Robust Non-Intrusive Record-Replay with Processor Extraction *

Filippo Gioachin
Department of Computer Science
University of Illinois at Urbana-Champaign
gioachin@ieee.org

Gengbin Zheng
Department of Computer Science
University of Illinois at Urbana-Champaign
gzheng@illinois.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
kale@illinois.edu

ABSTRACT

With the advent of increasingly larger parallel machines, debugging is becoming more and more challenging. In particular, applications at this scale tend to behave non-deterministically, leading to race condition bugs. Furthermore, gaining access to these large machines for long debugging sessions is generally infeasible. In this paper, we present a 3-step algorithm to perform what we call "processor extraction": a procedure to record the execution of a set of processors from a parallel application, and replay any of them in a controlled environment. Our technique generates very low interference in the recorded program thanks to the separation between non-determinism elimination, and detailed processor recording. In order to improve robustness and accuracy, we further augmented our algorithm with a self-correction mechanism.

1. INTRODUCTION

Parallel applications tend to behave non-deterministically, especially when they contain bugs. This means that even with the same input, the same application may produce different results over multiple runs. This can significantly complicate debugging, even in small scenarios. One common type of errors caused by non-determinism are race conditions. These are bugs where the outcome of the computation is unpredictable because it critically depends on the sequence and timing of the communication between processors.

One possibility to solve this problem is to capture the non-determinism that the application manifests, and make it repeatable. This is generally performed with a technique called "Record-Replay" [19, 22]. In order to properly work, this technique has a few requirements. First of all, the information recorded must be sufficient to allow the proper replay of the application deterministically. Secondly, the recording procedure must not perturb the application too much. If the recording has too much overhead, such that it makes the bug disappear, the record-replay technique becomes useless. For example, this happened while debugging ChaNGa [9], a cosmological simulator developed as a joint collaboration between the University of Washington and the University of Illinois. In this application, we discovered that certain messages were racing, and caused the application to crash when a particular ordering was executed. Also, when adding print statements for debugging purposes, the problem usually disappeared.

When the application is deployed in a production environment, or is benchmarked on much larger configurations, other latent race conditions may appear. Imagine the delays a network can introduce when routing packets through a torus or fat-tree interconnection, especially in the presence of congestion. On a small cluster, messages may never get out-of-order and expose a race condition. Furthermore, other bugs may appear only on large scales. For instance, the algorithm that distributes the application's input among the available processors may generate incorrect results when performing fine grain decompositions. These kinds of problems are very common in the early stages of production-level applications. They are also much more challenging to track, as they may manifest only when thousands of processors are involved in the computation, and disappear when fewer are used.

The programmer may try using smaller input datasets and smaller numbers of processors. Unfortunately, this is not always possible. As many scientific applications have physical phenomena driving the advance of the computation, using smaller input datasets may hinder the approximation of the real world. This may render the application not usable even for debugging. On the other hand, using large datasets on a small number of processors may not be possible due to the amount of memory required, or because the bug simply disappears. For example, this happened while debugging Rocstar [10], a rocket simulation program developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois. In this case, the problem only appeared when using more than 480 processors, and on a fairly large input dataset.

In both the examples given, the programmer could not reduce the number of processors if he wanted the bug to appear. While record-replay helps in making the manifestation deterministic and easy to follow, there is still the problem that the execution must be followed on possibly thousands of processing units. Luckily, in most situations, the bug appears on a specific processor in a clear way; for example with a segmentation fault or an assertion failure. Moreover, if the non-determinacy is captured correctly by record-replay, the bug's effects/causes are confined to a few processors. At this point, the programmer does not care anymore about the rest of the system, and desires only to focus on a few processors.

*This work was supported in part by the NSF Grant OCI-0725070 for Blue Waters, and by the Institute for Advanced Computing Applications and Technologies (IACAT). We used running time on the NCSA Abe cluster, which is under TeraGrid allocation grant ASC050040N supported by NSF.

PADTAD'10 July 12, Trento, Italy.

What we propose is a new technique that encompasses both the advantages of a full record scheme, which allows a single processor to be "extracted" from the application and executed as a stand-alone, and the advantages of a minimal record scheme, which incurs little overhead in the application without hiding the bug. Our technique combines these two record-replay schemes into a simple, yet powerful, three-step procedure that a user may follow to debug his application. Furthermore, since debugging may be a long process and access to large machines may be limited, we paid special attention to reducing the need for large parallel machines to the minimum. We do this by using a virtualized environment supported by a parallel emulator, which emulates the large machine using only a limited amount of resources.

In the remainder of this paper, after a comparison with related work, we illustrate our proposed three-step record-replay procedure in section 3. In section 4 we introduce the environment we used for our implementation, and in section 5 we describe the techniques we developed to make our record-replay scheme robust. We then proceed to illustrate the second phase of processor extraction, i.e the full record of a processor and its execution in a controlled environment, in sections 6 and 7. Finally, we analyze the performance of our infrastructure in section 8, and illustrate a case study in section 9. Remarks on future work are presented in the concluding section.

2. RELATED WORK

In the field of debugging, record-replay techniques have been studied extensively. Several articles [3, 21] provide broad overviews of how this technique has been applied to parallel and distributed debugging. Most of the literature focuses on applications written for shared memory systems, where races are represented by threads writing the same locations in memory. Fewer articles discuss issues in a distributed memory environment, where message passing is the cause of non-determinism.

Of the implementations of record-replay that treated distributed processing, [24] and [19] were among the first. In particular, they record the full content of all the messages exchanged in the system. Currently, a modern tool integrated into the TotalView debugger is ReplayEngine [6]. While these tools allow the full recording of the system, and its later deterministic replay, they all incur a high overhead during the recording phase, which might cause the problem to disappear.

The amount of data recorded during the execution of the program has always been of concern. In [17], the minimum amount of information necessary to replay the system is computed at runtime, and only this information is stored to disk. More recently, [25] has proposed to reduce the amount of data stored by grouping processors, and storing full content only for messages between processors in different groups. For processors within the same group, only the message ordering is stored. Both these approaches achieve a significant reduction in disk space usage when compared to full record techniques, but they still have a considerable overhead in the recording phase.

Our approach differs from previous ones by imposing a negligible overhead during the most time critical phase of the application, when the non-determinism is captured. In this phase, any overhead is a potential for a Heisenbug, and can make the bug disappear. We also succeed in minimizing the data stored by having a second recording phase, where only processors of interest are recorded in full detail. Even for very large executions with thousands, or potentially millions, of processors involved, our scheme will record the full content of the messages only for a few processors selected by the user.

The replay time has also been considered and analyzed in literature. In [26], checkpoint has been combined with record-replay to allow the replayed program to reach the failure point more quickly. In [18], the replay time has been further analyzed to provide an upper bound: the system automatically makes a tradeoff between checkpoint and recorded data to meet the user-specified replay time. Furthermore, these checkpoints have also been used to allow backward movements in time, such as in [2] and [7].

