dyninst are 445%, 343% and 1113% respectively. Note here that unlike basic block counting, Pin has lower overhead than PEBIL, both of which are substantially lower than Dyninst. Pin's improvement relative to the other platforms can be explained in terms of Pin's integration with the powerful post-link optimizer Ispike [31], which allows Pin to improve its performance relative to the other platforms as the frequency and complexity of the inserted instrumentation increases.

Figure 5 shows the overheads of collecting full memory address traces for OpenMP programs, again demonstrating that the overheads associated with providing support for multi-threaded programs are significant but reasonable for PEBIL and Pin, averaging 703% and 388% respectively. Using the difference between the OpenMP and MPI overhead for a particular program as a rough indicator of the marginal overhead caused by multithreading support shows that PEBIL averages an additional 302% overhead, Pin averages only 80% additional overhead and Dyninst averages an additional 79277% overhead. The likely explanation for Pin's low marginal overhead again has to do with Pin being able to optimize instrumented code very effectively relative to other platforms as the amount and complexity of the inserted code increases. Similar to basic block counting, Dyninst's high overhead is due to the frequency and high cost of the operation used by Dyninst to yield a thread identifier.

## V. Support for Fast Sampling

To support sampling, PEBIL provides facilities for switching instrumentation at any point between an active/inactive state. When inactive, the code at an inactive instrumentation point behaves as though it was not there – the original program runs but no side effects occur due to instrumentation. The instrumented program at that point behaves *as though no instrumentation had been inserted*. To facilitate switching between the active and inactive states, PEBIL prepares the

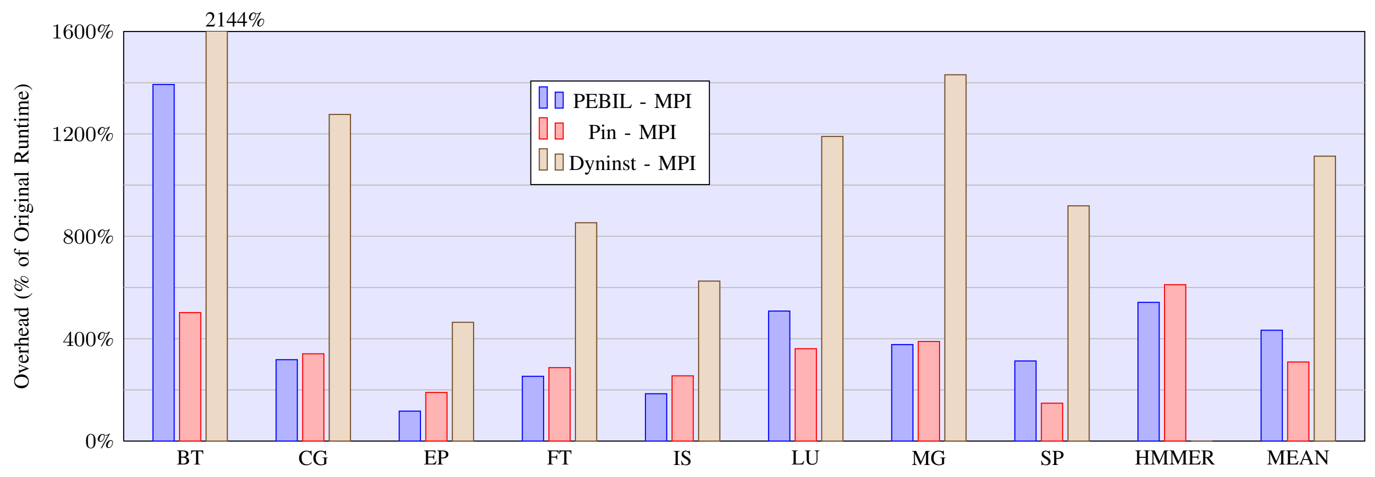


Fig. 4: Overhead of full memory address trace collection with PEBIL, Pin and Dyninst for MPI-parallelized programs, expressed as a percentage of the runtime of the uninstrumented program. Dyninst was unable to successfully instrument and run HMMER for memory address trace collection.

