DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements

Qi Gao Feng Qin Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
{gaoq,qin,panda}@cse.ohio-state.edu

ABSTRACT

While software reliability in large-scale systems becomes increasingly important, debugging in large-scale parallel systems remains a daunting task. This paper proposes an innovative technique to find *hard-to-detect* software bugs that can cause *severe* problems such as data corruptions and deadlocks in parallel programs automatically via detecting their abnormal behaviors in data movements. Based on the observation that data movements in parallel programs typically follow certain patterns, our idea is to extract data movement (DM)-based invariants at program runtime and check the violations of these invariants. These violations indicate potential bugs such as data races and memory corruption bugs that manifest themselves in data movements.

We have built a tool, called DMTracker, based on the above idea: automatically extract DM-based invariants and detect the violations of them. Our experiments with two real-world bug cases in MVAPICH/MVAPICH2, a popular MPI library, have shown that DMTracker can effectively detect them and report abnormal data movements to help programmers quickly diagnose the root causes of bugs. In addition, DMTracker incurs very low runtime overhead, from 0.9% to 6.0%, in our experiments with High Performance Linpack (HPL) and NAS Parallel Benchmarks (NPB), which indicates that DMTracker can be deployed in production runs.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Distributed debugging*

General Terms

Reliability, Design, Experimentation

Keywords

Bug Detection, Parallel Programs, Data Movements, Anomaly Detection

SC07 November 10-16, 2007, Reno, Nevada, USA

1. INTRODUCTION

Software bugs greatly affect the reliability of high performance systems. A recent study [52] with more than 20 different high-performance computer systems at Los Alamos National Laboratory has shown that software bugs account for as many as 24% of failures. Furthermore, software bugs may silently corrupt application data and remain unnoticed until termination of the whole task, which leads to incorrect results and significantly affects overall productivity. With peta-scale and many-core architectures becoming a mainstream in High Performance Computing (HPC) systems in the predictable future, software will become much more complex and software bugs may cause more frequent and more severe system problems.

Unfortunately, finding software bugs in high-performance systems is a daunting task due to the systems' inherent nature of non-determinism and large scale. Bugs such as data races manifested during one execution may not be triggered during another execution because of various non-deterministic events, such as process execution orders, thread interleaving, signal delivery timing, I/O events, etc. Such non-determinism makes it difficult to reproduce bugs and thus renders a significant challenge for detecting and locating a software bug. Furthermore, the problem becomes much more complicated due to the ever-increasing scale of high performance systems. For example, some software bugs can only be triggered in very large-scale systems. However, it would cause a huge resource waste if developers have to occupy the whole large-scale system for inefficient manual debugging. Therefore, it is imperative to have low-overhead tools deployed in production runs to help automatically locate software bugs.

Previous work on detecting bugs at execution time can be classified into two categories: *programming-rule-based* approach and *statistics-rule-based* approach [60]. While methods in both categories check certain rules during program execution, they focus on different types of rules. Programming-rule-based approaches detect violations of rules imposed by specific languages such as C/C++ or specific interfaces such as Message Passing Interface (MPI). For example, "bounds of the message buffer cannot exceed its allocated bounds" and "all members of one process group must execute collective operations over the same communicator in the same order" are two rules used by MPI-CHECK [38] and Umpire [55], respectively. Much research has been conducted in this category, such as Purify [25], Valgrind [42], Umpire [55], MARMOT [31], MPI-CHECK [38], etc. While

they are effective in detecting some types of software bugs at runtime, their rules are either too general to be able to detect semantics-related bugs [25, 42] or highly dependent on domain-specific expertise and human efforts [55, 31, 38].

Statistics-rule-based methods extract rules statistically at program runtime, and check the violations of the extracted rules. Statistical rules, or dynamic invariants [18, 24], are properties that *likely* hold at a certain point or points in a program. One can extract invariants at runtime over a single run or across multiple runs. Several recent works [18, 24, 60] have demonstrated that statistics-rule-based approaches are very promising due to their effectiveness in detecting bugs that do not violate any programming rules. Daikon [18] and DIDUCE [24] extract value-based invariants (i.e., the possible values of a given variable are always within a certain range) at runtime and can detect bugs that generate abnormal values. Similarly, AccMon [60] captures the runtime program-counter-based invariants (i.e. a given variable is typically accessed by only a few instructions), and use them to detect memory-related bugs. Many statistical rules extracted from program execution are related to program semantics, which are usually not accurately documented by programmers and difficult to infer from the code itself [53].

Parallel or distributed programs, the common type of most applications running on HPC systems, are especially suitable for applying statistics-rule-based methods. In addition to the *temporal* dimension explored by previous works [18, 24, 60] (i.e., invariants based on program behaviors in multiple runs or multiple phases of a single run), one can explore the *spatial* dimension (i.e., invariants based on the behaviors of multiple concurrently-running processes) in parallel systems. For example, in scientific parallel applications, usually similar or even identical tasks are performed in multiple iterations (the temporal dimension), as well as by multiple processes (the spatial dimension). Similarly, in some commercial HPC systems such as web server farms, a group of processes concurrently handle tasks in the same way for achieving high throughput.