Instead of having the replayed system execute in the same manner as the original execution, the recorded traces have also been used to force a different ordering during replay [15, 14]. This gives another possible dimension in which to search for a bug. A challenge with this technique is to prevent the user, or automatic tool, from specifying an ordering that is not feasible in the real application. If this were to happen, the user may follow a path and correct bugs that are not real bugs, but just caused by the infeasible ordering of messages.

3. THE THREE-STEP PROCEDURE

The three-step record-replay procedure we are going to propose is based on the following two algorithms. The first algorithm is a non-intrusive record-replay technique that records (in memory) only the minimum amount of information necessary to eliminate the non-determinism from the application. In particular, this recording consists of the ordering in which messages are processed by each processor. Since the amount of information is minimal, special care needs to be taken to detect situations where the information recorded becomes insufficient for the correct replay of the system. A technique based on the computation of checksum of the received messages is presented in Section 5.

The second algorithm is a more intrusive one, and records the full content of each message processed by a selected set of processors. The generated output can be as big as a few gigabytes, and contains enough information to replay the recorded processor by itself as a stand-alone. Note that given the possibly high volume of data recorded, this recording is performed only on a subset of the processors specified by the user. Since this algorithm is more invasive, if used alone, it has the potential to disrupt the timing of message receipt between processors. By combining it with the first algorithm that records only the message ordering, we can obtain excellent results.

The three-step procedure that is based on these two algorithms is depicted in Figure 1. In the first step, the entire application is executed on the large target machine, and basic information about message ordering is recorded. Optionally, the user may decide to enable the self-correction feature provided by our technique. Note that this step is the only one that actually requires the use of a machine with as many processors as those required by the application.

For the second step, the programmer first identifies a set of processors to focus on. Good candidates are processors that crash, or processors generating incorrect output. The entire application is then executed again, and the selected processors are recorded in detail. During this second execution, the message ordering recorded from the first step is used. This guarantees the determinism of the execution, and the more intrusive recording necessary for processor extraction will not affect the bug appearance.

In the third step, the detailed traces recorded in step two are used to replay a selected processor using a single processor. This re-execution can happen either on the same machine where the traces were recorded, or on a local machine. The possibility to move to a local machine depends mainly on the compatibility between the architectures of the parallel and local machines. On the replayed program, the programmer can use traditional sequential debuggers

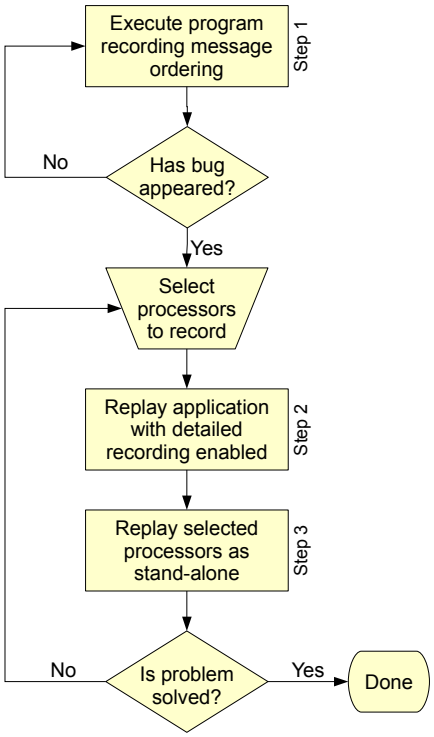


Figure 1: Flowchart of the three-step algorithm.

like GDB [4], and follow the problem in detail, re-executing it as many times as needed.

During the third step, the programmer may realize that some processor that has not been recorded is now needed. For example, he may realize that an extracted processor receives a corrupted message from a processor that was not recorded. By repeating step two, these missing processors may be extracted too. This establishes an iterative procedure that allows the programmer to identify an initial set of processors of interest, and expand this set later if necessary. Note that every time the second step is performed, the traces recorded during step one are used. Therefore, the same ordering of messages is guaranteed, and the processors extracted in different passes are compatible with the same manifestation of the bug under inspection.

As mentioned, of the three steps, only the first one actually requires a large machine to be used. Step three clearly requires only one processor to be allocated for the replay of an individual processor. As for step two, it can be performed using fewer physical processors than those needed by the user, by executing the application within a virtualized environment. This virtualized environment can still use the traces from step one to guarantee the deterministic replay of the application. Naturally, the time required by step two may increase as the application will have fewer computational resources available. The detailed traces generated in this virtualized environment can then be used in step three as before. We will discuss this in more detail in section 7.

Another important consideration regards optimizations performed by compilers. Generally, during debugging, the user's application must be compiled with debugging symbols, and without optimization. Debugging an optimized code can lead the debugger to not correctly correlate the generated assembly code with the original source code. On the other hand, optimizations can radically change the performance of a program, and in particular its timing. Oftentimes, an application that crashes when compiled with optimization enabled may succeed when no optimization is used. This creates a big problem for standard record-replay techniques during the recording phase, when timing is essential for the bug to appear.

In our approach, since the information recorded in step one is independent of the particular compilation, and depends only on the algorithm used, the user is allowed to switch between an optimized and a non-optimized code. In particular, he can use the optimized version in step one where timing is critical, and a non-optimized version in steps two and three. Since our message order recording scheme has a minimal impact on the application performance, as we shall see in section 8, using an optimized version greatly reduces the possibility of the bug disappearing.

When the application has to run for many hours before the bug appears, maintaining all the logs in memory during step one becomes impossible. To solve this problem, we have two solutions. One involves the application manually flushing the logs to disk at appropriate times, when the disk I/O does not disrupt the timing. As many scientific applications are iterative, and contain explicit barriers between iterations, adding one phase to flush the logs synchronously does not add significant overhead. Alternatively, the application can make use of the checkpoint/restart scheme available in CHARM++ [27] to automatically checkpoint and restart from a point in time closer to the problem. In this way, the total amount of log data each processor has to hold in memory is kept small.

4. BACKGROUND

We implemented the record-replay techniques proposed in the context of the CHARM++ runtime system, although the idea is general enough to apply to any other message-passing system. For example, our idea is trivially applicable to MPI applications by using the AMPI [8] implementation of the MPI standard.

4.1 Charm++

CHARM++ [13] is a C++ based parallel programming model based on object virtualization. In this approach [12], a programmer decomposes a problem into N migratable objects that will execute on P processors, where ideally $N \gg P$. The application programmer's view of the program is of migratable objects and their interactions; the underlying runtime system keeps track of the mapping of migratable objects to processors, and determines and performs any remapping that might be necessary at runtime.