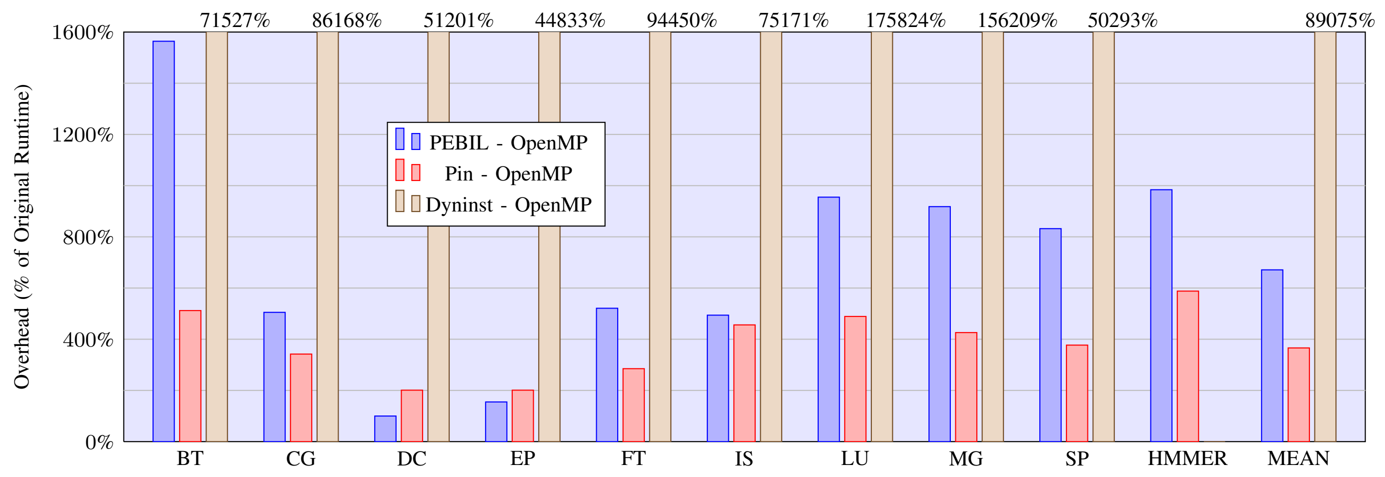


Fig. 5: Overhead of full memory address trace collection with PEBIL, Pin and Dyninst for OpenMP-parallelized programs, expressed as a percentage of the runtime of the uninstrumented program. Dyninst was unable to instrument HMMER for memory address trace collection.

executable by embedding into it information regarding the location and size of each of its instrumentation points. A runtime support library can then be used to specify how to change and set the state of each instrumentation point at any time during the instrumented program run. This runtime library activates and deactivates the code at an instrumentation point by relying on the information embedded into the instrumented binary by PEBIL, using that information to exchange `nop` instructions with the the text comprising the instrumentation code in order to deactivate and by swapping instrumentation code back in order to activate.

This technique differs markedly from runtime instrumentation manipulation as it is available in Pin and Dyninst, both of which are more full featured but more heavyweight. Pin provides support both for arbitrarily flushing all instrumentation for a program and for any part of the program to be instrumented at any time, thereby providing a super set of what PEBIL provides. When functioning as a dynamic binary instrumentation platform, Dyninst is similar. Instrumentation can be added to any part of the program at any time, while instrumentation can be removed at any time from a subset of instrumentation points. Pin and Dyninst also provide ways to detach themselves from a running program, which effectively disables all instrumentation and allows the program to proceed at its uninstrumented execution speed.

