

A Static Binary Instrumentation Threading Model for Fast Memory Trace Collection

Michael A. Laurenzano* Joshua Peraza† Laura Carrington* Ananta Tiwari* William Ward‡ 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—As hardware vendors push for higher levels of concurrency in multicore and manycore chips, the HPC software running on that hardware must increasingly utilize sophisticated models of parallelization, including interprocess message passing via MPI or SHMEM, intraprocess data sharing via threading models such as pthreads and OpenMP, and combinations of the two approaches. With this increase it is important for tools supporting HPC research activities to include support for all available models of parallelism. In this work we introduce a threading model recently integrated into PEBIL, an open source static binary instrumentation framework for x86/Linux focused on producing efficient instrumented HPC code. Previous versions of PEBIL support instrumenting only programs utilizing message passing models of parallelism; these upgrades allow PEBIL to support shared memory models of parallelism implemented with pthreads and OpenMP.

Binary instrumentation proves particularly useful in gaining insights into the the behavior of data-driven HPC programs via their interactions with machines’ memory subsystems. As such, we compare the principles of PEBIL’s threading model to two other popular x86/Linux binary instrumentation platforms, Pin and DyninstAPI. We then compare PEBIL’s threading model to Pin’s empirically using using a series of experiments centered around memory address trace collection for the OpenMP implementations of the NAS Parallel Benchmarks. Through these experiments we show that the raw overhead of PEBIL’s threading support and address stream generation is higher than Pin’s, but that PEBIL is able to take advantage of sampling techniques to decrease the overhead of address stream collection while sampling with Pin is sensible only for programs with small code footprints due to the overhead involved in repeatedly regenerating instrumentation for each memory operation.

I. INTRODUCTION

HPC programs rely increasingly on complex parallelization techniques, including message passing via MPI or SHMEM, threading models based on pthreads or OpenMP, or combinations of the two. It is critical that analysis tools remain able to handle the variety of models of parallelism that appear in HPC programs. Toward this end, PEBIL[1] has recently added support for handling multithreaded x86_64 code in addition to its existing support for codes parallelized using message

passing models. PEBIL is an open source binary instrumentation toolkit, which inserts instrumentation by rewriting a program executable *statically*. That is, PEBIL instruments a program by inserting extra code and data into a compiled executable, resulting in a runnable instrumented binary being written to disk. This instrumented binary collects information about the original program’s behavior as runs, retaining all of the original functionality of the program. A major goal of PEBIL is to produce instrumented HPC programs which capture this program information while introducing as small a runtime overhead as possible.

This work introduces the recently added support within PEBIL for instrumenting codes that are multithreaded with either pthreads or OpenMP along with several other novel optimizations that uniquely suit PEBIL’s static instrumentation model. PEBIL’s threading model is explored and compared to two other popular x86/Linux binary instrumentation platforms – Pin[2] and DyninstAPI[3]. We then go on to compare PEBIL to Pin experimentally in the context of capturing memory address traces under several sampling scenarios for the OpenMP-multithreaded implementations of the NAS Parallel Benchmarks[4]. Collecting a memory address trace is a useful way of stressing PEBIL’s thread support facilities, but is also important because gaining insights into the memory behavior of programs has been shown to be useful in a wide range of research applications. Computer architects use memory address traces to develop and refine memory subsystem designs. Program behavior in terms of performance and correctness[5] often depends heavily on memory behavior, making memory address traces a useful tool for dealing with those aspects of compiler and program development. Models of performance[6] and energy[7] can also depend on the ability to understand how programs utilize a memory subsystem.