In CHARM++, migratable objects are known as *chares*. Chares are C++ objects with special *entry* methods that are invoked asynchronously from other chares via messages. These entry methods may also be executed inside user-level threads. This allows an entry method to suspend execution, and resume it at a later time. A special keyword, "threaded", is reserved for this purpose. The order in which entry methods are invoked, and threads are scheduled, is determined by a processor-level scheduler, and by the priority of each message (which can be set by the sender).

The processor-level scheduler, also called CONVERSE scheduler, implements an infinite loop that examines different message queues in the system, and determines the order of execution. These queues are: (a) a network queue, which contains messages coming from other processors via network; (b) a node level queue that contains messages from other processors on the same SMP node; and (c) a local queue, which contains messages that objects on a processor send to other objects on the same processor. The messages from these three queues are combined together, and then messages are scheduled according to their priority.

4.2 Ordering Messages

Messages may arrive from the network in any order, and they are placed in the network queue in the order they arrive. Messages sent from the local processor will be picked up by the CONVERSE scheduler sooner or later depending on the presence of messages in the network queue. As a result, race conditions between messages may occur, and this can lead to hard-to-find application bugs. Therefore, in order to capture the parallel behavior of an application for debugging purposes, it is important to record the message ordering.

A simple deterministic record and replay scheme has been available in CHARM++ for several years [11]. This scheme is based on the assumption of piecewise deterministic execution [23]:

Definition 1. Let *obj* be an object in the system with associated state *s1*, and *msg* a message sent to *obj*. Suppose the processing of *msg* by *obj* causes the state of *obj* to transition from *s1* to *s2*, and a set of messages *M* to be sent. Then, if we deliver the same message *msg* to the object *obj* in state *s1*, the object will always transition to state *s2*, and will always send the set of message *M*.

Under this assumption, the only source of non-determinism in the application is the order in which messages are processed. Therefore, by recording a tuple containing the sending processor and a per-processor unique sequence number, the system can be replayed deterministically by re-ordering messages according to the recorded sequence. In addition to this tuple, the original scheme also saved the size of the message as a simple check to make sure the messages processed are indeed the same between executions.

5. A ROBUST AND ACCURATE MESSAGE ORDERING RECORD/REPLAY

The scheme for recording the message ordering in CHARM++ applications can be used in the first step of our proposed three-step procedure. However, it has several problems regarding robustness and accuracy. One limitation is the assumption made about piecewise deterministic behavior. Although in general this condition should hold, some applications may not entail such determinism. For example, the application may use timers. Imagine the scenario where an entry method receives a message and, depending on the elapsed time since last invocation, performs different operations, possibly sending different messages. In this scenario, a different timing in the network, maybe due to a sudden congestion, can modify the behavior of the application, even if the same ordering of messages is maintained.

Capturing system-level calls, like timers, may solve the problem for the given example. By following this path, many other system calls, along with their return values, need to be included in the recorded traces. Particularly voluminous may be those reading from files. The complexity and amount of data to be stored would rapidly increase in this scenario. This would lead to an increase of the overhead incurred by the recording scheme, making it more likely to disrupt the precarious timing which leads to the manifestation of the bug. Therefore, to maintain the overhead to the minimum, we do not record these additional information, and limit the applicability of our technique to piecewise deterministic applications. Fortunately, the vast majority of applications do oblige to the piecewise deterministic assumption, and for these applications, the simple message ordering is sufficient.

The key now is to understand when the piecewise deterministic assumption is satisfied by the application, and to detect when it is not satisfied. The original scheme tried to do this by using only the message size. Unfortunately, most applications tend to have many messages with the same size, yet completely different content. In order to guarantee the piecewise determinism, in addition to the message ordering, we would like to include also the content of the messages. By having available the full content of the messages, it would be trivial to determine if all the messages processed during the replayed execution are identical to those processed during the recorded execution.

Before continuing, a word of caution is in order. Even by extending the recording to assure that communicated messages have the same content, we cannot prevent a processor from having internal non-determinism completely. For example, the processor may still make a decision based on the current time, and modify its local state differently in different executions. However, since by definition all outgoing messages are the same between two executions, and the only way to influence another processor is through explicit message passing, the local non-determinism cannot propagate to other processors. Therefore, while searching for a bug, the causal relationship between processors does not change.

Clearly, having the receiving processor store (in step one) the whole content of each message received would defeat the purpose of our 3-step procedure which aims at a non-intrusive mechanism during the first step. Instead, we compute the checksum of the received messages, and store only this information into the recorded traces. The amount of data added by the checksum is only a few bytes per message, therefore adding little overhead, as we shall see in section 8. Of course, our technique can only capture a difference in the message content with high probability. If the content of a message in two different execution is such that the computed checksum is identical, then our method will fail to detect the change. Nevertheless, since this check is performed for every message processed, the probability that the non-determinism will remain latent as the application progresses is extremely low.

We implemented two commonly used checksums. Both of them produce a 32-bit integer value. The first is a simple XOR of the message data, reading four bytes at a time. This checksum is fast to compute, but has the disadvantage that it is easy for a message to contain differences not detected. The other is a more sophisticated Cyclic Redundancy Check (CRC32) checksum. This is a more computationally intensive algorithm, but can capture difference in the transmitted data with higher confidence. The programmer, during step one, can choose which of the algorithms to use. He can also choose to altogether skip this checksum computation to minimize the overhead.

In order for the checksum computation to yield correct results, it is important that the message content of each message is identical between the two executions. One problem is posed by the presence of garbage inside a message. Consider a data structure like that in Figure 2. In this case, the compiler has padded the data to maintain correct alignment of the data structures, in this case of the double type. When allocating this data structure, the padded memory region, which is shown in the light color, is not initialized, and may contain any random garbage. To overcome this, we developed a solution that makes use of the memory allocation sub-system present in CHARM++, and make sure every newly allocated memory is always initialized to a known pattern.

In the memory sub-system implemented in the CHARM++ runtime system, there is an interface to easily re-implement memory related functions, such as malloc and free, and place them into a dynamically loadable library. There are three types of re-implementation. One uses the `glibc` memory arena internally, and wraps it with the new function definitions. The other two are based on a direct usage of the memory allocator provided by the operating system. This is done either by dynamically loading the

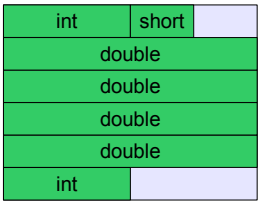


Figure 2: Example of data structure padded by the compiler.

specific function pointers using `dlopen`, or by using the hooks present in the operating system in the case of the GNU implementation. For our purposes, we extended an implementation created specifically for debugging. This implementation can use all the methods described above to link to the underlying memory system, making it very portable. In this extension, we could easily add a call to `memset` before returning the allocated memory to the user. Note that the memory can be pre-filled with any known pattern.