Recently, Mirgorodskiy et al. have conducted an initial study of using statistics-rule-based approach to diagnose software problems in large-scale systems [41]. Their approach extracts the invariant of function time distribution in control flow, and then identify the abnormal process among a large number of parallel processes. They have shown that this invariant is effective for locating problematic processes and functions. However, their approach has two limitations in detecting software bugs. First, they cannot detect bugs that do not cause abnormal function time distribution across multiple processes. For example, data corruption may only cause wrong results without affecting the function time distribution. Similarly, some bugs manifest themselves in all the processes, resulting in the same distorted function time distribution for all the processes, and thereby their approach cannot detect these bugs. Second, the function time distribution invariant is easily interfered by system-level noises [8] (e.g., process scheduling, signal delivery, network congestion), and thus, it does not reflect the semantics of programs very accurately. For example, the processes performing identical tasks on different nodes can show very different function time distribution if the network traffic load is unbalanced across nodes.

Our Contributions. In this paper, we propose a novel statistics-rule-based technique, called *data movement (DM)-based invariants*, to find *hard-to-detect* software bugs that can cause *severe* problems such as data corruptions and deadlocks in large-scale parallel programs. Our idea is based on the observation that data movement in parallel programs typically follow certain patterns. If we can extract such DM-based invariants at runtime, it is possible to detect abnormal data movements that are caused by potential software bugs.

Our idea is inspired by the fact that data movements in parallel programs are pervasive and bug-inducing. Different from sequential programs, parallel programs require multiple processes to coordinate with each other to perform large tasks. Therefore, processes in parallel programs usually communicate with each other very frequently. Unfortunately, programmers can easily make mistakes in various situations when performing data movements. On the application level, many parallel algorithms require data to be exchanged in non-trivial ways due to subtle boundary condition handling. In addition, parallel programming models, such as MPI, provide a variety of communication interfaces with different semantics such as point to point send/receive, collective calls, etc. Often, it is difficult for application programmers to precisely understand the subtle semantic differences and choose the correct interface. Furthermore, to achieve better performance, the applications and libraries may introduce aggressive and error-prone optimizations, which may work correctly for most of the time but affecting correctness in some corner cases. Therefore, bugs can be easily introduced and manifested in pervasive data movements in parallel programs.

More specifically, we propose two types of DM-based invariants: *frequent chain (FC)*-invariants, i.e., frequently occurring data movement chains, and *chain distribution (CD)*-invariants, i.e., clusters of data movement chain distributions of multiple processes. FC-invariants and CD-invariants focus on temporal and spatial similarity of data movements in parallel programs, respectively. The violations of them, abnormal data movement chains for FC-invariants or outlier data movement chain distribution in one process for CD-invariants, may indicate the potential software bugs such as data corruptions, livelocks, deadlocks, etc. Note that these two types of invariants are based on data movement chains formed by linking individual data movements where the destination of one data movement is the source of a subsequent data movement. The reason for doing so is that data movement chains reflect better semantic information than individual data movements.

Based on these ideas, we have built a tool, called DMTracker, to extract FC-invariants and CD-invariants at runtime and check for violations of them. Our experiments with two real-world bug cases in MVAPICH/MVAPICH2 [43], a popular MPI library, have shown that DMTracker can effectively detect them and report abnormal data movements to help programmers quickly diagnose the root causes of bugs. Complementary to existing programming-rule-based and statistics-rule-based tools, DMTracker has the following unique advantages, some or all of which are unavailable in other tools.

- To the best of our knowledge, DMTracker is the first automatic tool that utilizes statistical rules based on data movements (i.e., DM-based invariants) for detecting *hard-to-detect* software bugs that can cause *severe* problems such as data corruptions and deadlocks in parallel programs. To find software bugs, it focuses on

a key aspect in parallel programs – the data movements. Based on DM-based invariants, DMTracker can detect various types of severe bugs such as data corruptions and deadlocks that manifest themselves in data movements and help programmers to diagnose the root causes.

- DMTracker can detect both deterministic and non-deterministic software bugs that manifest themselves in only a few processes or across all processes. This is because it extracts invariants by exploring both temporal and spatial similarities of data movements. Our experiments have shown that DMTracker is effective in detecting both deterministic bug and non-deterministic bugs. In contrast, previous work [41] cannot handle software bugs manifested across all the processes, which is demonstrated by our first bug case.

- DMTracker can detect software bugs that do not violate the function time distribution since DM-based invariants capture "how data move" instead of "when data move." Our experiments with the second bug case have shown that the problematic function time distribution can be easily overshadowed by other functions or system-level noises and thus it is hard to be identified by previous work [41].

- DMTracker incurs low overhead due to our system design and usage of a low-overhead dynamic instrumentation tool called Pin [39]. Our experimental results show that the runtime overhead of DMTracker is only 0.9-6.0%. Therefore, it is possible to directly apply DMTracker to production runs.