The simple deactivation technique used by PEBIL will decrease the runtime of the code around an instrumentation point in such a way that its performance will be close to, but not exactly, that of the uninstrumented program. This effect is demonstrated in Figures 6 and 7, which show a set of experiments on the MPI-parallelized and OpenMP-parallelized NPBs respectively, measuring the effect of running a program that is instrumented for basic block counting and, instead of counting basic block executions throughout the life of the program, immediately deactivates all points within the program and allows execution of the program to run its course. Though completely useless for learning about program behavior, the disabled basic block counter demonstrates the difference between running a program with active versus inactive instrumentation throughout its run. Across the MPI benchmarks in Figure 6, this introduces an overhead that averages 2.5% with a maximum of 4.6% while introducing overheads that average 2.8% (maximum 7.6%) across the OpenMP benchmarks in Figure 7. These results show that, in isolation, the performance impact of running with deactivated instrumentation is very close to that of the uninstrumented program's performance.

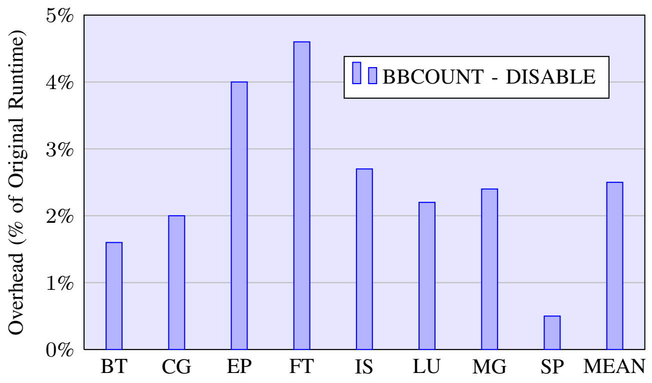


Fig. 6: Overhead for running PEBIL-instrumented MPI programs that contain disabled basic block counting instrumentation.

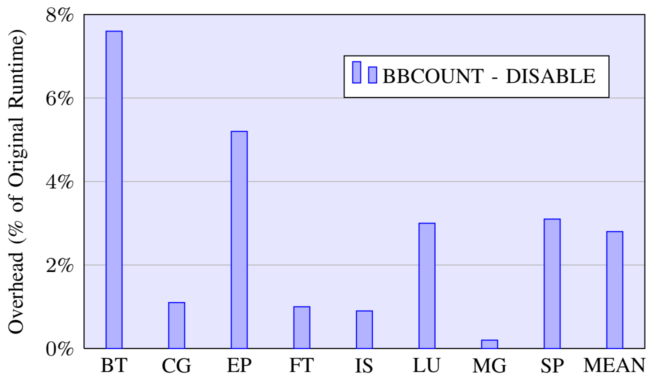


Fig. 7: Overhead for running PEBIL-instrumented OpenMP programs that contain disabled basic block counting instrumentation.

## A. *Deactivation for Sampling*

Despite the fact that Pin and Dyninst utilize more versatile and full-featured instrumentation manipulation schemes, these functional advantages come at the price of a relatively high runtime expense that can render them far less practical than PEBIL's simple activation/deactivation scheme when only simple activation/deactivation is needed. In such instances, activation/deactivation can be advantageous because it can be done far more cheaply than fully removing, re-generating then re-inserting the same code. Memory address trace collection is a good demonstration of this concept because it is often the case that sampling is utilized in address trace collection out of a desire to reduce the overhead of processing memory addresses, such as running the addresses through a cache simulator. Within a scheme in which instrumentation can be selectively activated and deactivated, it becomes possible to deactivate the code related to calculating effective memory addresses as well as the code copying those addresses to a buffer, effectively reducing the cost of *collecting* the sampled memory address trace as well. While a full treatment of the issue is out of the scope of this work, a brief tour of the address trace sampling literature indicates that very rudimentary sampling [21][22] can be used to discard at least 90% of a program's address trace while sophisticated techniques [46][47] can be used to discard upwards of 99% of an address stream while maintaining acceptable levels of accuracy.