Another problem that we encountered while using the original record-replay scheme in CHARM++ was the lack of ordering of threaded entry methods. As mentioned in section 4.1, CHARM++ programs may declare an entry method as "threaded", thus creating a user-level thread for the execution of each invocation of the entry method. Since thread operations, such as suspend and resume, are treated at the lower level of the runtime, the record-replay module was not aware of them. This produced a lack of recorded information for which threaded entry methods could be executed, and resumed, without a specific ordering with respect to the other entry methods. This clearly was a problem. To solve it, we placed a hook in CONVERSE's threaded library and exposed the occurrence of thread events to the record-replay module. This allowed these events to be properly logged.

With the enhancements described above, the new record-replay scheme has become a much more robust and accurate solution for record and replay. We understand that other problems may arise in the future, both from the introduction of new features in CHARM++ and from old features not properly treated. Should new issues arise, we plan to expand the recording scheme accordingly, as we have done for threaded entry methods.

6. PROCESSOR EXTRACTION

In step two, once the information we want to record is identified—all the messages received by a processor—the extraction is relatively simple. All we have to do is to record the content of the messages processed by the user-selected processors into a file, so that in step three a modified CHARM++ runtime system can replay the execution of any selected processor as a stand-alone. When replaying a processor, the CHARM++ runtime system loads messages from the corresponding trace file instead of receiving them from the network, and since the processor is replayed without a network, all outgoing messages are discarded. More important is considering the implications involved with re-executing the application.

As we mentioned at the end of Section 3, we can always change the executable used between steps one and two, provided the application performs the same operations—for example by changing the optimization level. On the other hand, the information recorded in step two contains information that may be specific to a particular executable, and may change if the application is recompiled. For example, in ChaNGa, several function pointers are sent through messages between processors. During normal execution, this is acceptable since the executable is the same for all processors. Nevertheless, recompiling the code means changing the placement of these function pointers, and thus invalidating the data contained in the messages. For ChaNGa, therefore, the same executable must be used in steps two and three. For other applications, like Jacobi, where the content of the sent messages will not change even if the application is recompiled, the optimized version of the code may still be used in step two.

Another problem appears if the two executable are compiled for different versions of CHARM++. Changing architecture would be desirable if the architectures of the parallel machine and of the local machine differ (say one used LAPI, the other pure ethernet). Unfortunately, this is not possible at the moment. In CHARM++, a different architecture signifies a different header added to each message. These headers are not only different in content, but also in size. Furthermore, the content of the message stored will follow the endian-ness convention of the machine where the data was recorded, and translating messages to a different architecture is not simple.

7. REDUCING THE NEED FOR LARGE MACHINES

In our proposed three-step procedure for recording and replaying a buggy application deterministically, the second step may be performed inside a virtualized environment. As we mentioned in section 3, this is very useful to reduce the number of physical processors needed, and to reduce the contention on the availability of a large machine.

This is particularly important when the bug appears only on large processor counts. In this case, executing multiple times the buggy application on a large parallel machine to extract different sets of processors may introduce long delays in the debugging process. This can easily happen when submitting jobs to a batch schedulers on heavily used machines. By using a virtualized environment, instead, we can perform the processors extraction operation using a much smaller machine, and increase productivity.

7.1 BigSim Emulator

To demonstrate the feasibility of this approach, we used the BigSim Emulator [28]. This is a part of the BigSim simulation framework [29], a framework which provides fast and accurate performance evaluation of current and future large parallel systems with different levels of fidelity using much smaller machines. It targets systems composed of possibly hundreds of thousands of multi-core nodes, including petascale-level machines. BigSim supports the emulation/simulation of applications written both in CHARM++ and in MPI (via the AMPI implementation of the standard).

BigSim consists of two components. The first component is a parallel *emulator* that provides a virtualized execution environment for parallel applications. This emulator generates a set of event logs during execution. The second component is a post-mortem trace-driven parallel *simulator* that predicts parallel performance using the event logs as input, and supports multiple resolutions for prediction of sequential and network performance. For example, the simulator can optionally predict communication performance accurately by simulating packets of each message flowing through the switches in detail, using a parallel discrete event simulation technique. Since the simulator only considers the trace logs and does not re-execute the application at the code level, it is not suitable for debugging purposes in this paper. However, the BigSim Emulator, which supports emulation of a very large application using only a fraction of the target machine, is useful for debugging. In the next section, we shall focus our attention on the emulator component.

7.2 Detailed Record-Replay in BigSim Emulator

We added support in the BigSim Emulator for the detailed record-replay scheme that stores the full content of messages similar to the one used in the processor extraction described earlier, but under the virtualized environment. When emulating an application, the user may specify a subset of processors that he wants to record in detail. During the emulation, on each of these emulated (or target) processors, the scheduler stores a copy of the message to its own trace file before it executes the entry function associated to that message.

We incorporated the BigSim Emulator's record-replay capability into the proposed three-step procedure as an alternative to reduce the need for large machines in the second step. This new three-step scheme thus becomes: (1) execute an application on a big machine, and record the message ordering; (2) replay the application on a machine emulated under the virtualized environment and record the detailed traces; (3) replay the execution of a selected target processor sequentially. Note that if step two was performed within the emulated environment, step three must also be performed in the same environment. This comes from the fact that BigSim Emulator is considered by the application as another communication layer, and at the moment we cannot change this layer between step two and three. Nevertheless, we are considering extending the possibility to perform step three in a different scenario (say outside BigSim, or ethernet vs. LAPI).

In this new scheme, the emulator needs to be able to read the trace logs generated in the first step from the non-emulated execution on a full machine, and replay the application in the emulator using a small machine. One challenge was to match the two executions of the application on these two totally different environments. Specifically, when the trace log tells that the next expected message is (srcpe, msgID), where *srcpe* is the source processor ID, and *msgID* is the message ID of that message, the message IDs must be identical on the two environments. This is easy to guarantee as long as the emulator emulates the CHARM++ runtime faithfully and both systems assign msgID to each message using a sequence number local to its sender processor. However, this becomes rather complicated when user-level threads are involved in an application. This is because emulated processors themselves are implemented as the same user-level threads. Therefore, tracing the suspend/resume events of user-level threads will mistakenly record the events of the emulator threads, creating mismatch of the thread event IDs between two executions. One way to handle this is to recognize two different categories of threads in the emulator – those created by the emulator system, and those created by the application, and ensure that only the events of threads that are created by the application are tracked. To do this, we used CONVERSE user-level thread API which allows a user to insert hooks to the thread scheduling events such as at the time of suspend and resume. When the emulator creates user-level threads for the application, it sets up special record-replay hooks for these threads that track thread suspend and resume in the same way as how it is done in the non-emulated CHARM++. When it creates internal user-level threads, it does not set these hooks.

In Section 8.2.2, we will demonstrate how the BigSim Emulator is useful in reducing the number of processors using a real world application.

8. PERFORMANCE

We evaluated the overhead of the proposed record-replay scheme for all three steps of the procedure. We used synthetic benchmarks, as well as two real applications.