The remainder of this paper is organized as follows. Section II discusses some of the more important design decisions encountered when adding support for multithreaded programs to PEBIL. Section III provides a discussion of some other

Report Documentation Page		Form Approved OMB No. 0704-0188
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.		
1. REPORT DATE 16 NOV 2012	2. REPORT TYPE	3. DATES COVERED 00-00-2012 to 00-00-2012
4. TITLE AND SUBTITLE A Static Binary Instrumentation Threading Model for Fast Memory Trace Collection		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER	
	5e. TASK NUMBER	
	5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California, San Diego, Performance Modeling and Characterization Laboratory, San Diego Supercomputer Center, La Jolla, CA, 92093		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		
13. SUPPLEMENTARY NOTES Presented at the International Workshop on Data-Intensive Scalable Computing Systems (DISCS), Salt Lake city, UT, 16 Nov 2012. U.S. Government or Federal Rights License		
14. ABSTRACT As hardware vendors push for higher levels of concurrency in multicore and manycore chips, the HPC software running on that hardware must increasingly utilize sophisticated models of parallelization, including interprocess message passing via MPI or SHMEM, intraprocess data sharing via threading models such as pthreads and OpenMP, and combinations of the two approaches. With this increase it is important for tools supporting HPC research activities to include support for all available models of parallelism. In this work we introduce a threading model recently integrated into PEBIL, an open source static binary instrumentation framework for x86/Linux focused on producing efficient instrumented HPC code. Previous versions of PEBIL support instrumenting only programs utilizing message passing models of parallelism; these upgrades allow PEBIL to support shared memory models of parallelism implemented with pthreads and OpenMP. Binary instrumentation proves particularly useful in gaining insights into the the behavior of data-driven HPC programs via their interactions with machines? memory subsystems. As such we compare the principles of PEBIL?s threading model to two other popular x86/Linux binary instrumentation platforms, Pin and DyninstAPI. We then compare PEBIL?s threading model to Pin?s empirically using using a series of experiments centered around memory address trace collection for the OpenMP implementations of the NAS Parallel Benchmarks. Through these experiments we show that the raw overhead of PEBIL?s threading support and address stream generation is higher than Pin?s, but that PEBIL is able to take advantage of sampling techniques to decrease the overhead of address stream collection while sampling with Pin is sensible only for programs with small code footprints due to the overhead involved in repeatedly regenerating instrumentation for each memory operation.		

15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report (SAR)	18. NUMBER OF PAGES 5	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

binary instrumentation platforms, their threading models, and how those models compare to PEBIL’s. Section IV describes a set of experiments which compare three binary instrumentation tools – PEBIL, Pin, and DyninstAPI – in terms of efficiency for capturing memory address streams for threaded codes under a variety of conditions. Finally, Section V concludes.

II. PEBIL’S THREADING MODEL

As a static binary instrumentation tool, PEBIL must decide prior to runtime whether to instrument a program for multithreading support. PEBIL provides support for detecting the presense of thread-related constructs in the program but ultimately lets user decides whether to support multithreading when instrumenting a given program with a given instrumentation tool. To aid this effort, PEBIL provides a command line flag `--threaded` to allow the user to indicate when they are instrumenting a program with support for threading. In cases where the indication of this flag mismatches PEBIL’s expectation of whether a program will use threads at runtime (e.g., the program contains a call to `pthread_create` but is not instrumented using the `--threaded` flag), a warning is generated indicating that mismatch to the user.

There are many considerations that must be made when designing and implementing support for the instrumentation of multithreaded codes. We divide these considerations into two broad categories – those which are required to produce correct, thread-safe instrumented code and those that are required to deliver thread-specific results from instrumented code.

A. Instrumenting for Thread Safety

PEBIL generates and inserts code into the program which has two principle functions: (1) to add functionality to a program, functionality which is usually focused on collecting some type of information about what the running program is doing, and (2) to protect the program’s state from being destroyed by that extra functionality. The central consideration when integrating support for multithreaded codes into PEBIL is to utilize *position independent code* (PIC) for any such instrumentation code. In many cases, position dependent code (PDC) can be translated directly into PIC by simply using indirect or PC-relative calculations. In these cases, PIC is rarely faster but also rarely much slower than the corresponding PDC because using indirect and PC-relative addressing adds only a small extra computational burden.