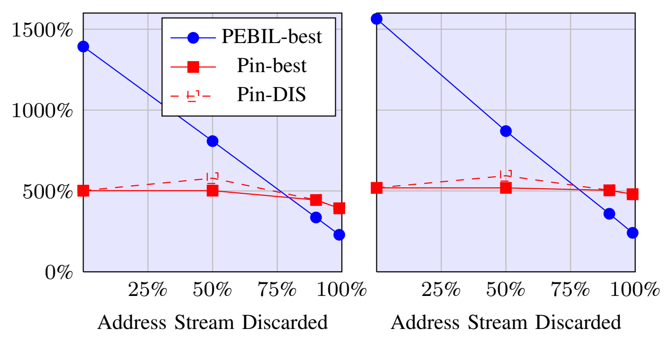
The experiments shown in Figures 8 through 16 reflect the overhead[2] of collecting a sampled[3] memory address trace using a sampling technique known as interval-based sampling that simply discards the last 50%, 90% and 99% of each interval of 1 billion memory references in the memory address stream. Each of Figures 8 through 16 are comprised of the results for a particular program which is parallelized with MPI in part (a) and OpenMP in part (b). The MPI and OpenMP results for a particular program are given on identical scales in order to allow for direct comparisons to be made between them. For example, Figure 8a shows the results for sampled memory address trace collection of the MPI implementation of BT and Figure 8b shows the overheads of sampled memory address trace collection for the OpenMP implementation of BT.

It is important to point out that 1 billion addresses is a fairly long sampling period, which, compared to shorter periods should benefit the platforms that have slower instrumentation removal/insertion mechanisms because a fewer absolute number of modifications to the state of instrumentation need to be made over the run of a particular program. In these experiments, PEBIL, Pin and Dyninst are employed on each of the test programs to collect the memory address stream subject to these various levels of sampling. When the address stream falls into or out of the sampling window, each of the platforms' specific instrumentation manipulation techniques are applied in order to disable/remove all instrumentation related to computing effective addresses and filling address buffers. Note that this does not mean disabling all instrumentation code – buffer maintenance code that counts the number of entries, checks for buffer fullness and resets the buffer counter is kept active during the entirety of a sampled run.

The overhead of sampled memory address stream collection with Pin under a removal/insertion regime is given by the series Pin-DIS, along with a second series Pin-best that is the minimum of the full trace overhead (given both by the overheads presented in Figures 4 and 5 and by the leftmost point, 0% of the address stream discarded, in Figures 8 through 16) and the Pin-DIS overhead at that particular sampling level. In cases for which removal/insertion does

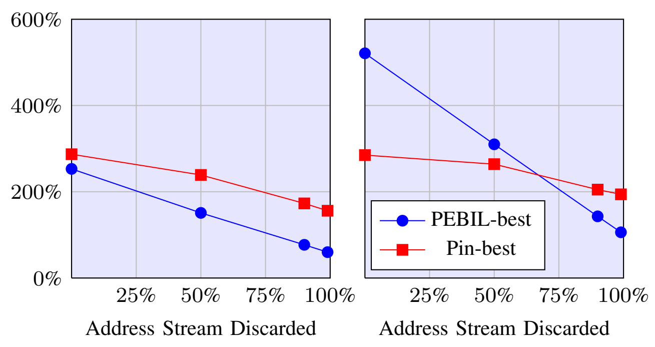
[2]In Figures 8 through 16 overhead is expressed as a percentage of the runtime of the uninstrumented program, given on the y-axis.

[3]In Figures 8 through 16 sampling rate is expressed as the percentage of the address stream that is discarded by sampling, given on the x-axis.



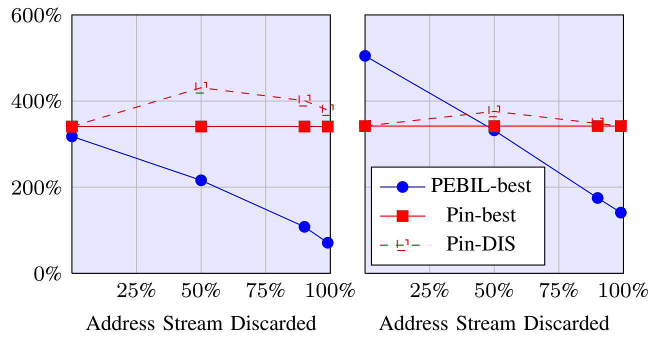
(a) BT/MPI (b) BT/OpenMP

Fig. 8: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for BT of the NPBs. Dyninst overheads are 2144% (MPI) and 71527% (OpenMP).