8.1 Synthetic Benchmarks

Our first test environment was Abe cluster, at the National Center for Supercomputing Applications (NCSA). This is a cluster of 1200 Dell PowerEdge 1955 server computers, each configured with two quad-core Intel Xeon processors, 8 gigabytes of memory, and infiniBand interconnect.

We tested a synthetic benchmark program called kNeighbor, and evaluated the overhead imposed by recording message orderings with and without checksums. KNeighbor creates a certain number of objects distributed on the parallel machine, and arranged in a 1-dimensional array. In each iteration, each object sends $2 * K + 1$ messages to its nearest K neighbors on each side, plus a message to itself. When an object receives $2 * K + 1$ messages, it performs a given amount of computation, and proceeds to the next iteration. In the following experiments, we used $k = 2$, and the total number of iterations was 100.

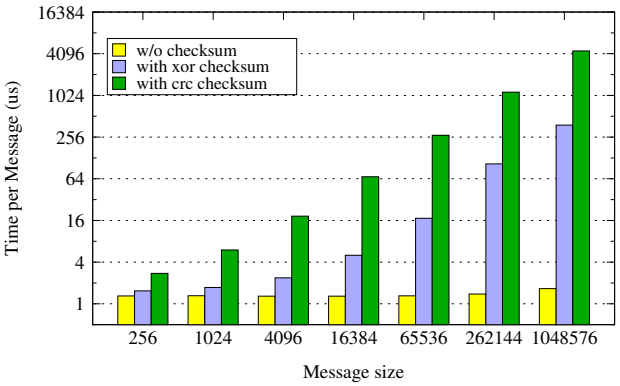
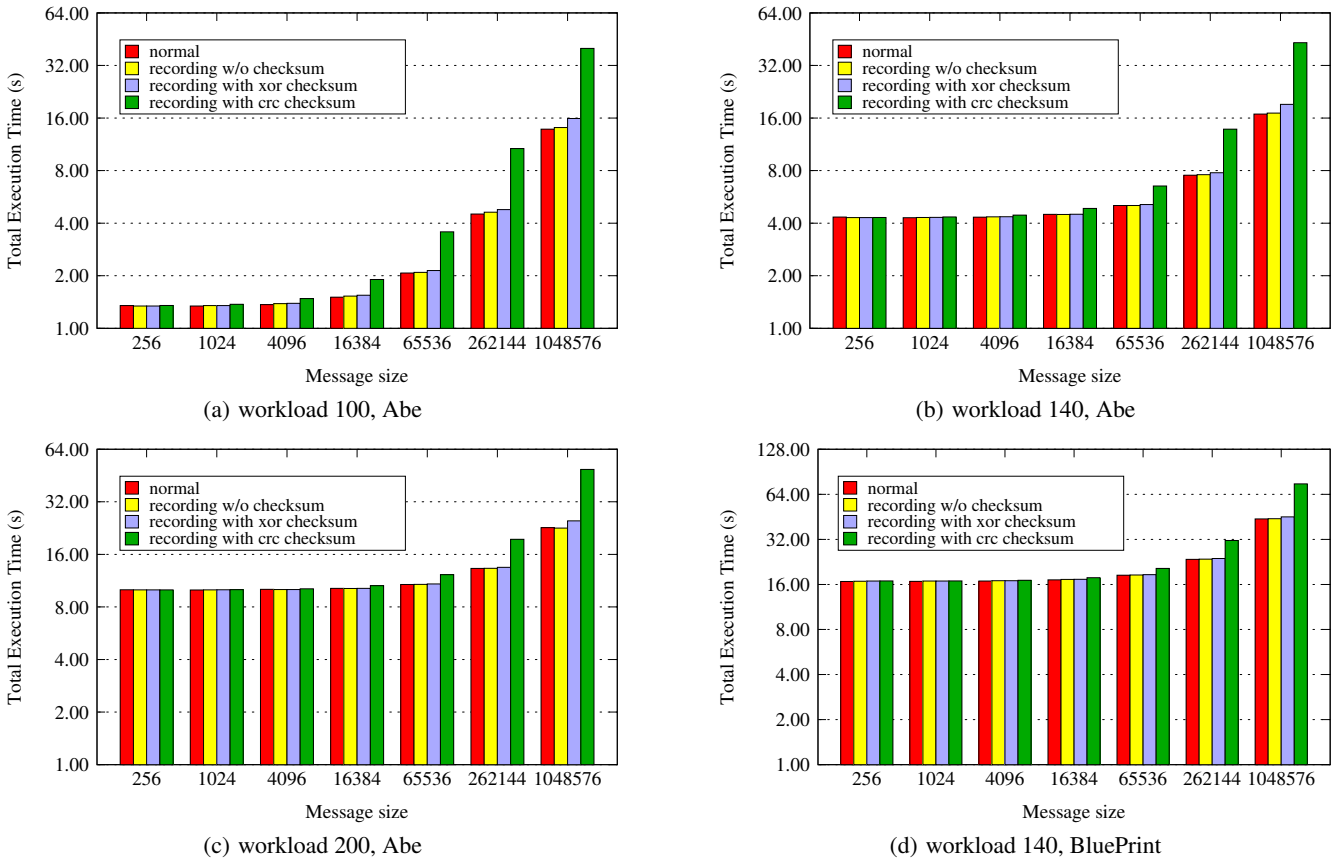


Figure 3: Recording overhead per message using the three schemes. Computed using kNeighbor test (NCSA Abe cluster).

First, we measured the average overhead per message during the recording phase in step one, varying the message size from 256 bytes to one megabyte. The results are shown in Figure 3. As expected, we can see that the overhead of the simple scheme remains about the same regardless of the message size. This is because, for each message, a constant amount of data is stored. When either XOR checksum or CRC checksum is calculated for each message, the overhead per message increases proportionally to the increase in message sizes. This is because the runtime needs to traverse the whole message in order to compute the checksum. When checksums are computed, for very small messages, specifically 256 bytes, we observe only less than three microsecond overhead per message. However, when message size increases to one megabytes, both checksum methods incur a much higher overhead per message.

Next, we evaluated how this overhead affects the total execution of the program. To measure the total execution time, we ran the test program on a single processor. The total number of messages generated during each execution is about 5600. Again, we tested with a varying message size. Figure 4(a) shows the results of the comparison when the workload is very small. We can see that when the message size is small, 256 bytes, the total execution time is only 1.36 seconds for 100 iterations. When message size increases, the total execution time increases proportionally. This is due to the fact that the program has to process the message, and traverse all the data in it. We see that doing simple recording, without checksum, the execution time is not affected, even for large messages.



(a) workload 100, Abe

(b) workload 140, Abe

(c) workload 200, Abe

(d) workload 140, BluePrint

Figure 4: Comparison of kNeighbor total execution time with and without recording schemes (the total time includes file I/O at the end of execution).