The rest of this paper is organized as follows: In Section 2, we discuss the background of statistics-rule-based bug detection. In Section 3, we introduce the DM-based invariants; In Section 4, we present our design of DMTracker and discuss several key design issues. We describe the evaluation results in Section 5 and discuss related work in Section 6; Finally, we conclude in Section 7.

2. BACKGROUND: STATISTICAL RULE BASED BUG DETECTION

Statistics-rule-based bug detection methods extract statistic rules, (i.e., dynamic invariants), at program runtime and check violations of the extracted invariants, which indicate potential software bugs. These approaches are motivated from an observation that the *hard-to-detect* bugs are usually those lurking in a corner case that rarely happens. Because the program behaves correctly for most of the cases, the bugs in corner cases are more difficult to be manifested by testing and thus more likely to cause problems in production runs. Due to their hidden nature, these bugs take much more time to detect and fix.

In recent years, research efforts have been made toward this direction [18, 24, 60, 37]. They have introduced several types of dynamic invariants to detect bugs that manifest themselves in different ways. Daikon [18] and DIDUCE [24] focus on the value ranges of variables and use them as invariants to detect the abnormal values of variables. AccMon [60] focuses on the program counter (PC)-based invariants to detect the abnormal instructions accessing to a certain memory location. AVIO [37] makes use of Access Interleaving (AI) invariants to detect the violation of atomicity execution in multi-thread programs. These works focus on sequential programs, and thus their proposed invariants are mainly to capture the temporal similarity of the programs behavior. Recently in [41], an anomaly-detection method was proposed to use function time based invariants for diagnosing problems in large distributed computing environments. This work focuses on identifying the abnormal process in a parallel application, by capturing the spatial similarity of parallel programs.

In our work, we focus on one of the key aspects of parallel programs, the data movements, and propose two data movement-based invariants to capture both temporal similarity and spatial similarity in parallel programs.

3. DATA MOVEMENT-BASED INVARIANTS

Data movement is the movement of a chunk of memory data from a source buffer to a destination buffer. For example, copying a chunk of memory data from buffer A to buffer B corresponds to one data movement $A \rightarrow B$. Typically, we can capture data movements in parallel programs on different levels: on application level by regarding each communication calls such as MPI library calls as one data movement, or on library level by analyzing each primitive operation such as memory copy, network send/receive, etc., as one data movement. For bug detection purposes, we choose the library level because it provides comprehensive information for finding bugs in applications as well as communication libraries and it is decoupled from particular programming models and communication interfaces.

Individual data movement reveals little information about program semantics. Thus it is difficult to extract invariants from them. To address this issue, we link a series of data movements to form a *data movement (DM)-chain* in a way that the destination of previous data movement is the source of the subsequent data movement. Figure 1 illustrates a simple DM-chain between two processes. The whole chain in the figure can be caused by a pair of communication calls such as MPI_Send and MPI_Recv on the application level. DM-chains are the basis of our proposed two types of invariants, which are described in the following two subsections.

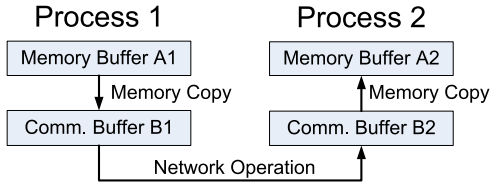


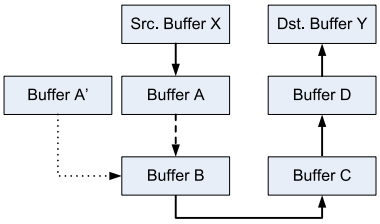
Figure 1: A simple DM-chain: A1→B1→B2→A2

3.1 Frequent Chain Invariants

FC-invariants are the frequently-occurring DM-chains. This is based on the observation that processes in parallel programs often exhibit temporal similarity, e.g., performing similar or identical tasks in multiple iterations. As a result, similar DM-chains occur many times during program execution. Based on this observation, we can group similar DM-chains together according to their type information, e.g., call sites of data movements and memory buffers. Then we can use large-sized groups, i.e., frequently-happening DM-chains, as FC-invariants.

```
1  int send (void* buf, int len, ...) {
2    //Send operation using bottom-fill
3    void *comm_buf_bottom = get_comm_buf_bottom(...);
4    void *data_buf = comm_buf_bottom - len;
5    memcpy (data_buf, buf, len);
6    if (need_piggyback) { //Rare case
7      Piggyback_t *pb = (Piggyback_t*) data_buf-sizeof(Piggyback_t);
                                  /*bug here: missing parentheses*/
8      ... /*Fill piggy back structure*/
9      return low_level_send((void*)pb, len+sizeof(Piggyback_t), ...);
10   } else { // Common case
11     return low_level_send(data_buf, len, ...);
12   }
13 }
```

(a)



(b)

Figure 2: An abstracted bug case reflected in abnormal data movement chain

Based on the FC-invariants, it is possible to detect abnormal data movements, i.e., similar to a FC-invariant but with slight difference. These abnormal data movements are potentially caused by software bugs and deserve programmers' attention. Typically, abnormal data movements are caused by buffer misuse and can lead to data corruptions as well as other errors such as crash or deadlock.

