A Static Binary Instrumentation Threading Model for Fast Memory Trace Collection

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 peformance, data intensive applications such as the real-time processing of surveillance feeds from unmanned aerial vehicles will 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 design decisions, memory traces gathered through binary instrumentation can be used to understand the low-level interactions between a data intensive code and the memory subsystem of a multi-core processor or many-core co-processor. Toward this end, this paper introduces the addition of threading support for PMaCs Efficient Binary Instrumentation Toolkit for Linux/x86 (PEBIL) and compares PEBILs threading model to the threading models of two other popular Linux/x86 binary instrumentation platforms – Pin and Dyninst – on both theoretical and empirical grounds. The empircal comparisons are based on experiments which collect memory address traces for the OpenMP-threaded implementations of the NASA Advanced Supercomputing Parallel Benchmarks (NPBs). This work shows that the overhead of collecing full memory address traces for multithreaded programs is higher in PEBIL (7.7x) than in Pin (4.7x), both of which are significantly lower than Dyninst (897x). This work also shows that PEBIL, uniquely, is able to take advantage of interval-based sampling of a memory address trace by rapidly disabling and re-enabling instrumentation at the transitions into and out of sampling periods in order to achieve significant decreases in the overhead of memory address trace collection. For collecting the memory address streams of each of the NPBs at a 10% sampling rate, PEBIL incurs an average slowdown of 2.9x compared to 4.4x with Pin and 897x with Dyninst.**

I. Introduction

Large data streams, such as those obtained through surveillance, typically require preprocessing before they can be used to determine frame-to-frame differences or track object motion. To acheive higher throughput, this preprocessing can often utilize highly parallel hardware such as the Xeon Phi coprocessor or some of the emerging multicore commodity systems of today. When designing such systems, memory performance is critical. Computer architects use memory address traces to develop and refine their memory subsystem designs in order to improve performance [1] and reduce energy consumption [2]. Software developers similarly use the characteristics derived from memory address trace analysis to restructure codes in order to achieve higher levels of performance.

With these factors as motivation, PMaC's Efficient Binary Instrumentation for Linux/x86 (PEBIL) [3] has recently been upgraded to support multithreaded x86_64 code. PEBIL is an open source binary instrumentation toolkit, which previously provided support for message passing codes only. PEBIL functions by inserting extra code and data into a compiled executable, *statically* creating an instrumented executable binary. When this instrumented binary executes, the original functionality of the program is retained while information about the program's behavior (e.g., its memory address stream) is collected as a side-effect.

This work introduces PEBIL's support for the instrumentation of codes that are multithreaded using pthreads or OpenMP. PEBIL's threading model is explained (Section II) and compared to two other popular x86_64/Linux binary instrumentation toolkits, Pin [4] and Dyninst [5], with particular attention given to each of the tools' support for providing thread-specific program analysis (Section III). The threading support overhead for PEBIL, Pin and Dyninst are then compared empirically by timing the capture of memory address traces for the OpenMP-threaded implementations of the NASA Advanced Supercomputing Parallel Benchmarks (NPBs) [6] using interval-based sampling rates ranging from 1% to 100% (Section IV). These experiments show that full memory address trace collection (a 100% sampling rate) slows the threaded NPBs by an average of 7.7x for PEBIL, 4.7x for Pin and 897x for Dyninst. Furthermore, PEBIL is solely able to take advantage of interval-based sampling to significantly reduce overheads – at a 10% sampling rate the overhead of memory address trace collection is reduced to 2.9x for PEBIL, 5.5x for Pin and is unimproved over full tracing for Dyninst.

II. PEBIL's Threading Model

Several issues must be addressed when designing and implementing support for the instrumentation of multithreaded codes. These are divided into two broad categories: those required to produce correct, thread-safe instrumented code and those required to deliver thread-specific results from instrumented code.

A. Instrumenting for Thread Safety

PEBIL's generation and insertion of code into a target program have two important purposes: (1) to add functionality to the program, generally to collect some type of information about what the running program is doing, and (2) to protect the program's state from being altered by that extra functionality. The central mechanism introduced when integrating support for multithreaded codes into PEBIL is the utilization of *position independent code* for any instrumentation code serving either of these purposes. In many cases, position dependent code can be translated directly into position independent code simply by using indirect or PC-relative addressing. In those cases, the performance penalty is quite low since these addressing modes require only a small amount of extra computation relative to their position dependent counterparts.