Even with XOR checksum enabled, there is not much overhead. When switching to the more expensive CRC-based checksum, we can observe a significant overhead for large messages. However, for message sizes below 4KB, the overhead is still minimal.

When we increased the workload in Figure 4(b) (by 3 fold), and Figure 4(c) (by 6 fold), we observe a similar behavior. However, the results exhibit a decreasing overall affect of the CRC checksum computation, mainly due to the increasing computation-to-communication ratio. Similar results were also obtained on a different machine, called BluePrint, as shown in Figure 4(d). BluePrint is a Blue Waters [16] interim system also at the National Center for Supercomputing Applications (NCSA). It is a 2000-core IBM Power 5+ system.

These experiments show that simple recording scheme performs very well with almost no overhead to the execution time. XOR-based checksum is a cheap solution to improve the robustness of our scheme, and it incurs very little overhead. The more expensive CRC checksum computation indeed adds a significant overhead for very large messages. However, since most applications do not send large messages often, and if they do they generally perform a large computation thereafter, we believe this is not a problem for real-world applications.

To study the performance of the second step, we ran the kNeighbor benchmark on 256 processors of BluePrint. The results are illustrated in Figure 5. In each cluster in the figure, the first bar from the left is the total execution time without any overhead; the second bar represents the execution time when replaying on the same machine using the traces from step one; while the third bar represents the execution time of the benchmark both replaying the previously recorded message ordering, and recording the detailed information for one processor. We see that replaying on the full machine generally is slightly slower than the normal execution time for message sizes smaller than 64K bytes. However, for very large messages, the replay time tends to increase more drastically.

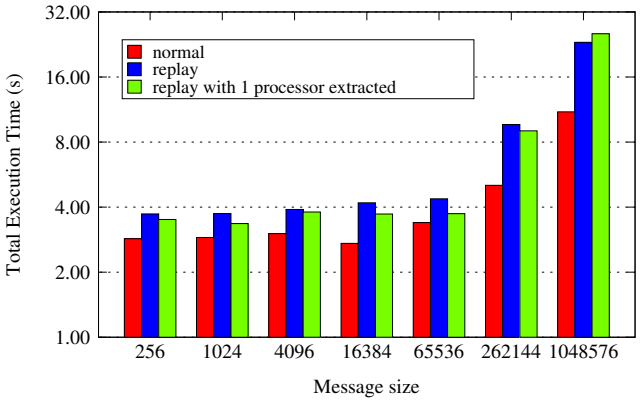


Figure 5: Total replay time in step two for kNeighbor (NCSA BluePrint 256 processors).

The number of processors recorded in detail during step two may

affect the replay time due to file I/O. Figure 6 illustrates the effect on the total execution time of kNeighbor when varying the number of recorded processors during step two on 256 processors. We can see that for messages smaller than 1KB, the effect of recording full traces is minimal. As expected, when the message size increases, the overhead of recording message contents for more processors increases significantly due to the file system becoming a bottleneck. However, as we explained, this does not affect the correctness of the replay, since the message ordering is already guaranteed. In practice, we believe that a user does not need to extract all processors, and only a small subset of processors is usually enough to understand the nature of a bug.

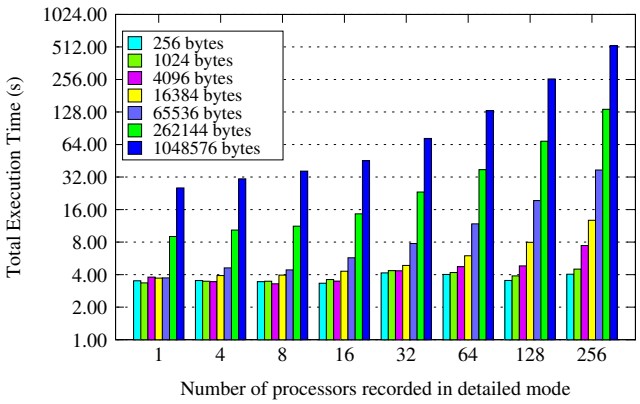


Figure 6: Comparison of kNeighbor total execution time in step two when recording varying number of processors in full detailed mode. (NCSA BluePrint 256 processors).

8.2 Scientific Applications

In addition to synthetic benchmarks, we used two production-level scientific applications to show the performance impact of our approach. The two applications are ChaNGa and NAMD.

8.2.1 ChaNGa

ChaNGa [9] is a cosmological application used for the simulations of the evolution of the universe. It handles forces generated by both gravitational and hydrodynamic interaction. The benchmark we used is a snapshot of a multi-resolution simulation of a dwarf galaxy forming in a $28.5Mpc^3$ volume of the universe, with 30% dark matter and 70% dark energy. The dataset size is nearly five million particles, with most of the particles clustered in the center of the simulated volume. In our tests, we ran the application for three timesteps.

Figure 7 shows the performance of ChaNGa in step one, using a varying number of processors, on the NCSA BluePrint cluster. Each execution was repeated five times, and the average and standard deviation are plotted. As mentioned earlier, we could run the optimized code for step one of our 3-step procedure. As a comparison, the black bar (to the left) represents the execution time with a non-optimized version. The optimized code is more than twice as fast as the non-optimized one, and the interleaving of messages potentially very different. It can be seen that even on a highly optimized code the impact of the recording schemes is so small that it disappears when compared to the normal time fluctuation of ChaNGa, even when computing checksums. This re-emphasizes the negligible perturbation caused by our recording scheme.

Subsequently, with the recorded data from step one, we proceeded to test steps two and three. These are plotted in Figure 8.

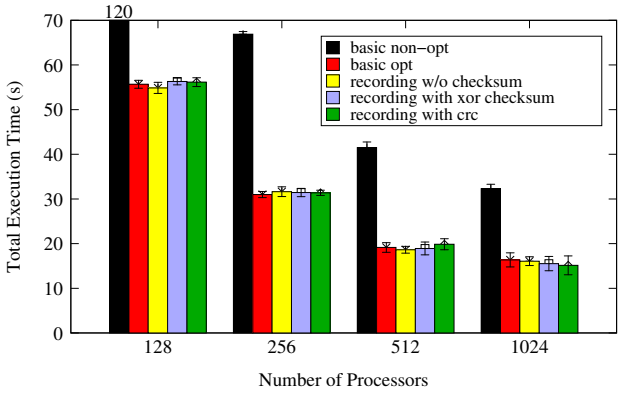


Figure 7: Recording overhead for ChaNGa application using the three schemes (on NCSA BluePrint cluster). The tests were performed with optimized code, except for the first bar in black.