Figure 2(a) shows a simplified bug case extracted from a communication library, where data corruption is caused by pointer misuse. While the code path of common case goes through line 11, the bug is at line 7, in the code path to deal with a rare case, where some data need to be piggybacked to the packet to notify its peer about some event, e.g. out of resource. Since the programmer tends to think more of bytes rather than data objects when programming network protocols, $sizeof$ is used to calculate the offset. Unfortunately, without parentheses, the type cast happens in a higher precedence and the address is subtracted in unit of $sizeof(Piggyback\_t)$, so the actual address change is $sizeof(Piggyback\_t) \times sizeof(Piggyback\_t)$, which causes the pointer to point to another buffer. This bug can easily slip through normal software tests since it deals with uncommon cases only when some rare event occurs. Furthermore, it is difficult to detect this bug during production runs since it may manifest itself as silently sending incorrect data to the remote receiver.

Figure 2(b) shows data movements related to the $send()$ routine, including both common cases and uncommon cases. It clearly tells that the bug manifests itself in the abnormal data movement. During normal program execution, DM-chains related to the $send()$ routine are $X \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow Y$. When the bug occurs, the DM-chain changes to $A' \rightarrow B \rightarrow C \rightarrow D \rightarrow Y$, similar to the FC-invariant for the common case but with the link from $A \rightarrow B$ broken. Obviously, it violates the FC-invariant.

FC-invariants can be extracted from one run or based on multiple runs. DMTracker does not require reference data to distinguish abnormal chains from normal. But in some cases, the incorrect DM-chains can be the "common cases" in one specific run when the bug happens. To be more effective in detecting bugs in these cases, DMTracker also allows users to provide a "training set", a number of traces known to reflect correct behaviors, so that it can extract patterns only from those traces. Then DMTracker can be more effective in locating incorrect behaviors in traces from problematic runs.

3.2 Chain Distribution Invariants

CD-invariants are the clusters of chain distributions that the chain distribution of each process should fit in. It is based on the observation that processes in parallel programs often exhibit spatial similarity, e.g., performing similar or identical tasks and following the same symmetric communication patterns in multiple processes. As a result, the distribution of various groups of DM-chains (grouping DM-chains using the same method as we did for FC-invariants) are similar across multiple processes. Therefore, we can use chain distributions clusters as CD-invariants.

Figure 3 demonstrates the chain distributions in all processes for High Performance Linpack (HPL) benchmark [2] and SP, MG in NAS Parallel Benchmarks (NPB) [56]. (Most benchmarks in NPB show similar trends, we omit them due to lack of space). The x-axis indicates the process ID of each process in the parallel programs and each column of a graph shows the chain distribution in that process. It is clear that during normal execution, all 64 processes for the tested benchmarks share very similar chain distributions.

Based on CD-invariants, it is possible to capture bugs that happen in a small number of processes. DMTracker compares chain distributions across all processes and automatically locates the manifesting process in a large number of peers, so that the search space for the bug can be greatly narrowed down. Typically, an abnormal chain distribution is caused by some algorithm or protocol error in parallel programs, which can manifest itself as infinite loop (i.e., livelock), deadlock, etc.

In addition to the chain distributions of the whole run, comparing the chain distributions within a certain time period can be more effective for detecting bugs in some cases. For example, to diagnose deadlock bug, chain distributions of all processes in the *last phase* of execution are especially useful. Since in a deadlock situation, processes usually stop to proceed their executions and also stop communicating, the distorted chain distribution only shows in the last phase and can potentially be overshadowed by the chain distribution in earlier phases.

In cases where processes in a parallel application are designed to perform different tasks (e.g. in master-worker model), the chain distributions for some processes (e.g. rank 0) will be very different from others even in correct runs. In these cases, CD-invariants are only valid for groups of processes which perform similar or identical tasks. DMTracker can be configured to only analyze chain distributions of a specific subset of processes rather than all processes in the whole parallel program.

4. DESIGN OF DMTracker

DMTracker consists of two major components, the online tracking component and the offline analysis component, as shown in Figure 4. The online tracking component (Section 4.1) collects *data movement (DM)-traces* at runtime by

