Parallelizing Heavyweight Debugging Tools with MPIecho

Barry Rountree
Lawrence Livermore National Laboratory
rountree@llnl.gov

Guy Cobb
University of Colorado
guy.cobb@gmail.com

Todd Gamblin
Lawrence Livermore National Laboratory
tgamblin@llnl.gov

Martin Schulz
Lawrence Livermore National Laboratory
schulzm@llnl.gov

Bronis R. de Supinski
Lawrence Livermore National Laboratory
bronis@llnl.gov

Henry Tufo
University of Colorado, National Center for Atmospheric Research
tufo@ucar.edu

ABSTRACT

Idioms created for debugging execution on single processors and multicore systems have been successfully scaled to thousands of processors, but there is little hope that this class of techniques can continue to be scaled out to tens of millions of cores. In order to allow development of more scalable debugging idioms we introduce *MPIecho*, a novel runtime platform that enables cloning of MPI ranks. Given identical execution on each clone, we then show how heavyweight debugging approaches can be parallelized, reducing their overhead to a fraction of the serialized case. We also show how this platform can be useful in isolating the source of hardware-based nondeterministic behavior and provide a case study based on a recent processor bug at LLNL.

While total overhead will depend on the individual tool, we show that the platform itself contributes little: 512x tool parallelization incurs at worst 2x overhead across the NAS Parallel benchmarks, hardware fault isolation contributes at worst an additional 44% overhead. Finally, we show how *MPIecho* can lead to near-linear reduction in overhead when combined with Maid, a heavyweight memory tracking tool provided with Intel's Pin platform. We demonstrate overhead reduction from $1,466\%$ to 53% and from 740% to 14% for cg.D.64 and lu.D.64, respectively, using only an additional 64 cores.

1. INTRODUCTION

Existing debugging approaches in high performance computing have not been able to scale beyond roughly 10K MPI ranks. Classic debugging interfaces such as Totalview [20] have been overwhelmed by the need to control and display the state of all processors from a single screen. Heavyweight debugging tools such as Valgrind [16] and Parallel Inspector [10] are indispensable when solving smaller problems, but their overhead precludes their use at scale except as a last resort: memory checking can reach 160x slowdown and thread checking can reach 1000x. Current best practices for debugging hundreds of thousands of cores relies far too much on the ingenuity of the debugger to stretch these tools beyond their limitations. This will only become more difficult as machines scale to tens of millions of cores.

In this paper we propose leveraging these additional cores for *parallelizing heavyweight tools* in order to reduce the overhead incurred by existing tools and allowing the development of novel approaches. Per-instruction instrumentation such as used by the Maid memory access checking tool can be rendered effectively embarrassingly parallel. The more interesting cases, such as parallelizing read-before-write detection, can still show substantial reduction in runtime overhead by duplicating write instrumentation and parallelizing read instrumentation.

We also show this platform is flexible enough to be used in hardware debugging and performance analysis. By assuming cloned ranks should exhibit identical execution we can perform fast *hardware fault detection* by observing when this assumption is violated and correlating the fault to a particular node. We examine a case study of a recent processor bug at LLNL that has informed the design of *MPIecho*. Additionally, identical execution allows for the parallelization of *hardware performance counter collection*, allowing measurement of an arbitrary number of computation-related counters for a single rank during a single run — a feature particularly useful when dealing with "chaotic" codes such as ParaDiS [2]. Finally, cloned ranks may allow for scalable *sensitivity analysis*, allowing hundreds of experiments per rank per run.

We present a design and implementation overview in section 2 and measure its overhead across the NAS Parallel Benchmark suite [14]. We then describe two simple implemented tools: *SendCheck* (Section 3.4) and Maid (Section 4). The former checks to see if all *send* buffers on all clones are identical: clones that aren't may indicate a node-specific hardware fault. The latter demonstrates how a heavyweight tool based on Intel's Pin [13] can be successfully parallelized. We detail related work in section 5 and list several possibilities for future work in section 6.