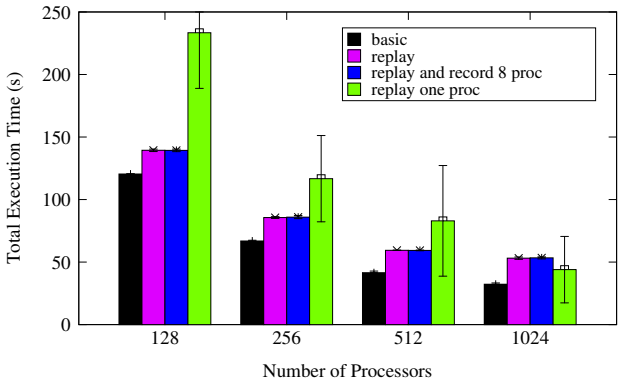


Figure 8: Overhead during replay in steps two and three for ChaNGa application (on NCSA BluePrint cluster).

Again, each execution was repeated five times. In this case, we had to use the non-optimized version to be able to follow the code in a sequential debugger. Compared to the execution without record-replay enabled, the forced replay of the message ordering caused an overhead between 25% and 65%. This overhead is still very small considered to the potential that the scheme yields in terms of allowing a deterministic debugging.

Step three is represented on the fourth bar of each cluster in Figure 8. As different processors may have a different workload (we didn't apply load balancing), the variation in the execution time between different processors is very large. Surprisingly, the execution time of a single processor was greater than the time to execute the whole application. We suspect this might depend on the system pre-loading too many messages from the traces, and we plan to further investigate the reasons. Nevertheless, even with the current performance, the replay time is within a factor of two from the basic execution.

In addition to the overhead caused during the execution of the application, we also measured the amount of information that is stored to disk during the various phases. Table 1 reports this information in megabytes. As it can be seen, the amount of information recorded per processor is quite small—less than one megabyte—

and can be easily maintained completely in memory until the application shuts down. Therefore, flushing to disk is generally avoided during the first step. During the second step, we can see that the amount of data recorded is much larger. Nevertheless, this does not create a problem since usually only few processors are recorded in detail.

Number of processors		128	256	512	1024
Record	**per-proc.**	**0.87**	**0.67**	**0.54**	**0.44**
	total	112	173	279	453
Record+checksum	**per-proc.**	**1.49**	**1.14**	**0.92**	**0.75**
	total	190	292	473	765
Detailed record	**per-proc.**	**111**	**79**	**59**	**47**

Table 1: Amount of data stored to disk using different recording schemes for ChaNGa. All data in megabytes.

8.2.2 NAMD on BigSim Emulator

In this section, we demonstrate the utility of using BigSim to perform processor extraction using the 3-step procedure. The application we chose for this is NAMD [1, 20]. NAMD is a scalable parallel application for Molecular Dynamics simulations written using the CHARM++ programming model. It is used for the simulation of biomolecules, and to understand their function. The following experiments were done on the NCSA BluePrint system.

First, we benchmarked step one by running NAMD on 1024 processors using the Apolipoprotein-A1 (ApoA1) benchmark for 100 timesteps, and repeating using all the recording schemes. The results are shown in Table 2. The "normal" column is the time when running without recording for comparison. Note that the total number of messages processed during the entire execution is about 20,000.

Mode	Normal	Record	Rec.+XOR	Rec.+CRC
NAMD Time	24.08	25.33	24.55	24.55
with I/O	-	27.99	26.85	25.82

Table 2: NAMD execution time in seconds with different recording schemes in step one, running on 1024 processors of NCSA BluePrint. The last row is the total time with file I/O.

We see that there is virtually no overhead to the NAMD actual execution while recording the message ordering, even when checksums are computed. This is because in NAMD the average message size is relatively small, around 1KB to 2KB. For 20,000 messages total, even the most expensive scheme using CRC checksum only cost about 0.27 second. Therefore, we believe that by using our recording schemes, the NAMD application behavior is not affected significantly. Furthermore, due to cache effects, the actual overhead of computing checksum may be even less, if the entry function triggered by the receipt of the message has to traverse the message data immediately. Similar to ChaNGa, the NAMD traces recorded for each processor are less than one megabyte in size. The process of flushing the traces to disk takes about 2 seconds, which increases the total execution time, as shown in the second row of the table. The file I/O time is constrained by the bandwidth of the file system, and may be stressed by simultaneous writing, in this case by 1024 processors. However, since this is done only at the very end of the execution, it does not affect the ordering of the messages during execution.

For the second step, we ran it under the BigSim Emulator, and we replayed NAMD using the message logs obtained from the first

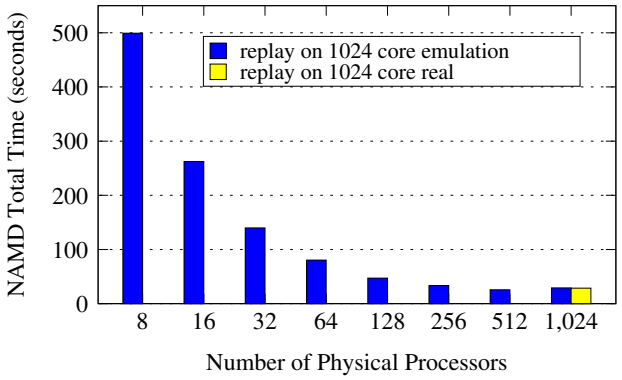


Figure 9: NAMD execution time in replay mode on 1024 emulated processors using varying numbers of physical processors, recording 16 emulated processors. The last bar is the actual runtime in the non-emulated replay mode on 1024 processors. (on NCSA BluePrint).

step. We instructed the emulator to emulate the same 1024 processors by using only a portion of the entire machine. While replaying, we also chose 16 emulated processors for detailed recording of message content. We measured NAMD total execution time running on the emulator using varying number of physical processors. The results are shown in Figure 9. When 1024 physical processors are used to emulate the 1024-processor machine, we see that replaying NAMD on the emulator is about as fast as when replaying it on the real 1024 processor BluePrint machine, showing little overhead of the emulator. Moreover, on 512 processors, NAMD replaying in the emulation mode is even slightly faster than the actual replaying run on 1024 processors. This is due to the saving in the startup: faster global synchronization on fewer nodes.

This demonstrates that in terms of the time-cost, it is feasible to replay an application in a virtualized environment under the emulator using fewer processors. Although it takes much longer (17 times slowdown) to replay NAMD under the emulator when using only 8 physical processors, being able to replay an application on a much smaller machine, and generate detailed trace logs, greatly reduces the need for large machine during interactive debugging.

In the third step, the detailed NAMD trace logs recorded in the second step were used to replay a selected processor using a single processor. Figure 10 shows the last few lines of screen output of replaying processors number 0 and 960 on the emulator respectively. For this benchmark, each detailed trace log was about one to two megabytes, as shown in the output. The replay time of processor 0 on the emulator finished in about 23.8 seconds, which matches the total execution time of 24 seconds when running NAMD in normal parallel execution. The replay time of processor 960, however, took much less time (only 5.5 seconds). This is because during NAMD start-up, most of the work is done by processor 0, and the other processors are mostly idle. Since this is a short simulation that has only 100 timesteps, most time was spent in the start up. On 1024 processors, the start-up time is measured around 19 seconds.