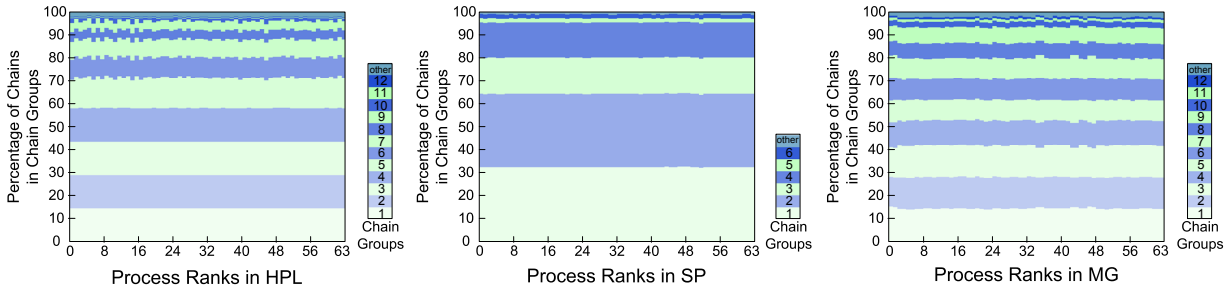


Figure 3: Distribution of Chains for 64 Processes

leveraging lightweight binary instrumentation. Based on the collected DM-traces, the offline analysis component forms DM-chains (Section 4.2), extracts FC-invariants and CD-invariants respectively (Section 4.3) and detects anomalies that violate the extracted invariants (Sections 4.4 and 4.5).

4.1 Lightweight Data Movement Tracking

The online data movement tracking component records data movement related information into traces, called DM-trace, by instrumenting binary code of the parallel programs. A dynamic instrumentation tool called Pin [39] is used in our current implementation. Unlike traditional bug detection tools such as Purify [25] that track every memory accesses, DMTracker only instruments function calls related to memory management (e.g. allocation/deallocation) and data movements (e.g., memory copy and network operations). Therefore, it incurs very low overhead. Furthermore, by directly instrumenting binary code, DMTracker requires no recompilation of the source code, which can avoid some inconvenience for usage.

To capture data movement semantics, DMTracker records the following information of instrumented function calls: (1) key arguments and return values, (2) call sites that contain stack context when the call is made, (3) thread IDs for multi-threaded processes, and (4) local timestamps when the call is made. More specifically, for memory allocation calls such as malloc, calloc, etc., DMTracker records both request size and the memory object address; for memory copy calls, it records source and destination addresses and the copy length; for network operations, it records the buffer, length, and the information to identify the network endpoint. The call sites are useful in analyzing traces and providing more diagnosis information to programmers. The timestamp is used to order local operations. Note that for the programs using customized memory management module, additional instrumentation needs to be done regarding the customized interfaces.

DM-trace grows moderately since DMTracker only records data movement related functions. Its growth rate largely depends on the communication patterns of parallel applications. In our experiments, typically 20MB of disk space can store traces for several minutes for parallel benchmarks such as HPL and NPB, and thus it is usually not a big problem for storing the whole traces into local disks. In addition, we can leverage existing techniques such as trace compression [44] and streaming processing [5] to reduce storage overhead if it becomes a big concern.

4.2 Preprocessing: DM-Chain Formation

Based on the collected DM-traces, DMTracker forms DM-chains by parsing data movement operations, DM-operations, and linking related data movements. For scalability, DMTracker processes individual traces and forms DM-chains in parallel instead of processing all the traces in a central node.

Parsing DM-Operations. DMTracker parses information of each function call recorded in the DM-traces and correlates each DM-operation to its source and destination buffers' allocation information. For example, a memory copy $A \rightarrow B$ correlates to the allocation information of A (e.g. malloc at PC_a) and B (e.g. memalign at PC_b). This correlation information provides contexts for linking data movements and grouping chains, which are discussed in detail later.

In parallel programs, some data movements such as data pack/unpack may contain multiple memory copies. In many cases, data in multiple small non-contiguous buffers are packed to a larger contiguous buffer, or data in a large buffer are unpacked to multiple smaller non-contiguous buffers. Since a memory copy usually requires source and destination to be contiguous, these operations require multiple memory copies. DMTracker aggregates multiple memory copies for a pack or unpack operation and parses them as a single DM-operation to better reflect the semantics of data movements.

For network operations, in this step, DMTracker can only correlate partial allocation information, either the source buffer (for send) or the destination buffer (for receive), to the data movement within each process. In order to link data movements across the network, DMTracker keeps the connection end-point information temporarily.

Movement Concatenation. Based on the parsed DM-operations, DMTracker forms chains by linking related data movements where the destination buffer of a previous DM-operation is the source buffer of a subsequent DM-operation. To form a complete chain, DMTracker performs two steps, intra-process linking and inter-process linking. The intra-process linking is to link DM-operations within one process. For example, it links memory copies with related memory copies or network operations. To efficiently match the source buffer of a DM-operation with the destination buffer of some previous DM-operation, DMTracker maintains an *active chain table* for existing DM-chains that can potentially have successive DM-operations. If matching happens, DMTracker extends the matched DM-chain and updates the table correspondingly. Otherwise, it inserts the DM-operation as a new DM-chain into the table.