Certain operations, however, have no immediate translation when switching from PDC to PIC. Since PEBIL must leave the original functioning of a program intact around extra inserted code, at each instrumentation point PEBIL may have to save any registers that are defined by the instrumentation point and restore them prior to returning control back to the program. As a convenient and efficient way of accomplishing state preservation, previous versions of PEBIL used a single fixed memory region in the program as the storage site for spilled program state. However, this simple technique is insufficient in the presence of multiple threads because more than one thread cannot safely share the same region of memory without also

using some mechanism to prevent basic concurrency errors. Such synchronization mechanisms are very slow relative to the typical operations being performed by instrumentation code, making them untenable because they conflict with PEBIL’s goal of generating efficient instrumented code. In order to safely and efficiently save and restore state around instrumentation in the presence of multiple threads, any necessary state preservation operations are done on top of each thread’s private execution stack. We now discuss two optimizations crafted onto this basic state preservation model.

1) *The flags Register:* In the x86_64 architecture, a variety of information about the status of the processor is held in the `flags` register by way of a series of predefined status bits contained within it. Since this register contains information relevant to the operation of the program, it potentially needs to be saved and restored around each instrumentation point. PEBIL provides two levels of protection for the `flags` register, a faster method which protects only its lower 8 bits via the `sahf/lahf` instructions and a slower method which protects a larger set of bits via the `pushf/popf` instructions. To select the fastest available `flags` protection mechanism, or no protection mechanism at all, PEBIL detects which `flags` bits are live at a given instrumentation point and uses the fastest method that provides protection to all live bits which can be defined by the code at the instrumentation point.

PEBIL goes beyond this with its unique way of handling of the `overflow` bit of the `flags` register. This treatment stems from the following observations:

- `overflow` is one of a set of bits – `carry`, `parity`, `adjust`, `zero`, `sign` and `overflow` – which are the only `flags` bits defined by hundreds of (mostly arithmetic) instructions in the x86_64 ISA.
- It is the only member of that set of bits outside of the lower 8 bits of `flags`, requiring it to be handled by the relatively slow `pushf/popf` protection mechanism.
- It is often obvious to the user that a sequence of code will not produce an overflow.

Using these observations PEBIL exposes a knob allowing the user to tell PEBIL that the code within a certain instrumentation point will not overflow despite the fact that the instructions comprising that code can theoretically generate overflows. For example, a common operation for instrumentation code is to increment a 64-bit counter by 1. At modern clock rates, it would take hundreds of years¹ to overflow such a counter even if it were executed once per cycle. By utilizing the user’s guarantee of a non-overflow, PEBIL is more likely to find points in the program where it can use the more efficient `sahf/lahf` mechanisms instead of `pushf/popf` when protecting the `flags` register.

2) *Handling Leaf Functions:* According to the x86_64 ABI, leaf functions can utilize up to 128 bytes above the stack, the so-called red zone, without concern that some asynchronous event like an interrupt will modify those bytes.

¹The capacity of a 64-bit counter is 2^{64} and modern clock rates rarely exceed 3.0 GHz. $2^{64}/3.0 \text{ GHz} = 194.98 \text{ years}$.

This allows compilers to optimize leaf functions that have small static data footprints since in those cases the compiler can avoid setting up and tearing down a stack frame. However, PEBIL's reliance on the stack for protecting program state causes this optimization to be a complication for PEBIL when instrumenting the code within those functions. When generating the code for an instrumentation point that is both in a leaf function and requires state protection, PEBIL must take care to not overwrite any data potentially inside the red zone. To accomplish this, PEBIL performs static analysis on all of a program's functions in order to recognize which are leaf functions. When generating code for an instrumentation point inside a leaf function that requires other state protection, PEBIL automatically generates extra code around the instrumentation point which keeps the stack pointer out of the red zone. This is done by decrementing the stack pointer by 128 bytes just before executing all other state saving operations then incrementing the stack pointer by 128 bytes just after executing all other state restoration.