In summary, this example of NAMD on the BigSim emulator demonstrates that it is feasible to use an emulator in the second step as an alternative way of replaying an application using the message ordering logs obtained from the previous step, and producing detailed trace logs to be used in the third step. This approach incurs reasonably low overhead, and the overhead itself can be considered

```
BgReplay> Emulation replay finished at 25.304625
due to end of log.
BgReplay> Replayed 12288 local records and 7891
remote records, trace log is of 14539488 bytes.
```

(a) Processor 0

```
BgReplay> Emulation replay finished at 5.690778
due to end of log.
BgReplay> Replayed 19714 local records and 10822
remote records, trace log is of 24904148 bytes.
```

(b) Processor 960

Figure 10: Screen outputs from replaying two different processors under the emulator.

```
../charmrun +p16 ../ChaNGa cube300.param +record +recplay-crc
../charmrun +p16 ../ChaNGa cube300.param +replay +recplay-crc +record-detail 7
gdb ../ChaNGa
» run cube300.param +replay-detail 7/16
```

Figure 11: 3-step procedure used for debugging ChaNGa.

proportional to the reduced number of physical processors used for emulation.

9. CASE STUDY

To assess the usability of our technique, we used the ChaNGa application, and searched for the bug we mentioned in the introduction. This bug has already been fixed using standard techniques, such as print statements, and a tedious process given that the bug often disappeared after code modifications. We re-introduced it in the application temporarily. We ran the application using a relatively small dataset (a simulation of a LCDM concordance cosmology large volume with 48^3 particles and 300 Mpc on a side). The bug did not appear on eight processors, but started to appear on sixteen processors or more. The manifestation was intermittent, sometimes right at the beginning, sometimes after a few timesteps of the application. Also, the processor in which an assertion failed kept changing from execution to execution.

According to our 3-step procedure, we first executed the application with the message ordering recorded. We used CRC checksum as robustness protection. In the execution we recorded, processor seven triggered the assertion. At this point we re-executed the application in replay mode, and recorded the faulty processor. We also repeated the execution in replay mode a few times to confirm that processor seven was always the culprit. With the detailed trace of processor seven, we executed ChaNGa sequentially under GDB, and followed the problem. To track the bug we had to repeat the sequential execution a couple of times, each time setting a few different breakpoints in the code. Compared to the way the original bug was hunted, this new procedure allowed for the parallel problem to be transposed into a sequential one, without compromising the timing of the application, and without allowing the problem itself to disappear. The commands we used in the different steps of our analysis are reported in Figure 11.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we described a new procedure to extract processors from a parallel application, and replay any of them on a local cluster. This procedure is based on an extension of record-replay techniques, and it generates minimal interference in the application while capturing the non-determinism in the application, as we have seen both with benchmarks and real applications. This was possible thanks to the separation of the recording phase into two sub-phases: the first one were only minimal information—message ordering—is collected, and the second one where more information—full content of the messages—is recorded for a few selected processors. During the execution, checksums are also used to assert that the mechanism has really captured the non-determinism of the application. We have also seen that our approach does not require the use of large machines in steps other than one, since processor extraction can be performed using an emulated environment. Moreover, in step one, the code used can be fully optimized by the compiler to avoid a change in the message interleaving.

We have several research directions for the future. One regards the possibility to use different executables for steps two and three. In particular, the possibility to change the underlying communication layer of CHARM++. Another direction is to detect other information that our current scheme is lacking, and include that into the recorded traces. For example, the creation of single chares in the system may not be fully deterministic if a chare is inserted in an underloaded processor. When replaying the system, these chares will have to be inserted in the same processors as the original execution.

The record-replay scheme presented is stand-alone, and is not tied to any particular debugger. For step three, while doing in-depth debugging, the user can choose any sequential debugger. In CHARM++, the de-facto standard debugger is CHARMDEBUG [5]. This parallel debugger can show the user information pertinent to the CHARM++ programming model, such as messages in queue and chare objects. While an expert user can retrieve this information also using a common sequential debugger, CHARMDEBUG can greatly simplify the lookup of relevant information. The simple scheme of recording the message ordering, including the checksum contribution, can already be used with CHARMDEBUG seamlessly. On the other hand, CHARMDEBUG can not be used in step three, as the application is run sequentially without the parallel infrastructure. We are planning to incorporate also this step into the CHARMDEBUG debugger, and allow the user to conduct his debugging as if he was running the full application.

11. REFERENCES

[1] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[2] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM.

[3] C. Dionne, M. Feeley, and J. Desbiens. A Taxonomy of Distributed Debuggers Based on Execution Replay. In *In Proceedings of the 1996 International Conference on*

Parallel and Distributed Processing Techniques and Applications, pages 203–214, 1996.

[4] Free Software Foundation. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[5] F. Gioachin, C. W. Lee, and L. V. Kalé. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid'09*, Arlington, VA, USA, June 2009.

[6] C. Gottbrath. Quickly Identifying the Cause of Software Bugs with ReplayEngine. Technical report, August 2008.

[7] C. Gottbrath. Reverse Debugging with the TotalView Debugger. In *CDROM*. Cray User Group Conference 2008, May 2009.

[8] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[9] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[10] X. Jiao, G. Zheng, P. A. Alexander, M. T. Campbell, O. S. Lawlor, J. Norris, A. Haselbacher, and M. T. Heath. A system integration framework for coupled multiphysics simulations. *Engineering with Computers*, 22(3):293–309, 2006.

[11] R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[12] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[13] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[14] R. Kilgore and C. Chase. Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages. In *In Proceedings of the International Conference on System Sciences*, page 423, 1997.

[15] D. Kranzlmüller, C. Schaubschläger, and J. Volkert. A Brief Overview of the MAD Debugging Activities. In *AADEBUG*, 2000.

[16] National Center for Supercomputing Applications. Blue Waters project. http://www.ncsa.illinois.edu/BlueWaters/.

[17] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Minneapolis, MN, November 1992.

[18] R. H. B. Netzer, S. Subramanian, and J. Xu. Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs. Technical report, Providence, RI, USA, 1994.

[19] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Not.*, 24(1):124–129, 1989.

[20] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[21] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay and debugging. In *Automated and Algorithmic Debugging*, 2000.

[22] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[23] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[24] L. D. Wittie. Debugging distributed C programs by real time replay. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):57–67, January 1989.

[25] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. MPIWiz: Subgroup reproducible replay of MPI applications. In *In PPoPP*, 2009.

[26] F. Zambonelli and R. H. Netzer. An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications. In *In Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 392–398. IEEE, 1999.

[27] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing*, pages 93–103, San Dieago, CA, September 2004.

[28] G. Zheng, A. K. Singla, J. M. Unger, and L. V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.

[29] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.