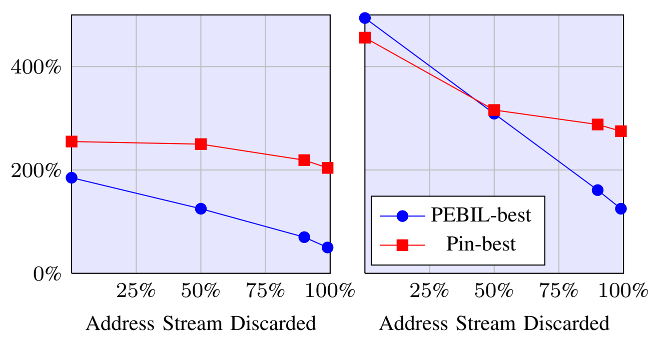
(a) FT/MPI (b) FT/OpenMP

Fig. 11: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for FT of the NPBs. Dyninst overheads are 853% (MPI) and 94450% (OpenMP).



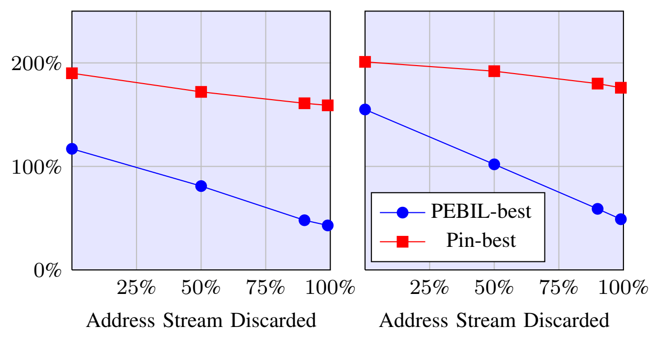
(a) CG/MPI (b) CG/OpenMP

Fig. 9: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for CG of the NPBs. Dyninst overheads are 1276% (MPI) and 86168% (OpenMP).



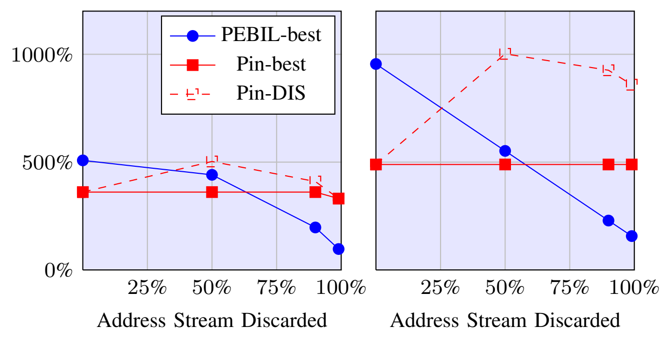
(a) IS/MPI (b) IS/OpenMP

Fig. 12: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for IS of the NPBs. Dyninst overheads are 625% (MPI) and 75171% (OpenMP).



(a) EP/MPI (b) EP/OpenMP

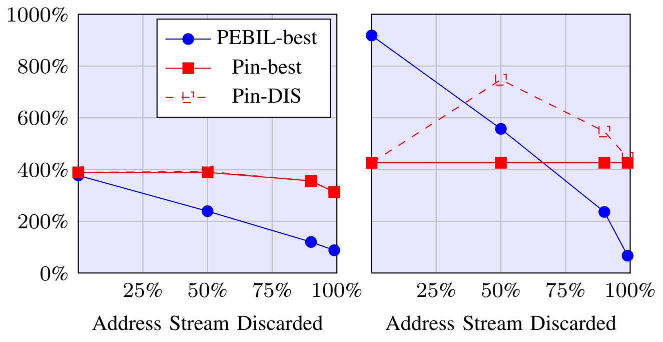
Fig. 10: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for EP of the NPBs. Dyninst overheads are 464% (MPI) and 44833% (OpenMP).