B. Thread-local Instrumentation Data

In order to provide thread-specific information about the activities of an instrumented program, it is necessary for PEBIL to provide a mechanism which allows a thread to quickly find a thread-local copy of its instrumentation data structures. The distinction between maintaining a thread-local copy of a data structure and maintaining synchronization around a single shared data structure is important because performance data gathered using the former approach can often be far more useful than the latter. Whether data structures are thread-local can also have dramatic consequences for the performance of the instrumented program. For example, a basic optimization for handling the memory address stream emanating from a program is to buffer those addresses and process them in batch. A tool which attempted to share a single buffer between all threads would require synchronizing access to that buffer either at a very fine granularity, which would invariably generate large amounts of coherence-related bus traffic, or 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 which allows thread-local data structures to be generated for each new running thread. These data structures are made accessible through a single table, shared by all threads, 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 so in practice is limited to holding the addresses of other interesting data structures for all but the simplest instrumentation tools. For collecting memory access traces, for example, this pool contains just the address of a buffer which holds memory addresses that are waiting to be batch processed. The remainder of this section details how a thread accesses its private memory pool at runtime as well as an optimization which allows the location of that memory pool to be cached for short periods of time.

1) *Finding Thread-local Data 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 program. For some thread, that thread's index into this table `IDX` is derived using the formula $IDX = (TID \gg 12) \& 0xffff$, 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 scratch² in as few as three instructions in `x86_64`:

```
mov %fs:0x10,%r10 // move TID into %r10
shr $12,%r10      // %r10 = %r10 >> 12
and $0xffff,%r10  // %r10 = %r10 & 0xffff
```

Note however that this will generate identical indices for any threads which share bits 12-27 in their unique identifiers. In principle, no guarantee of uniqueness of these bits between threads exists. In practice, though, we have seen conflicts of this sort only begin to appear when running over 16 threads per process. To resolve cases like this, currently PEBIL intercepts all thread create calls, verifying for each new thread that no existing thread has a table index which conflicts with the new thread's table index. Upon conflicts, rather than backing into a slower mode which is able to guarantee a resolution, PEBIL currently generates a runtime error. None of the experiments described in Section IV ever encountered this failure.

2) *Caching Thread-local Data:* Even though the instruction sequence generating the location of a thread's private memory pool is short, it needs to be executed at every instrumentation point which refers to the thread's private memory pool. Unfortunately, we can generally expect detailed instrumentation tools to require access to the memory pool very frequently, as often as every basic block or memory instruction. Instead of requiring the location of a thread's private memory pool to be re-computed 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's current implementation examines code at the function level to determine whether any single register is dead at every point in the function. If no such register is found, PEBIL falls back to computing the thread's private memory pool location each time it is required. However, if such a register is found, PEBIL inserts code to compute the location of the memory pool only at the entry and re-entry (that is, immediately following a call to another function) points to the function. Future work within PEBIL should include facilities which analyze the program at more granular levels such as within loops or basic blocks in an attempt to further utilize dead registers to cache this value.

III. RELATED WORK

Here we identify other notable x86/Linux binary instrumentation projects. There are many other x86/Linux binary

²In `x86_64` a thread's unique identifier is always stored in `%fs:0x10`.

instrumentation projects – Pin [2], DyninstAPI [3] and Valgrind [8] being the most popular. This section focuses on Pin and DyninstAPI due to space considerations and because Valgrind was designed for particular types of useful but heavyweight instrumentation tools, distinguishing itself in terms of functionality at the cost of efficiency. Similarly, numerous binary instrumentation tools covering other architectures and operating systems exist but are out of the scope of this work.

Pin is a popular dynamic binary instrumentation tool which supports threads by providing API hooks for thread creation/destruction and for associating a number of data structures with each thread. PEBIL performs all state preservation on a thread’s stack, while Pin sets aside a distinct region of heap memory for every thread. In Pin this region of memory is made accessible to a multithreaded instrumented program by storing its location at all times in some general purpose register which is stolen from the program. This approach is possible because Pin utilizes a sophisticated dynamic code optimization engine to transform the program around the stolen register.