Certain operations, however, have no immediate translation when replacing position dependent code with position independent code. For instance, PEBIL must preserve the program's semantics when inserting extra code. Therefore, for any inserted instrumentation code which defines some register that is live in the program PEBIL must save that register prior to running any inserted code then restore it to its original value before returning control back to the program. As an efficient way of accomplishing state preservation, previous versions of PEBIL used a single fixed memory region as the storage site for spilled program state. For multithreaded codes, however, this simple technique is inadequate because more than one thread cannot safely access the same memory region absent some mechanism to prevent basic concurrency errors. Such mechanisms are generally very slow relative to the typical operations performed by inserted instrumentation code and hence are undesirable because they would significantly increase the runtime overhead of the instrumented program. To preserve and restore a multithreaded program's state safely and efficiently, PEBIL now saves and restores state to the top of each thread's private execution stack.

B. Thread-Local Instrumentation Data

To provide thread-specific statistics about program behavior, it is necessary for PEBIL to provide a mechanism that allows an executing thread to quickly find its private copy of instrumentation data structures (e.g., counters, buffers, or other collected program state). The difference between maintaining a private copy of a data structure and maintaining synchronization around a single data structure shared between all threads is important because performance data gathered using the former approach can often be far more useful than the latter – for example, thread-local execution counters can reveal imbalances between threads. Whether data structures are thread-local can also have dramatic consequences in terms of the performance of the instrumented program. For example, a basic optimization for handling the memory address stream generated by an execting program is to buffer those addresses and process them in batch. A tool which attempted to share a single buffer between all threads could either synchronize access to that buffer either at a very fine granularity, which would invariably generate large amounts of coherence-related bus traffic, or synchronize access at a coarse granularity, allowing only a single thread to make progress at a time.

PEBIL provides thread-local data structures to an instrumented multithreaded program by providing a hook to thread creation that allows thread-local data structures to be generated for private use by each newly created thread. 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, for all but the simplest instrumentation tools this memory pool is therefore limited to holding the addresses of other interesting data structures. When collecting memory access traces, for example, this pool contains the address of a buffer which 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 and introduces an optimization that allows the location of that memory pool to be cached for short periods of time within a running program.

1) Runtime Thread-Local Data Access: 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. For a given thread, the default formula for deriving the thread's index into this table `IDX` is `IDX = (TID >> 12) & 0xffff`, where `TID` is the unique identifier for the thread. This formula yields a value for `IDX` within the range `[0,65536)`, which is simply the value of bits 12-27 of the thread's unique identifier. From the standpoint of efficiency, this method is perfect since it can generate `IDX` from scratch[1] in as few as three x86_64 instructions.

This method will generate identical indexes for any threads that share bits 12-27 in their unique identifiers and in principle there is no guarantee of uniqueness of these bits. In practice, however, conflicts of this sort have never been encountered when running up to 16 threads per process. To detect conflicts as such, PEBIL currently intercepts all thread create calls, verifying for each new thread that no existing thread has a table index that conflicts with the new thread's table index. If a conflict is detected, PEBIL generates a runtime error rather than falling back to a slower mode which can resolve conflicts or guarantee that they will not occur. In such cases execution can be retried, allowing PEBIL to use a larger number of bits of the thread ID to reduce the likelihood of conflict, though it should be noted that each additional bit used doubles the size

[1] In x86_64 a running thread's unique identifier is stored in `%fs:0x10`.

of the table of memory pools.

2) Caching Thread-local Data: Even though the instruction sequence generating the location of a thread's private memory pool is short, that sequence of instructions potentially must be executed at every instrumentation point that refers to the memory pool. Because detailed instrumentation tools typically require frequent access to the memory pool, sometimes as often as every basic block or every memory instruction, instead of requiring the location of the memory pool to be recomputed every time a thread executing instrumentation code 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 subsequent instrumentation point. 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 which recomputes the memory pool location 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, immeditately following a call to another function) points of the function. In order to increase its effecicy, future work within PEBIL should include extending this optimization to smaller code structures like loops or basic blocks.

III. Related Work

There are many other x86/Linux binary instrumentation projects: Pin [4], Dyninst [5], and Valgrind [7] being among the most popular. This section focuses on Pin and Dyninst due to space limitations and because Valgrind was designed for certain useful but heavyweight instrumentation tasks, distinguishing itself in terms of functionality at the cost of efficiency [8]. Many binary instrumentation tools covering other architectures and operating systems exist, but are outside the scope of this work.