(a) LU/MPI (b) LU/OpenMP

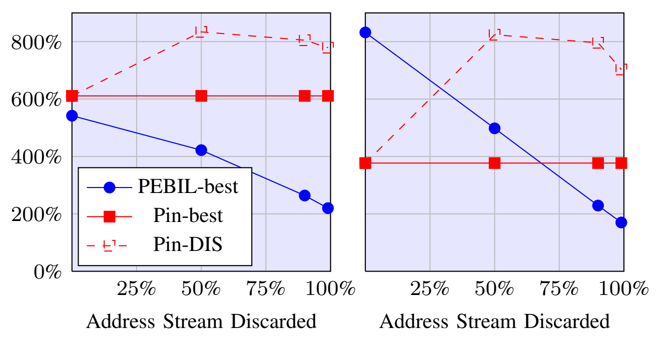
Fig. 13: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for LU of the NPBs. Dyninst overheads are 1190% (MPI) and 175824% (OpenMP).

not improve overhead, the optimal strategy is to *collect* the entire memory address trace while *processing* only the sampled portion. Since these experiments contain no processing, this minimum represents the best overhead that can be realized. Note also that in cases for which Pin-DIS and Pin-best are identical for all sampling levels, only Pin-best is included. With Pin the performance effects of removing and reinserting instrumentation are not straightforward, seemingly a function of both the sampling rate as well as the amount of work that must be done in order to remove and re-insert instrumentation, which has to do with the size and makeup of the affected program code. With Pin-best the average overheads are 343%, 335%, 313% and 295% for MPI programs and 388%, 371%, 360% and 355% for OpenMP programs when throwing away 0%, 50%, 90% and 99% of the address stream respectively. Even choosing the optimal method between Pin-DIS and full



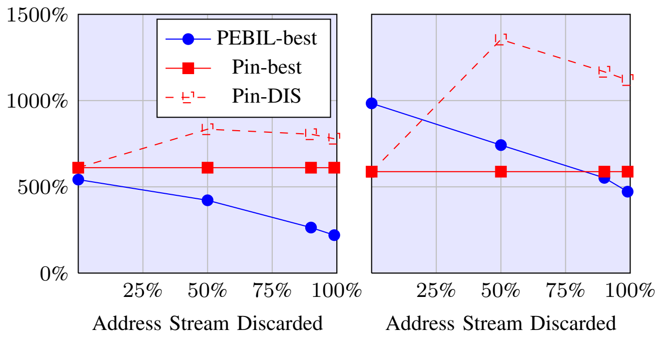
(a) MG/MPI (b) MG/OpenMP

Fig. 14: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for MG of the NPBs. Dyninst overheads are 1431% (MPI) and 156209% (OpenMP).



(a) SP/MPI (b) SP/OpenMP

Fig. 15: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for SP of the NPBs. Dyninst overheads are 919% (MPI) and 50293% (OpenMP).



(a) HMMER/MPI (b) HMMER/OpenMP

Fig. 16: Overhead (y-axis[2]) of sampled (x-axis[3]) memory address tracing with Pin and PEBIL for HMMER. Dyninst was unable to successfully instrument and run HMMER for memory address trace collection.

memory address tracing, on average Pin improves only slightly as larger amounts of the address stream are discarded due to sampling.

Dyninst never improves under a scheme of removing and reinserting instrumentation, implying that the best overhead that can be achieved for any sampling level with Dyninst is the overhead of full memory address trace collection shown in Figures 4 and 5. To facilitate comparison, the overhead for each benchmark is again given in the captions of Figures 8 through 16. These overheads average 1113% for MPI programs and 89075% for OpenMP programs respectively.