The inter-process linking is to link the chains across different processes together by matching the send operations and receive operations. To achieve this, DMTracker maintains the information of network connections (e.g., file descriptors for sockets, queue pairs (QPs) for InfiniBand [29]). For the FIFO communication channel (e.g. TCP and InfiniBand RC), send and receive can be matched simply by the order. For the channels which do not guarantee FIFO, further in-

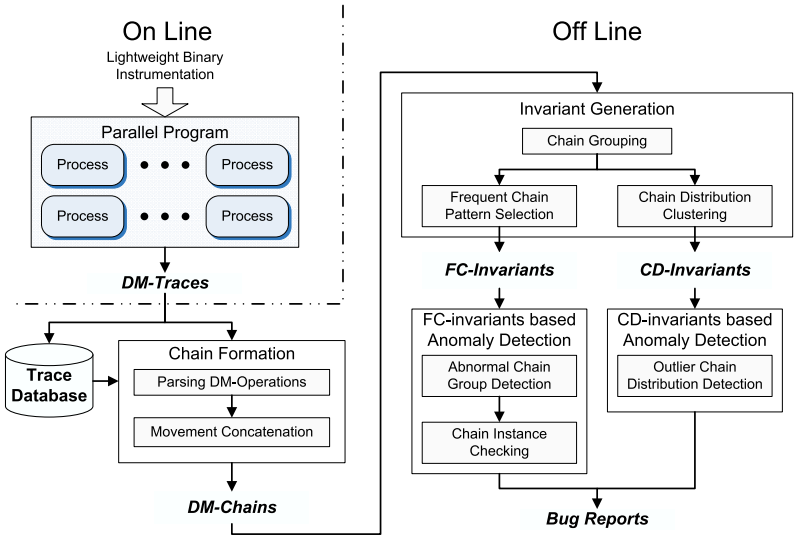


Figure 4: DMTracker Design Overview

strumentation needs to be done to track sequence numbers with the packets. When a send matches with a receive, the chain starting with the receive is linked to the end of the chain ending with the send. This way the DM-chain that reflects the whole process of data movement across multiple processes can be formed.

4.3 Invariants Generation

To generate FC-invariants and CD-invariants, DMTracker first groups the same type of chains together based on the information of data movements in each chain since chain groups provide better data movement semantics. More precisely, DMTracker regards two chains as *the same type* and puts them into *the same group* if the corresponding individual data movements within the two chains have the same call sites for the DM-operations and the same allocation call sites for their source and destination buffers. For example, the chain $A_1 \rightarrow A_2 \rightarrow A_3$ and the chain $B_1 \rightarrow B_2 \rightarrow B_3$ are of the same type and belong to the same chain group if the data movement $A_i \rightarrow A_{i+1}$ has the same call site as $B_i \rightarrow B_{i+1}$ and A_j has the same allocation call site as B_j, where $i = 1, 2$ and $j = 1, 2, 3$. We use call site information for grouping chains due to two reasons: (1) data movements at different program locations usually handle different cases, capturing program semantics in communication, and (2) the same memory allocation site usually allocates the same type of data, capturing characteristics of memory buffers.

4.3.1 Generating FC-invariants: Frequent Chain Pattern Selection

DMTracker extracts FC-invariants from the above-formed chain groups based on two criteria. Obviously, the chains of that group must happen frequently, i.e., the number of chains in the chain group should be relatively large. One naive way is to set an absolute number as the threshold and require all the FC-invariants with chain number larger than that threshold. However, it is difficult to do so in practice since the threshold is highly dependent on the number of chains in the trace. Therefore, DMTracker uses a relative number, the percentage of total number of chains, as the threshold.

Another criteria is that, the chain type for each chain group must preserve unique characteristics so that we can easily determine whether another chain matches the FC-invariant or not, and how much it matches. If a chain type contains only one data movement, there is no point to select it as invariant and compare other chains with it because other chains are either 0% match or 100% match, therefore they can not be the violations we are interested in, which are mostly similar to but have some difference from the invariant chain type. The uniqueness of a chain type is determined by the length of the chain as well as the uniqueness of each individual data movement that forms the chain. Therefore, we define the uniqueness value of a chain type C, $Uniqueness(C)$, as the sum of the uniqueness values of its DM-operations $(O_1, O_2, ...)$:

$$Uniqueness(C) = \sum Uniqueness(O_i) \quad (1)$$

Uniqueness values of different types of data movements should be assigned in different ways since they reflect different semantics. We normalize the single memory copy to 1, and define the uniqueness of a DM-operation O_i, $Uniqueness(O_i)$, as follows

$$Uniqueness(O_i) = \begin{cases} 1, & \text{if } O_i \text{ is a single memory copy;} \\ M, & \text{if } O_i \text{ is a network operation;} \\ \min(N, n), & \text{if } O_i \text{ is a data pack/unpack.} \end{cases} \quad (2)$$

In Equation 2, n represents the number of segments of data to pack/unpack, and M and N are tunable parameters. Since network operation involves two processes, we use $M = 2$ in our experiments. Uniqueness of data pack/unpack is designed that way because: a) it should reflect the number of segments it has, and b) it should not overwhelm other movements. We use $N = 10$ in our experiments.

The thresholds to select FC-invariants should partially depend on the target program. In our experiments, we set the uniqueness threshold to 5, and require the number of chains in a chain group for FC-invariants to account for more than 10% of the total number of chains in the chain groups whose uniqueness value is above the threshold.