2. OVERVIEW

The goal of the *MPIecho* platform is to provide duplicate execution of arbitrary MPI ranks. Overhead should be kept

WHIST 2011 Tucson, Arizona, USA
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

to a minimum and the behavior of the clones should not perturb the correctness of execution. In this section we describe the software architecture of the platform as well as the experimental measurement of the tool's overhead. Further information about MPIecho can be found in a companion document [3].

2.1 Design and Implementation

At at high level, the design is simple. At startup, a cloned MPI rank r (the *parent*) will distribute all messages it receives to one or more clones c. Messages sent from the clones are routed to the parent if necessary but are not otherwise distributed to the rest of the system (see Figures 1, 2, and 3). So long as the state of a rank depends only on the initial program state and inputs received via the MPI API, this general approach will guarantee identical execution on the clones. The overhead of this approach is dominated by the additional messages sent to the clones. If the cloned rank is on the *critical path*, any additional overhead will accrue in overall execution time.

At a lower level, the function calls provided by the MPI API have IN, OUT, and INOUT parameters. When any function call completes, the OUT and INOUT parameters should be identical across the parent and clones. For example, an `MPI_Recv` call has a buffer updated by the incoming message. This condition also applies to the parts of the API that are not involved with communication, for example querying the size of a communicator or constructing a derived datatype. A naive implementation could simply copy every OUT and INOUT parameter to the clones. This approach incurs unnecessary overhead and relies on non-portable knowledge of how opaque handles are implemented. Instead, we have minimized communication between the parent and clones using the following techniques:

1. *Broadcast.* The parent communicates with the clones via `MPI_Bcast` using a communicator dedicated to that purpose. In MPI implementations we are familiar with, this implies the parent sends out only $\log_2(|c|)$ messages per communication (where $|c|$ is the number of clones). We have found this sufficiently fast for our needs, but scaling this approach to thousands of clones may require the parent sending a single message to a single lead clone with the lead clone then (asynchronously)broadcasting to the remaining clones.

2. *Opaque handles.* MPI relies on several different opaque handles that cannot be queried except through the API. Copying these requires knowledge of their internal structure; this tends to be non-portable. Instead, we only communicate on a "need-to-know" basis. For example, a call to `MPI_Comm_split` will not be executed by the clones. Instead, the parent will send the associated index value of the new communicator to the clones, along with values for size and rank. Clones will not use this communicator for communicating and so no additional information is needed. Calls to `MPI_Comm_size` and `MPI_Comm_rank` can be resolved locally without any further communication with the parent, thus cutting out a potentially significant source of overhead.

3. *Translucent handles.* The `MPI_Status` datatype is only partially opaque: users may query several values directly without going through the API. Any call that potentially modifies a status results in copying just these values to the clones. Calls such as `MPI_Waitall` with partially visible status values have these values batched together and sent using a single broadcast.

4. *Vectors.* MPI calls such as `MPI_Alltoallv` allow sending and receiving (possibly non-contiguous) vectors. Using derived datatypes, we construct a datatype that describes all updated buffers and uses this to issue a single `MPI_Bcast` to the clones.

5. *Non-blocking communication.* Both the clone and parent record the count, type and buffer associated with each `MPI_Request`. In the case of the `MPI_Wait` family of calls both the parent and clone implicitly wait on an identical index and the broadcast occurs when the wait returns. In the case of `MPI_Test` the results are communicated in a separate broadcast, followed by the updated buffer if the test returned true.

6. *Barriers.* There exist MPI calls with no OUT or INOUT parameters, such as `MPI_Barrier`. These need not be executed by the clone at all, as no program state is changed. The clones resynchronize with the parent at the next communication call.