With sampling-related instrumentation deactivation, the overheads for PEBIL improve at all levels of sampling and for all programs, and the fastest result for a particular sampling level is the result achieved by deactivation rather than by simply ignoring the non-sampled parts of the address stream. These results for PEBIL given by the series PEBIL-best show that, usually, the improvement in overhead decreases as a linear function of the amount of the address stream that is discarded due to sampling. To state it simply, the more of the address stream one is willing to throw away, the lower the overhead that can be achieved. Overheads for sampled memory address trace collection with PEBIL average 445%, 296%, 144% and 101% for MPI programs and 703%, 435%, 222% and 171% for OpenMP programs when throwing away 0%, 50%, 90% and 99% of the address stream respectively. PEBIL has lower overhead than Dyninst for all sampling levels on both MPI and OpenMP programs. PEBIL also has lower overheads than Pin on average for all experiments, both for MPI and OpenMP programs, which discard 90% or more of the address stream and higher overheads on average on those that discard less than 90% of the address stream.

Worth noting is the fact that these experiments do not include processing overheads, so these results should not be taken to mean that sampling should be abandoned as a technique for reducing overhead in conjunction with any instrumentation platform. Independent of the specific instrumentation technology used, sampling is still likely to result in substantial reductions in overhead, particularly for heavy-weight processing functions of a memory address stream like reuse distance tracking or multiple cache simulation.

## VI. Conclusions

This work described two novel mechanisms within the PEBIL binary instrumentation platform that make it useful as a platform for writing tools that analyze data-intensive programs. The first is its built-in support for providing the location of instrumentation thread-local storage (ITLS) to a running thread by storing it in a process-wide table that each thread accesses through its unique id. Though a thread can use this mechanism to access its ITLS in as few as three x86_64 instructions, PEBIL goes further and attempts to cache the location of a thread's ITLS in a dead register in order to further reduce its overhead. Through a set of experiments in Section IV, this work measures the overhead of two binary instrumentation applications that are critical for the analysis of data-intensive programs (basic block counting and memory address trace collection) across a set of three binary instrumentation platforms (PEBIL, Pin and Dyninst) and two parallelization models (MPI and OpenMP). These experiments show that both PEBIL and Pin maintain reasonable overheads for both basic block counting and for memory address trace collection while delivering thread-specific results. For basic block counting PEBIL's overheads are shown to be lower than Pin's, at absolute levels for MPI and OpenMP programs as well as for the marginal overhead required to support delivering ITLS.

Because of Pin's capacity for optimizing heavily instrumented code well, for full memory address tracing Pin has lower overhead than PEBIL for MPI and OpenMP programs as well as for the marginal overhead of adding support for ITLS.

The second facility shown was PEBIL's support for switching individual instrumentation points into an active/inactive state by embedding instrumentation metadata into the instrumented binary then using that metadata to direct a runtime support library to quickly exchange `nop` instructions into and out of the program's text. This facility was evaluated as a strategy for supporting sampling of the information that is generated by a program. Section V shows a set of experiments that employ PEBIL, Pin and Dyninst to sample the memory address streams of a series of OpenMP and MPI programs at levels varying between 100% and 1%. These experiments show that, while functionally more limited than what is provided by Pin and Dyninst, PEBIL's facility for quickly switching instrumentation into active/inactive states is unique in that it allows runtime overhead to be reduced gracefully as larger fractions of the memory address stream are discarded due to sampling. Utilizing this mechanism, PEBIL is able to achieve overheads for 10%-sampled memory address trace collection that average 2.2x lower than Pin's and 7.7x lower than Dyninst's for MPI programs and 1.6x lower than Pin's and 401.2x lower than Dyninst's for OpenMP programs.

## Acknowledgements

The authors acknowledge the support of this project by the DoD HPCMP's User Productivity Enhancement, Technology Transfer, and Training (PETTT) Program (Contract No:GS04T09DBC0017 though High Performance Technologies, Inc.). This work was also supported in part by the U.S. Department of Energy Office of Science through the SciDAC award titled SUPER (Institute for Sustained Performance, Energy and Resilience).

## References

[1] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, 1997.

[2] A. Jaleel, R. Cohn, C. K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation*, pages 28–36, 2008.