4.3.2 Generating CD-invariants: Chain Distribution Clustering

DMTracker extracts CD-invariants, i.e., clusters of chain distributions that the chain distribution of individual processes should fit in, based on the chain distributions. We define the chain distribution in a trace T for a process as a vector, $CD(T)$. Each element in the vector represents the percentage of all the chains one chain group accounts for.

$$CD(T) = \left\langle \frac{Count(C_1,T)}{Count(T)}, \frac{Count(C_2,T)}{Count(T)}, ..., \frac{Count(C_m,T)}{Count(T)} \right\rangle \tag{3}$$

where m is the total number of distinct chain groups for all the processes, $Count(C_i,T)$ represents the number of chains in the i^{th} chain group C_i in the trace T, and $Count(T)$ represents total number of chains occurred in T. Even though in many cases a chain goes across multiple processes, to analyze the distribution, we consider a chain to belong to a certain process when the chain starts from that process. In this way, we do not calculate a chain multiple times and still have a symmetric measurement.

To cluster chain distributions, DMTracker uses Manhattan distance [32] to the k^{th} nearest neighbor [48] as our metric, which is also used in previous work [41]. The Manhattan distance between two traces T_i and T_j, $Distance(T_i,T_j)$, is the sum of the absolute differences of each element in the chain distribution vector as defined in Equation 4. $Distance^k$ is the distance to k^{th} nearest neighbor (Equation 5), which reflects how well a chain distribution fits into a cluster with multiple peers. A lower value of $Distance^k$ means the chain distribution fits better into a cluster. Thus, DMTracker uses it to measure how similar a chain distribution in one process is compared to its peers'.

DMTracker uses a certain range of $Distance^k(T_i)$ as CD-invariants. Similar to [41], in our experiments, the values of parameters k is set to $n/4$, where n is total number of processes.

$$\begin{aligned} Distance(T_i,T_j) &= \left| \frac{Count(C_1,T_i)}{Count(T_i)} - \frac{Count(C_1,T_j)}{Count(T_j)} \right| \\ &+ \ ... \\ &+ \left| \frac{Count(C_m,T_i)}{Count(T_i)} - \frac{Count(C_m,T_j)}{Count(T_j)} \right| \end{aligned} \tag{4}$$

$$\begin{aligned} Distance^k(T_i) &= Distance(T_i,T_{j_k}), \\ \text{where} \quad & Distance(T_i,T_{j_l}) < Distance(T_i,T_{j_{l+1}}), \\ & i \neq j_l, 1 \leq l \leq n-1 \end{aligned} \tag{5}$$

4.4 FC-invariants Based Anomaly Detection

Based on the extracted FC-invariants, DMTracker can detect abnormal chain groups potentially caused by software bugs, and validate the reported bug by checking each chain instance in the abnormal groups with its context.

4.4.1 Abnormal Chain Group Detection

DMTracker detects abnormal chain groups by comparing each chain group with the extracted FC-invariants. It considers a chain group C being an abnormal case of a FC-invariant P if they are similar enough and C is relatively rare compared to P. On one hand, C needs to be similar to P so that we can determine it as an abnormal case of P. The more similar they are, the more likely that C is an abnormal case of P, except C is identical to P. For example, if C matches 90% with P, then we can consider the rest 10% unmatched part to be abnormal. But if C only matches 10% with P, it is likely to be just an infrequent type of chains instead of being an abnormal case of P.

To measure the similarity between a chain group C and a FC-invariant P, DMTracker uses a metrics derived from *Jaccard Coefficient* [54], a widely used metric to measure similarity, to capture the matched part and unmatched part between C and P. Equation 6 shows the formal definition of similarity. To find the largest matching part ($C \cap P$), we symbolize the DM operations in the chain and convert the problem to a *longest common substring* problem. The longest common substring problem can be solved in $O(m+n)$ time with the help of a *generalized suffix tree* [23], where m and n are the lengths of the two strings.

$$\begin{aligned} Similarity(C,P) &= \frac{Uniqueness(C \cap P)}{Uniqueness(C \cup P)} \\ &= \frac{Uniqueness(C \cap P)}{Uniqueness(C) + Uniqueness(P) - Uniqueness(C \cap P)} \end{aligned} \tag{6}$$

On the other hand, C needs to be relatively rare compared to P. Otherwise, if C is almost as frequent as, or even more frequent than P, it is very likely to be another pattern instead of an abnormal case of P. To measure the rareness, DMTracker compares the frequency of C, $Frequency(C)$, with the frequency of P, $Frequency(P)$. There is no point to compare the C with P if $Frequency(C)$ is larger than $Frequency(P)$. We define the rareness, $Rareness(C,P)$, as below:

$$\begin{aligned} Rareness(C,P) &= \frac{Frequency(P) - Frequency(C)}{Frequency(P)}, \\ \text{where} \quad & Frequency(P) > Frequency(C) \end{aligned} \tag{7}$$