7. *Return values.* MPI calls return an error value, but there is no provision for recovery if the value is anything other than `MPI_SUCCESS`. We make the assumption that if an error condition occurs on either the parent or a clone, the only sensible thing to do is indicate where the error occurred and halt. Future versions of MPI may make better use of the return codes; if so they will need to be distributed to the clones as well.

The implementation proper is built on the PMPI profiling interface. We intercept each MPI call and route it to our library. We used the `wrap` [8] PMPI header generator to create the necessary Fortran and C interface code. These design choices allow us to use a single broadcast in the common case, and never more than two broadcasts per MPI call. Overhead is dominated by the size and number of messages. Effectively, the worst case cost is:

$$\begin{aligned}\text{Overhead} =& \text{Bcast (nranks } \times \text{ typesize } \times \text{ messagesize)} \\ &+ \text{Bcast (nranks } \times \text{ statussize)}\end{aligned}$$

Because the clones do not execute any Send functions (unless required by a tool implementation) they will tend to not remain on the critical path: the overhead should be limited to the direct cost of the barriers except in pathological cases.

2.2 Experimental Measurement of Overhead

We intend this platform to support parallel tools, but the time saved by committing more processors to the tools will eventually be offset by the additional time necessary to communicate with those processors. It is important for the overhead of the tool itself to contribute as little as possible to the overall overhead.

All experiments in this paper were executed on the Sierra cluster at Lawrence Livermore National Laboratory. We compiled the NAS Parallel Benchmark Suite [14], *MPIecho* and tools using GCC 4.1.2 Fortran, C and C++ compilers and -O3 optimizations. We ran the experiments using MVAPICH2 version 1.5. All results are expressed in terms of

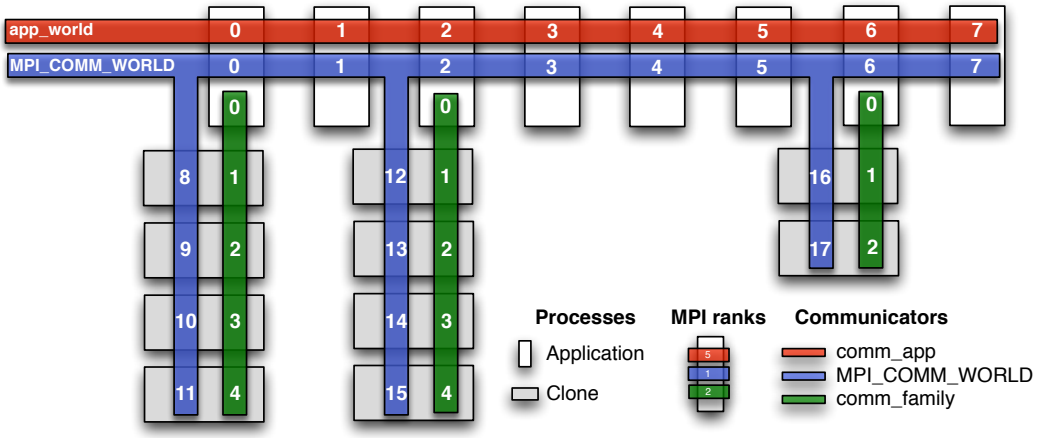

Figure 1: Architecture of *MPIecho*

Bench-mark	Number of clones						
	8	**16**	**32**	**64**	**128**	**256**	**512**
bt	1.5	-2.7	-2.0	7.3	7.5	6.3	11.5
cg	-2.0	-0.3	2.6	3.6	9.0	8.5	15.7
ft	0.3	3.8	2.6	3.5	3.9	3.4	2.7
is	8.7	14.0	13.4	14.4	11.4	10.5	17.5
lu	-1.4	4.4	2.2	2.1	-1.5	1.5	9.2
mg	26.5	30.8	33.7	41.0	59.5	67.7	99.0
sp	2.3	0.0	3.6	1.4	6.5	10.2	15.5

Table 1: Percent overhead for cloning rank 0

percent time over the baseline case run without *MPIecho*. Process density has a significant affect on execution time: using 64 12-core nodes to run the baseline 64-process benchmarks can be usefully compared with 64 processes and 8×64 clones also running on 64 12-core nodes due to increased cache contention. We made a best-effort to keep process densities similar across all runs.