Pin is a popular dynamic binary instrumentation tool which is maintained by Intel for use on its x86 and x86_64 platforms; it supports threads by providing API hooks for thread creation/desctruction and for associating a number of data structures with each thread. Whereas PEBIL performs all state preservation on a thread's private stack, Pin sets aside a distinct region of heap memory for each thread. Pin-instrumented multithreaded code accesses this region of memory by storing its location at all times in some general purpose register that is stolen from the program, which is very effective at minimizing the overhead of accessing that thread-local data. This register stealing approach is possible because Pin utilizes a sophisticated dynamic code optimization engine to reorganize the program around the stolen register where necessary.

Dyninst is a dynamic binary instrumentation tool and static rewriter which supports a variety of platforms, including x86 and x86_64. Dyninst supports thread-awareness in the development of instrumentation tools by by providing a class that allows the control and examination of running threads. Dyninst also provides the facilities for building limited expressions to implement hand-coded instrumentation code sequences, which can be written to utilize the identifier of an executing thread. The mechanisms used to control and interact with threads at runtime and the facilities through which hand-coded instrumentation is expressed are richer than what PEBIL provides, but they are far more heavyweight. The 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 the resulting necessary state protection. 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 the following section, these factors combine to introduce very large overheads when utilizing thread-local data in Dyninst.

IV. Experimental Results

This section presents the results of an empirical study whose purpose is to show the overhead of gathering memory address traces for multithreaded codes with PEBIL, Pin and Dyninst. These experiments use aggressively optimized instrumentation tools developed for each instrumentation platform which collect memory address traces for the OpenMP implementations of the NAS Parallel Benchmarks (NPBs) [6]. Descriptions of these benchmarks are given in Table I. The test system used in all experiments is a dual-socket, 8-core Intel Xeon X3450. Each core 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 memory. Each experiment is run using a single process with a number of OpenMP threads specified in Table I. Furthermore, all results presented here are computed as the mean of three independent runs of the particular experiment.

TABLE I
NAS Parallel Benchmark Descriptions

Name	Description	Input Size	Thread Count	# of Static Mem. Insns.
BT	block tri-diagonal solver	B	4	4888
CG	conjugate gradient	B	8	1170
DC	data cube	W	8	2240
EP	embarassingly parallel	B	8	573
FT	3D fast Fourier Transform	B	8	1889
IS	integer sort	C	8	372
LU	lower-upper Gauss-Seidel	B	8	5339
MG	multi-grid on mesh sequence	B	8	2568
SP	scalar penta-diagonal solver	B	4	4456

A. Thread Support Overhead

The first experiment is intended to illustrate the overhead of frequently accessing thread-specific data in the multithreaded workload described in Table I. PEBIL, Pin and Dyninst are used to instrument each program in this workload in order to fill a set of buffers (one for each thread) with every memory access issued during program execution. That is, these experiments are designed to collect the full memory address stream that results from program execution, also known as a

full memory address trace. The per-thread buffer size for each tool was optimized for speed using a small set of empirical tests, resulting in 32KB buffers for PEBIL, 128KB buffers for Pin and 512KB buffers for Dyninst. Because this experiment is intended to quantify the overhead of providing thread-specific instrumentation as opposed to demonstrating the use of or overhead related to a particular application of memory address traces, upon encountering a full buffer the memory addresses found in that buffers are simply discarded as quickly as possible in order to return control to the instrumented program to begin filling the buffer again. The sequence of interactions that a thread has with its private buffer are to:

1) Fill the buffer with memory addresses.
2) Call a minimal processing function which updates a count of the number of memory accesses encountered so far and marks the buffer as being empty.
3) Return control to the instrumented application which begins to fill the buffer again.