DyninstAPI is a dynamic binary instrumentation tool and static rewriter. DyninstAPI supports threading by providing a class which allows the control and examination of runtime threads. DyninstAPI also provides the tools for building limited expressions to implement hand-coded instrumentation code sequences, which also have access to the unique identifier of the 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 anything PEBIL provides, however they are for 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, DyninstAPI uses a far more heavyweight mechanism which dives through at least two functions plus any necessary state protection that results. DyninstAPI also lacks a facility for caching the thread identifier or other thread-related information that might allow instrumentation code to reuse thread-related values once computed. In the following section, our experiments focus on PEBIL and Pin because the thread access mechanisms in DyninstAPI generate overheads that are too large to practically run multithreaded codes of any significant size with fine-grained instrumentation.

IV. EXPERIMENTAL RESULTS

We now present the results of an empirical study on the overhead of gathering memory address traces for multithreaded codes. These experiments use aggressively optimized PEBIL and Pin tools to collect memory address traces for the OpenMP implementations of the NAS Parallel Benchmarks[4] A list of all benchmarks along with brief descriptions of them is provided in Table I. The test system used in all experiments is a dual-socket, 8-core Intel Xeon X3450 where each core has a 32KB dedicated L1 cache and 256KB of L2 cache. All four cores on a socket share 8MB of L3 cache and all 8 cores on the board share 16GB of memory. Each experiment is performed

using a single process and 8 OpenMP threads, chosen because it is equal to the number of CPU cores available on the system. 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	Code Size (KB)
BT	block tri-diagonal solver	B	65.9
CG	conjugate gradient	B	14.6
DC	data cube	W	25.6
EP	embarassingly parallel	B	7.9
FT	discrete 3D fast Fourier Transform	B	19.8
IS	integer sort	C	5.3
LU	lower-upper Gauss-Seidel solver	B	67.6
MG	multi-grid on mesh sequence	B	26.9
SP	scalar penta-diagonal solver	B	58.3

A. Thread Support Overhead

The first experiment we present is intended to demonstrate the overhead of frequently accessing thread-specific data in the multithreaded workload described in Table I. To show this we use PEBIL and Pin to instrument each program in this workload to fill a buffer with every memory access emanating from program executable’s image. To avoid the negative performance consequences of providing concurrent access to a single buffer by all threads, each thread interacts with a private buffer throughout the instrumented program run. The sequence of interactions a thread has with its private buffer are to (1) fill it with memory addresses, (2) call into a minimal processing function which updates a shared count of the number of memory accesses seen by the process then empties the buffer, and finally (3) return control to the instrumented application to refill the buffer. The buffer size for each tool was optimized for speed using a small set of empirical tests resulting the following buffer sizes which are used throughout the remainder of these experiments – 32KB for PEBIL and 128KB for Pin. Since the focus of this experiment is on the overhead of providing threading support rather than any specific application of using memory address traces, the memory addresses in those buffers are simply discarded as quickly as possible, then control returned to the program to begin filling the buffer again. The results of these experiments are given in the upper part of Table II and are phrased as the slowdown of the instrumented application runtime relative to the uninstrumented runtime. These results demonstrate that, while the overhead of collecting all memory addresses with PEBIL is higher than with Pin, the level of overhead in PEBIL is generally reasonable.

B. The Effects of Sampling

Gathering and utilizing the entire memory address stream of a program is generally impractical due to the amount of processing that is required on top of the overhead of collecting the addresses. A common approach to easing this difficulty is to introduce interval-based sampling, reducing the fraction of the memory address stream that is processed. In many cases this can result in extracting the desired properties of the