In figure 2.2 we show the percent measured overhead for several clone counts. Creating up to 512 clones of node 0 only incurs 18% execution time overhead for all benchmarks other than MG. In the case of MG, we observed an unusually high ratio of communication time to computation time which did not afford the opportunity to amortize the cost of the broadcasts to the same extent. However, even in this worst case we note that this approach still scales well: 512 clones of node 0 resulted in only doubling execution time. These results establish the overhead incurred by the platform is low enough to be useful for parallelizing high-overhead tools.

3. SEND BUFFER CHECK

In this section we give a brief outline of the design, implementation and performance of *SendCheck*, a tool used to detect intermittent hardware faults. We illustrate how such a tool could have been useful in diagnosing a recent CPU bug at Lawrence Livermore National Laboratory. With increasing processor and core counts, we expect similar tools to be increasingly important.

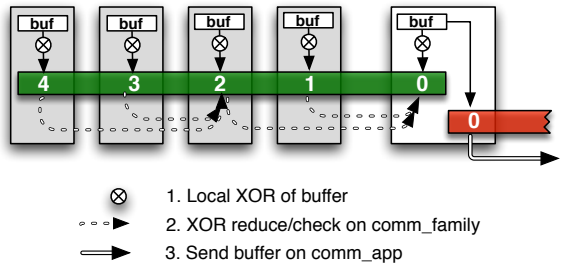

Figure 2: Sending from a cloned process (Used in Send Buffer Check)

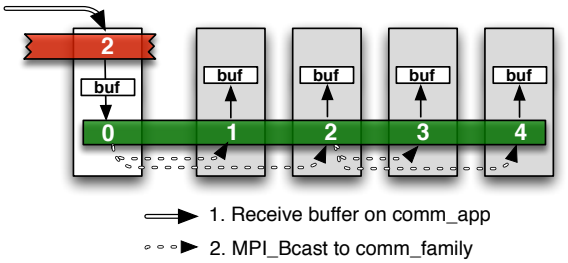

Figure 3: Receiving to a cloned process

3.1 Case Study

A user had reported seeing occasional strange results using the CAR [4] climate model. To the best of the user's knowledge the problem was isolated to a particular cluster and did not always manifest itself there. At this point, the members of the Development Environment Group at LLNL was asked to lend a hand with debugging. Since the problem appeared to be isolated to a particular cluster, the possibility of a hardware fault was raised early on in the process. This was a bit of good luck; many installations do not have multiple clusters that would allow running suspect programs on multiple separate hardware configurations.

The first task was to determine if the problem was caused by a particular compute node. Node assignments on this particular cluster are nondeterministic and the user might only occasionally be assigned a particular bad node. After dozens of test runs the faulty node was eventually isolated. For a particular sequence of instructions a particular core failed to round the least significant bit reliably. Because CAR climate model is highly iterative, repeated faults ultimately caused significant deviation from correct execution.

This bug was pathological in several ways: only a very specific instruction sequence triggered the fault, the fault is intermittent, and when the fault does occur it will only be noticeable in calculations that are sensitive to the values of the least significant bit. Indeed, one of the most curious manifestations of this fault was the observations that error could be introduced into partial results, disappear and the then reappear later. The design of the software allowed only partial results to be checked per timestep. These results did not provide sufficient granularity to isolate the fault to before or after a particular MPI message. For a complete description of the bug and the debugging process, see Lee [12].

3.2 Design and implementation

We begin by noting that we are only interested in such faults that affect the final state of the solution at the root node. The remaining nodes only affect the root node by sending messages to it. Rather than having to validate the entire machine state across multiple clones we have a far more tractable problem of validating only messages that would be sent by the clones.