The results for this experiment are given in the top row of Table II labeled "Full Trace"; the entries in this table are the slowdowns involved in collecting the full memory address traces for each of the test benchmarks using PEBIL, Pin and Dyninst, computed as the instrumented application runtime divided by the uninstrumented runtime. Examination of these results matches our expectations fairly closely – the average slowdowns for PEBIL, Pin and Dyninst are 7.7x, 4.7x and 897x respectively. PEBIL is on average 1.7x slower than Pin when producing full address traces, which is a reasonable value given the relatively long runtimes of collecting the traces. The high overheads associated with Dyninst are due principally to the fact that the thread identifier is to give a thread access to its private address buffer, which is done at every memory instruction; the overhead of this operation is high in Dyninst and is the result is never reused between instrumentation points, so it must be done *very* frequently. This contrasts with both PEBIL and Pin. In PEBIL, accessing the thread's private buffer is much faster because accessing the thread identifier is faster and because that identifier might be cached and reused between instrumentation points. In Pin, the location of a thread's private data, which in turn contains the thread's private address buffer, is accessed through a register that stolen from the program. This potentially introduces overhead in the form of a program that is reorganized to not use that register, but results in very fast access to the thread's private address buffer since most of the computation of the buffer's address only needs to be computed *once*.

B. Sampling the Memory Address Stream

It is often the case that full memory address streams are subjected to some form of sampling, either in the form of interval-based sampling [1] or resulting from a sohpisticated phase-based analysis scheme [9]. The motiviation for introducing sampling of any form is simple – HPC application runtimes are often measured in hours and utilizing their full memory address traces consumes too many resources (time and/or disk space). Sampling is popular because it often allows significant fractions of the memory address stream to be discarded while still allowing the properties of the memory address stream to be ascertained with an acceptable level of fidelity.

Throwing away memory addresses after they are collected allows the overhead of *processing* memory addresses to be reduced, yet it also introduces the opportunity to reduce the overhead of *collecting* those memory addresses. If the instrumentation tool can disable or remove instrumentation during the instrumented program run, then interval-based sampling can potentially reduce the cost of collecting the memory address trace because large amounts of unnecessary computation (computing effective addresses and copying addresses as well as other metadata into the address buffer) can be avoided. Each of PEBIL, Pin and Dyninst are capable of modifying instrumentation at runtime in this fashion. PEBIL can rapidly disable and re-enable arbitrary instrumentation points at runtime using a simple operation which swaps inserted instrumentation code for nops. Pin and Dyninst can remove and arbitrarily reinstrument code, which is far more versatile than PEBIL's approach but which comes at the expense of runtime overhead to perform the removal and re-instrumentation operations.

The effectiveness of interval-based sampling within PEBIL, Pin and Dyninst are explored by repeating the experiments in Section IV-A but capturing only the first 10% and 1% of the memory addresses of each interval of 1 billion addresses issued by the running program. During these runs, the instrumentation tool disables and reenables instrumentation around the sampled regions of the program's memory address stream. The results of introducing sampling are seen in Table II, which show the overhead for three sampling rates for both PEBIL and Pin, labeled PEBIL-SAMPLE and Pin-SAMPLE respectively. Dyninst-SAMPLE is excluded from Table II because reconfiguring instrumentation with Dyninst in this fashion always resulted in increased overheads. Because introducing this optimization into these Pin experiments does not reliably result in overhead improvements, a results series labeled Pin-BEST is also shown which gives the best achievable overhead between collecting the full memory address trace (Pin-FULL) and the sampling-aware Pin tool used at that sampling rate (Pin-SAMPLE). These results show that the overhead of collecting memory accesses with PEBIL decreases gracefully as the sampling rate decreases. Pin does not demonstrate the same reliable performance improvements despite the very large sampling period[2] used here – 1 billion addresses; on average Pin-FULL performs better than Pin-SAMPLE, even for the lowest sampling rate of 1%.

Overall, PEBIL-SAMPLE is 1.9 times faster than Pin-SAMPLE at a 10% sampling rate and over twice as fast at 1%. PEBIL-SAMPLE is generally far better than PEBIL-FULL because PEBIL's method of reconfiguring instrumentation at runtime is very lightweight, involving a simple exchange of instrumentation code and nops, resulting in runtime overheads

[2]A larger sampling period implies that fewer instrumentation removal and re-insertion operations will be done per instrumented program run.