[3] J. Pierce and T. Mudge. The effect of speculative execution on cache performance. In *Proceedings of the Eigth International Parallel Processing Symposium*, pages 172–179. IEEE, 1994.

[4] L. DeRose and F. Wolf. CATCH – A call-graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications. *European Conference on Parallel Processing*, pages 167–176, 2002.

[5] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, pages 17–30, 2005.

[6] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 74–83. ACM, 2008.

[7] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2):149–170, 2003.

[8] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for QoS in warehouse scale computers. In *Proceedings of the 10th Annual International Symposium on Code Generation and Optimization*, pages 1–12. ACM, 2012.

[9] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. pages 21–21. IEEE, 2002.

[10] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. *European Conference on Parallel Processing*, pages 79–90, 2011.

[11] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. Modeling power and energy usage of hpc kernels. In *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 990–998. IEEE, 2012.

[12] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[13] L. Carrington, M. M. Tikir, C. Olschanowsky, M. A. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the International Conference on Supercomputing*, pages 202–212. ACM, 2011.

[14] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 86–96. ACM, 2002.

[15] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

[16] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. *Parallel Processing Letters*, 11(02n03):267–280, 2001.

[17] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[18] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *International Symposium on Performance Analysis of Systems & Software*, pages 175–183. IEEE, 2010.

[19] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[20] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[21] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[22] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. *Computational Science*, pages 701–701, 2003.

[23] M. D. Smith. *Tracing with pixie*. Computer Systems Laboratory, Stanford University, 1991.

[24] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2):197–218, 1994.

[25] J. R. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *ACM Sigplan Notices*, volume 30, pages 291–300. ACM, 1995.

[26] A. Srivastava and A. Eustace. *ATOM: A system for building customized program analysis tools*, volume 29. ACM, 1994.

[27] M. M. Tikir, M. A. Laurenzano, L. Carrington, and Snavely A. The pmac binary instrumentation library for PowerPC/AIX. *Workshop on Binary Instrumentation and Applications*, 2006.

[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[29] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[30] M. A. Laurenzano, J. Peraza, L. Carrington, A. Tiwari, W. A. Ward, and R. Campbell. A static binary instrumentation threading model for fast memory trace collection. *International Workshop on Data-Intensive Scalable Computing Systems*, 2012.

[31] C. K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link optimizer for the Intel® Itanium® architecture. In *International Symposium on Code Generation and Optimization*, pages 15–26. IEEE, 2004.

[32] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks – summary and preliminary results. *The ACM/IEEE Conference on Supercomputing*, pages 158–165, 1991.

[33] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[34] P. P. Chang, S. A. Mahlke, and W. M. W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[35] D. W. Wall. *Predicting program behavior using real or estimated profiles*, volume 26. ACM, 1991.

[36] Y. T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, pages 456–461. ACM, 1995.

[37] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *IEEE International Workshop on Workload Characterization*, pages 149–156. IEEE, 2001.

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 45–57. ACM, 2002.

[39] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, pages 9–16. ACM, 2011.

[40] W. H. Wang and J. L. Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9(3):222–241, 1991.

[41] C. Ding and Y. Zhong. Reuse distance analysis. *University of Rochester, Rochester, NY*, 2001.

[42] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, pages 245–257. ACM, 2003.

[43] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 50. IEEE, 2005.

[44] A. Milenkovic and M. Milenkovic. Exploiting streams in instruction and data address trace compression. In *IEEE International Workshop on Workload Characterization*, pages 99–107. IEEE, 2003.

[45] C. Olschanowsky, M. M. Tikir, L. Carrington, and A. Snavely. PSnAP: accurate synthetic address streams through memory profiles. *Languages and Compilers for Parallel Computing*, pages 353–367, 2010.

[46] T. M. Conte, M. A. Hirsch, and W. M. W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, 1998.

[47] M. A. Laurenzano, B. Simon, A. Snavely, and M. Gunn. Low cost trace-driven memory simulation using SimPoint. *ACM SIGARCH Computer Architecture News*, 33(5):81–86, 2005.