The naive implementation is straightforward. At each MPI call where data is sent to another node, each clone copies its send buffer to the parent and the parent compares the buffers. If the buffers are not identical, the program is nondeterministic in a sense not accounted for by *MPIecho*. If all of the buffers are unique the cause of the nondeterminacy likely lies in the software. If only a single clone has a different buffer, the most likely explanation would be a fault isolated to that node. A small number of subsequent runs should be able to distinguish between the two cases.

Copying potentially large messages is both expensive and unnecessary. Our implementation takes an md5 hash [5] of the message buffer and then executes a bitwise MPI XOR reduce across all clones of a given parent (see Figure 2). This limits the amount of communication to a 16-byte hash per clone and allows the parent to perform an inexpensive check for differences. The saving in communication far outweighs the expense in creating the hash. This approach will not catch all errors: pathological conditions may be masked by the XOR. While unlikely, if this is suspected the user may fall back to the naive implementation.

Benchmark	Number of clones						
	8	**16**	**32**	**64**	**128**	**256**	**512**
bt	2.1	6.7	-0.2	0.0	-0.8	2.5	-0.3
cg	7.5	8.4	13.0	11.3	11.3	14.9	13.4
ft	-0.5	-1.8	-0.1	-0.3	-0.7	-0.7	-0.4
is	46.8	46.9	55.0	48.1	50.8	50.8	42.4
lu	6.0	0.2	1.6	3.4	10.9	8.6	-1.5
mg	1.2	2.8	4.1	5.1	4.5	1.2	3.7
sp	-1.8	5.4	-0.7	5.7	1.6	-1.8	-2.6

Table 2: Percent additional overhead for Sendcheck

Benchmark	Number of clones						
	8	**16**	**32**	**64**	**128**	**256**	**512**
bt	2.0	2.3	-3.7	5.7	5.1	7.4	9.46
cg	11.7	14.5	22.9	22.3	28.5	32.1	39.16
ft	0.5	2.6	3.2	3.9	3.9	3.3	3.01
is	62.2	70.3	78.6	72.2	70.8	69.4	70.05
lu	2.8	2.9	2.1	3.8	7.5	8.4	5.76
mg	26.2	32.7	37.4	46.3	64.5	67.4	103.59
sp	0.3	5.2	2.7	7.0	7.9	8.0	12.08

Table 3: Percent total overhead for Sendcheck

3.3 Experimental Setup

For each of the benchmarks run we dynamically generated an md5 hash value for every send buffer before each MPI call executed. This was not restricted to the `MPI_Send` family, but included any call that transferred data to another MPI rank, including `MPI_Isend/MPI_Wait` and `MPI_Alltoallv`. The bitwise XOR of all hashes was send to the parent using an `MPI_Reduce`.

3.4 Discussion

Tables 3.4 and 3.4 show the additional and total overhead of the *SendCheck*. The overhead is marginal except in the case of IS. Here, the overhead introduced by the extra calls to md5sum could only be amortized over small amount of program execution. Validating messages up through 512 clones usually requires at most a 5% additional overhead with the worst case being less than 55%. (In this case, numerous small messages did not allow for the usual amortization of the md5sum costs. We expect this overhead can be lowered significantly by selectively applying md5sum based on message size.)

Had *MPIecho* been available when diagnosing the Opteron hardware fault we would have been able to either rule out hardware faults quickly (if the error appeared to be independent of node assignment) or identify the hardware using a small number of runs. This would have saved months of debugging time, and we expect this tool be brought into service debugging production code.