TABLE II
MEMORY ADDRESS TRACE COLLECTION OVERHEAD

		BT	CG	DC	EP	FT	IS	LU	MG	SP	GEOM-MEAN
Full Trace	PEBIL	14.12	5.78	2.01	2.50	6.23	5.75	10.75	10.45	5.24	5.90
	Pin	7.08	4.42	4.14	2.95	4.37	6.04	5.71	6.26	3.49	4.76
50% Sampled	PEBIL	8.51	4.11	1.86	1.98	4.04	3.99	6.59	6.41	3.38	3.27
	Pin	11.46	4.69	3.46	2.83	3.63	4.95	11.60	8.67	8.98	5.89
10% Sampled	PEBIL	4.09	2.74	1.85	1.56	2.36	2.53	3.32	3.37	1.91	2.52
	Pin	11.20	4.48	3.52	2.71	3.05	4.67	11.06	6.30	8.13	5.40
1% Sampled	PEBIL	3.07	2.43	1.81	1.47	1.98	2.19	2.59	2.63	1.58	2.14
	Pin	9.52	4.35	3.40	2.73	2.92	3.98	10.10	5.81	8.14	5.07

memory address stream while greatly reducing the cost of that extraction[6]. In the event that the instrumentation tool can disable or remove instrumentation code during the instrumented program run, sampling in this fashion can also reduce the cost of collecting the memory addresses themselves. PEBIL is capable of quickly disabling and re-enabling arbitrary instrumentation points at runtime by swapping instrumentation code for nops. Pin can remove and arbitrarily reinstrument code, which is more versatile than PEBIL’s approach but comes at the expense of runtime efficiency.

We demonstrate this by introducing the concept of interval-based sampling into the experiment discussed in Section IV-A. In this set of experiments, we use three shades of sampling whereby we capture the first 50%, 10% and 1% of the memory addresses of every interval of 1 billion addresses. During these runs, the instrumentation tool disables and re-enables instrumentation around the sampled regions of the program’s memory address stream. The results of this are seen in Table II, which show that the overhead of collecting memory accesses with PEBIL diminishes as more of the program’s memory addresses are lost to sampling. This is not necessarily the case for Pin even with the large sampling windows used in these experiments since instrumentation must be regenerated at each sampling exchange. How effective removal and re-insertion of instrumentation is depends principally on the number of instrumentation points that are present in the program.

For the tools being used here these values can be approximated by the size of the code in the benchmark executable, which can be seen in Table I. Because of the computationally expensive methods Pin uses to insert and remove instrumentation, proportional improvements in overhead is not seen across the board when performing sampling. Note the increases in overhead for sampling with Pin for benchmarks with larger code footprints like BT, LU and SP in comparison to the far better behavior on benchmarks with a smaller code footprint like CG, EP and IS. It is important to point out here that these results do not destroy the case for using sampling in cases where the processing of the captured memory addresses is expensive because sampling can still be used to mitigate the processing overhead, though they indicate that the tool writer must be careful in determining how to implement sampling.

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 will

utilize complex models of parallelization to include both inter-process and shared memory models built on top of threading platforms like OpenMP and pthreads. Support for sophisticated analysis tools support is necessary for helping the community better understand what is required to make this transition in a way that minimizes errors and performance pitfalls. Toward that goal, this work presented an extension the open source static binary instrumentation framework PEBIL which provides support for instrumenting multithreaded programs implemented with both pthreads and OpenMP. We discussed PEBIL’s threading model, some optimizations surrounding that model, and how that model compares to two other popular binary instrumentation platforms – Pin and DyninstAPI – in theoretical terms. We then used a series of memory address trace collection tools to demonstrate that PEBIL has a reasonable thread model in terms of the functionality offered and the overhead that it introduces, and that PEBIL is well-suited to taking advantage of sampling in order to reduce the overhead of collecting memory address traces.

REFERENCES

- [1] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snaveley. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.
- [2] 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.
- [3] B. Buck and J.K. Hollingsworth. An api for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
- [4] 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, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing’91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991.
- [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] L. Carrington, A. Snaveley, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. *Computational Science/CCS 2003*, pages 701–701, 2003.
- [7] M. Laurenzano, M. Meswani, L. Carrington, A. Snaveley, M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. *Euro-Par 2011 Parallel Processing*, pages 79–90, 2011.
- [8] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.