TABLE II
MEMORY ADDRESS TRACE COLLECTION OVERHEAD (SLOWDOWN RELATIVE TO UNINSTRUMENTED RUNTIME) FOR THE NBPS (OPENMP)

		BT	CG	DC	EP	FT	IS	LU	MG	SP	MEAN
Full Trace	PEBIL-FULL	16.6	6.1	2.0	2.6	6.2	5.9	10.6	10.2	9.3	7.7
	Pin-FULL	6.2	4.4	3.0	3.0	3.9	5.6	5.9	5.3	4.8	4.7
	Dyninst-FULL	739	863	531	449	922	752	1759	1556	504	897
10% Sampled	PEBIL-SAMPLE	4.6	2.8	1.8	1.6	2.4	2.6	3.3	3.4	3.3	2.9
	Pin-SAMPLE	6.0	4.5	3.6	2.8	3.1	3.9	10.3	6.5	9.0	5.5
	Pin-BEST	6.0	4.4	3.0	2.8	3.1	3.9	5.9	5.3	4.8	4.4
1% Sampled	PEBIL-SAMPLE	3.4	2.4	1.8	1.5	2.1	2.3	2.6	2.7	2.7	2.4
	Pin-SAMPLE	5.8	4.4	4.5	2.8	2.9	3.8	9.6	5.5	8.0	5.3
	Pin-BEST	5.8	4.4	3.0	2.8	2.9	3.8	5.9	5.3	4.8	4.3

in PEBIL-SAMPLE that reliably decrease as more of the program's address stream is discarded during sampling. The difference between Pin-FULL and Pin-SAMPLE is the confluence of two competing effects – (1) increased overhead due to the acts of removing and re-inserting instrumentation around sampling and (2) reduced overhead as a result of running code which is not wasting time computing addresses and copying those addresses into a buffer. Which of these effects is stronger for a particular experiment depends on the sampling rate as well as the amount of code that is reinstrumented each time sampling is removed or re-inserted. The latter is too complex to be treated in this work in great detail, though it is related to the static size of the code and number of memory operations in the program. Indeed if the number of memory operations (see Table I) in a benchmark is taken as a guide, it can be observed that programs with the largest numbers of memory instructions (LU and SP) show the worst performance declines in Pin-SAMPLE over Pin-FULL whereas the program with the smallest number of memory instructions (IS) shows the most improvement in Pin-SAMPLE of Pin-FULL.

Finally, it is important to note that these results in no way cast doubt on the generally-realizable benefits of introducing interval-based sampling into *any* scheme which utilizes a program's memory address trace because sampling will still have the effect of proportionally reducing the computation or storage required to handle the address trace. That is, memory address trace *collection* overhead can be improved in ways that depend on the nature of the tool being used to collect the address trace, but sampling can be used to improve memory address trace *processing* overheads in conjunction with any binary instrumentation tool.

V. CONCLUSIONS

HPC software will continue to evolve and transform to utilize the high levels of concurrency offered by current and upcoming multicore and manycore chips. This evolution has used (and will continue to use) complex models of parallelization to include both interprocess and shared memory models built on top of threading platforms like OpenMP and pthreads. Sophisticated program analysis tools, such as the binary instrumentation toolkit PEBIL, are necessary for understanding how to effectively use increasingly complex combinations of hardware and supporting system software. This work presented the extensions made to PEBIL in order to provide support for instrumenting programs that are multithreaded with pthreads or OpenMP. Details of PEBIL's threading model, along with some optimizations surrounding that model, were described and compared to the models of two other popular binary instrumentation tools – Pin and Dyninst. A series of experiments surrounding the collection of programs' memory address traces using each of these three instrumentation tools were performed, demonstrating that Pin is somewhat faster than PEBIL for collecting full memory address traces for the OpenMP implementations of the NAS Parallel benchmarks, with Dyninst far slower than either. Beyond this, the introduction of interval-based sampling reduces the overheads observed for collecting memory address traces with PEBIL in proportion to the sampling rate, increases the overhead with Dyninst, and has varying results with Pin. These experiments show that PEBIL has a reasonable threading model and is well-positioned for performing practical analysis of the memory address streams of HPC programs.

REFERENCES

[1] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. *International Conference on Computational Science*, 2003.

[2] 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*, 2011.

[3] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: efficient static binary instrumentation for Linux. *International Symposium on Analysis of Systems & Software*, 2010.

[4] 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. *Programming Language Design and Implementation*, 2005.

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

[6] 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 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1991.

[7] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Programming Language Design and Implementation*, 2007.

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

[9] M. A. Laurenzano, B. Simon, A. Snavely, and M. Gunn. Low cost trace-driven memory simulation using simpoint. *Workshop on Binary Instrumentation and Applications*, 2005.