4. MAID

The Intel binary instrumentation platform Pin [13] provides several examples of tools that leverage its features, for example address trace generators, cache simulations and memory checkers. In this section we examine the Maid tool and explain not only why the serial case exhibits so much overhead but also how this kind of tool may be parallelized

Bench-mark	Number of clones						
	1	**2**	**4**	**8**	**16**	**32**	**64**
bt	1466	737	375	207	124	78	53
cg	317	138	131	144	187	185	194
ft	521	294	146	98	106	109	97
is	375	239	144	127	110	107	128
lu	740	369	183	100	58	24	14
mg	810	428	217	136	108	90	101
sp	376	180	84	37	23	14	12

Table 4: Percent total overhead for Maid

using Pin's ability to selectively instrument individual instructions. We then present results using the combination of Pin, Maid, *MPIecho* and NAS Parallel Benchmark suite [14].

4.1 Overview

Pin can be thought of as a just-in-time compiler with two separate user interfaces: *instrumentation* and *analysis*. Instrumentation code determines where dynamic code will be inserted, and analysis code is what executes at the instrumentation points. Instrumentation code executes once when the binary is loaded into memory. Analysis code executes whenever the underlying code executes. The overhead due to instrumentation is paid only once and thus this code can be relatively complex. The overhead due to analysis will be paid every time the instruction is executed.

Maid identifies all instructions that access memory (both reads and writes) and inserts analysis code that checks if the effective address used by the instruction matches any of the addresses provided by the user. Similar functionality can be found in watchpoints in modern debuggers. However, a serial debugger must check every instruction to see if any memory references are of interest. A naive parallel implementation would divide memory regions among the clones and thus have each clone check only a small fraction of the potential address space. However, this ends up being as slow as the serial case: every clone still checks every instruction.

Instead, given n nodes, each clone instruments every nth instruction, starting from the ith instruction (where i is the node rank). This does not necessarily distribute the work evenly. In the pathological case were a single instruction in a very tight loop forms the majority of execution time, effectively no parallelization will have occurred. However, more realistic cases involve dozens to hundreds of instructions at the innermost loop level. Given sufficient clones each clone will instrument only a single instruction in the loop.

4.2 Experimental Setup

Only clones executed the dynamic binary instrumentation. We measure overhead against the no-clone, uninstrumented case: a single clone with all memory instructions are instrumented. For two clones, each clone instrumented half of the instructions, etc. The Maid tool is set up to check multiple memory locations with the overhead increasing as more locations are checked. For this set of experiments we check only a single memory location.

4.3 Results

Table 4 lists our results. For the common case over a small number of clones we achieved near-linear scaling. In the most dramatic cases, bt went from an overhead of $1,466\%$ to only 53% and lu went from an overhead of 740% to 14%, both using only 64 additional cores as clones. However, we note that in several cases the overhead due to additional nodes begins dominating the savings gained by those nodes: in the cases of cg, is, and mg performance is worse at 64 clones than at 32 clones. The worst case, cg, still drops from 317% to 131% using only 4 clones.

4.4 Discussion

Moving from a serial to parallel model of computation not only led to faster solutions, it also allowed new kinds of problems to be solved. *MPIecho* allows a similar transformation to be brought to bear on debugging tools, especially those built using heavyweight techniques such as dynamic binary instrumentation. Methods that may be been left unexplored or unused to do prohibitive overheads may now be feasible, so long as they can be parallelized.

5. RELATED WORK

This paper is the most recent in an extensive body of work using PMPI (the MPI profiling interface) to record or modify MPI program behavior (e.g., Umpire [24], Libra [9]) and Adagio [21]). In creating a platform for future tools, *MPIecho* comes closest to PNMPI [22] which allows chaining disparate tools together into a single software stack. In a sense, PNMPI allows for vertical integration of compatible tools while *MPIecho* allows horizontal integration of (potentially) incompatible tools. We are actively exploring how the two platforms may be integrated.

The idea of node duplication via PMPI has been used in the fault tolerance community, particularly rMPI [7], MR-MPI [6] and VolPEX [11]. Here, duplication ensures that if any particular node goes down its duplicate will step in to allow execution to continue without interruption. This approach requires effectively building a second or third supercomputer in order to provide a minimum of redundant capacity, and this cost has prevented wide adoption of this technique in the high performance computing community. In contrast, *MPIecho* targets only a subset of MPI ranks and only for the purposes of debugging and measurement.

Heavyweight software debugging tools have existed for decades and continue to be reliable and essential components of programmers' toolboxes. Valgrind and its memcheck [16] are the canonical example. Both Pin [13] and dyninst [1] also provide dynamic binary rewriting and instrumentation platforms and provide tools that range from thread checking (Parallel Inspector [10]) to performance analysis (Open Speedshop [23] to cache simulation (cachegrind [15]). However, to the best of our knowledge none of these platforms or tools have been parallelized, with the single exception of an attempt to introduce multithreading into Valgrind [19]. While this approach shows promise, it is limited to allowing existing multithreaded code to execute in parallel.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have sketched the design and implementation of *MPIecho*, a platform that allows cloning of arbitrary MPI ranks. We have demonstrated how this tool can be used to validate buffers in order to isolate suspected hardware faults as well as show how cloned ranks may be used to reduce the overhead of heavyweight parallel tools. We

envision many potential tools that will be built on top of this platform.

Additional Heavyweight Tool Parallelization.

While Valgrind's *memcheck* is the most well-know heavyweight tool, there are many others based on dynamic binary rewriting that have even larger overheads. For example, the thread checking functionality in Intel's Parallel Inspector can exceed 1000x. At this point, we do not know how much of this overhead is parallelizable. However, future supercomputing architectures will be far more dependent on aggressive threading, and even bringing this overhead down by one order of magnitude should be a significant help to making these upcoming architectures useful.

Performance Counter Sampling.

The range and detail of hardware performance counters has blossomed across processor architectures, but Intel-based systems are still constrained by a limited number of registers that can be used to read these processors. Multiplexing has been used in several tools, for example perfctr [18], but this kind of sampling technique can be problematic for counters that do not increment often. Where multiplexing is inappropriate, the only other alternative has been multiple runs until all desired counters have been recorded. For measurements that are not affected by MPI communication patterns, *MPIecho* provides a way of gathering an arbitrary number of counters for a single rank during one execution. For example, newer Intel processors provide a counter that counts sampled loads where latency exceeds a user-specified threshold. The most effective way of using this tool is to make multiple runs, each using a different threshold. *MPIecho* allows the user to make a single run with each clone using a unique threshold.

Dynamic Cloning.

The work presented in this paper has focused on making a large number of clones available to a single rank. We envision use cases where the rank of interest may change as the program progresses. *MPIecho* in combination with task migration could be used to dynamically assign clones to ranks, perhaps focusing on the critical path of execution.

Multiple Tool Chains.

The population of clones need not be focused on a single task. We envision a scenario where some nodes are used for performance counter collection while others monitor integer overflow conditions and memory accesses. This could be particularly powerful when combined with PNMPI [22].

Sensitivity Analysis.

Floating point calculations use doubles by default, and it's an open research question as to how much faster code might run if floats were used versus how much more accurate the code could be if quads were used. Existing work measures the effect of individual instructions, but *MPIecho* could allow a much faster, coarser granularity by operating on transmitted buffers.

Deterministic Execution.

Some non-deterministic codes such as ParaDiS are optimized to operate on messages as they arrive. This is non-deterministic on most systems, which leads to unique execution on every run, even given the same initial starting conditions. This makes studying these codes difficult: performance counters, for example, are valid only on the run for which they are gathered. *MPIecho* enforces a particular message order, but might also be modified to perturb the message order in deterministic ways.

Taint Analysis.

This approach need not be limited to scientific parallel kernels. A low-level understanding of how input changes program behavior requires dynamic binary instrumentation. A successful taint analyzer, TaintCheck, was built using the Valgrind platform [17]. *MPIecho* would allow multiple instances of a serial program can be launched with MPI, each instance of which would be instrumenting only a subset of the entire binary. The same technique could be applied to multithreaded and MPI-based applications where tracing the effects of potentially unfriendly input is required.

Parallel Simulation.

Simulations are perhaps the most heavyweight of all tools, where overhead is commonly measured in terms of tens of thousands of percent. *MPIecho* allows much of this overhead to be executed in parallel. Using checkpointing, new clones are generated continually. Each node runs as normal until it reaches a user-specified point where it transitions into simulation mode. At this point the node is disconnected from the rest of the family communicator and it continues simulation until it reaches the next MPI call. At that point it exits and is a candidate to be brought back via checkpointing. Ideally, thousand of clones could be simulating different segments of code in parallel. While the individual simulation segments are not run any faster, nearly all of the simulation execution will execute in parallel.

7. REFERENCES

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

[2] V. Bulatov, W. Cai, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable line dynamics in ParaDiS. In *Supercomputing*, Nov. 2004.

[3] G. Cobb and B. R. H. Tufo. MPIecho: a framework for transparent MPI task replication. Technical Report CU-CS 1082-11, University of Colorado, 2011. Submitted to Cluster 2011.

[4] W. D. Collins and P. J. Rasch. Description of the NCAR community atmosphere model (CAM 3.0). Technical Report TN-464+STR, National Center for Atmospheric Research, 2004.

[5] L. P. Deutsch. *Independent implementation of MD5 (RFC 1321)*. Aladdin Enterprises, 2002.

[6] C. Engelmann and S. B ohm. Redundant execution of hpc applications with mr-mpi. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, 2011.

[7] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Kordenbrock, and R. Brightwell. Increasing fault resiliency in a

message-passing environment. Technical Report SAND2009-6753, Sandia National Laboratories, 2009.

[8] T. Gamblin. *The wrap MPI Wrapper Generator*. `https://github.com/tgamblin/wrap`, 2011.

[9] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *IEEE/ACM Supercomputing (SC)*, 2008.

[10] Intel Corporation. *Parallel Inspector*. `http://software.intel.com/en-us/articles/intel-parallel-inspector/`, 2011.

[11] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. VolpexMPI: an MPI library for execution of parallel applications on volatile nodes. In *The 16th EuroPVM/MPI 2009 Conference*, 2009.

[12] G. Lee. Comparative debugging: Identifying the source of rounding error on atlas587. Technical report, Lawrence Livermore National Laboratory, 2010. https://computing.llnl.gov/linux/corefputest.html.

[13] C. 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 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[14] NASA Advanced Supercomputing Division. *NAS Parallel Benchmark Suite*. `http://www.nas.nasa.gov/Resources/Software/npb.html`, 2006. Version 3.3.

[15] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, 2004.

[16] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dyanmic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[18] M. Pettersson. *The Linux Performance Counter Patch*. `http://user.it.uu.se/mikpe/linux/perfctr/2.6/ANNOUNCE-2.6.38`, 2011.

[19] D. Robson and P. Strazdins. Parallelisation of the valgrind dynamic binary instrumentation framework. In *IEEE International Symposium on Parallel and Distributed Processing with Applicationsa*, 2008.

[20] Rogue Wave Software. *Totalview*. `http://www.roguewave.com/products/totalview-family/totalview.aspx`, 2011.

[21] B. Rountree, D. K. Lowenthal, B. de Supinski, M. Schulz, and V. W. Freeh. Adagio: Making DVS practial for complex HPC applications. In *International Conference on Supercompuing (ICS)*, 2009.

[22] M. Schulz and B. R. de Supinski. PNMPI tools: A whole lot greater than the sum of their parts. In *IEEE/ACM Supercomputing (SC)*, 2007.

[23] The Krell Institute. *Open | SpeedShop Users Guide*. `http://www.openspeedshop.org/docs/users_guide/`, 2010.

[24] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *IEEE/ACM Supercomputing (SC)*, 2000.