We are looking for abnormal chain groups with high *similarity*, meaning the chain group is very similar to the FC-invariants, and with high *rareness*, meaning the chain group is relatively rare compared to the FC-invariants. We choose to use harmonic mean to combine similarity and rareness because it is a commonly used way which prefers high scores in both dimensions. Thus the overall metric of *abnormality* can be defined as follows:

$$Abnormality(C,P) = \frac{2}{\frac{1}{Similarity(C,P)} + \frac{1}{Rareness(C,P)}} \tag{8}$$

For each pattern P_i, DMTracker detects a list of chain groups, noted as $C_{i_1}, C_{i_2}, ...$, with $Abnormality(C_{i_k},P_i) > Threshold_{Abnormality}$, $C_{i_k} \neq P_i$, and $Frequency(C_{i_k}) < Frequency(P_i)$. We call each pair, (C_{i_k},P_i), a *violation*. DMTracker then combines lists of violations for all patterns, and ranks them according to the abnormality score. In our experiments, DMTracker reports all violations with an abnormality score more than 0.7

4.4.2 Chain Instance Checking

Since not all DM-chains in the abnormal chain groups are necessarily caused by bugs, DMTracker needs to check each chain instance in the context of chain trace for validation and providing more detailed diagnosis information to the programmer.

In this step, DMTracker goes through each chain instance in the abnormal chain groups, and examines the DM-opera-

tions that happened immediately before/after the chain instance to see whether they match the previously unmatched part between the abnormal chain and its matching FC-invariant. For instance, assume that a chain group C with three DM-operations (X, Y, Z) matches third to fifth DM-operations in a FC-invariant P (U, V, X, Y, Z) and c is a chain instance of C. If some DM-operations closely before c in the chain trace match the DM-operations U and/or V in P, it strongly suggests a broken chain. DMTracker will highlight it by marking as "context match."

Furthermore, DMTracker can provide the detailed information with each abnormal instance into report, and rank the anomalies with "context match" and high abnormality score on the top. The reported information includes buffer address, buffer allocation call sites, data movement call sites, etc., of both the abnormal chains and their contexts (i.e. data movements chains happened before/after it). It is likely to be helpful for diagnosing the problem.

4.5 CD-invariants Based Anomaly Detection

Based on the CD-invariants, reflected as a certain range of $Distance^k$ scores, it is relatively straightforward to detect the outliers. Different from [48], where the number of outliers is given by users, here DMTracker uses another criterion to find all traces that are not similar enough to their peers. If a trace has a significantly larger value of $Distance^k$ than the average, e.g. K times larger, DMTracker will report it as an abnormal trace. In our experiments, we use 5 as the value of parameter K for fewer false positives. The chain group that contributes the most to the $Distance(T_i, T_{j_k})$ in Equation 4 is also included as diagnosis information in bug reports.

In some cases, the most interesting part of the chain distribution is about the *last phase* of execution. For instance, if a parallel program deadlocks and we want to find out which process causes this problem, the chains that happen much earlier in the run may dilute the difference. To deal with this case, we can use a binary search method. When no outliers have been found but users want to do in-depth diagnosis, DMTracker can perform multiple rounds of CD-invariant based detection with the chains in the halved time range.

4.6 Issues and Discussions

Additional Inter-process Communication Channels. In communication libraries, in addition to common network send and receive, there are other approaches to perform inter-process communication, such as through a shared memory region on the same host, or using Remote Direct Memory Access (RDMA) to directly access memory on a remote host. For data movements through shared memory, DMTracker tracks them as data movements through non-shared memory. However, our current prototype has not yet supported the shared memory region construction such as memory mapping to a common file. Therefore, DMTracker does not link the chains crossing processes if they are communicated through shared memory regions. This may affect the accuracy in some cases, but is not a major problem in our experiments because the separate chains are still analyzed in each process. We plan to track the construction of shared memory regions in our future work.

RDMA, an advanced feature provided by modern networks such as InfiniBand [29] and Quadrics [47], etc., allows one-sided communication. Our current prototype does not link chains through RDMA channels because only the process on the active side has a DM-operation. To address this issue, we need to modify device drivers or firmware of network interface cards to expose RDMA operation to the user-level process on the passive side.

Online Analysis and On-the-fly Detection. With more computing power provided by multi-core systems, DMTracker could process and analyze traces using dedicated cores in each node. Since the processing cores can directly access the traces in memory instead of via expensive file I/Os, it can achieve high performance. In addition, the storage overhead can be alleviated because much smaller intermediate results are needed for further analysis after preliminary local processing. These will enable us to extend DMTracker for performing on-the-fly detection in future work.

5. EVALUATION AND CASE STUDIES

The experiments described in this section were conducted on a 64-processor cluster with 32 nodes. On each node, there are two 3.6GHz, 2MB L2Cache CPUs and 2GB memory. These nodes are connected using InfiniBand PCI-Ex DDR adapters with 10Gbps peak unidirectional bandwidth. The Operating System is Linux with kernel version 2.6.17.7.

To evaluate DMTracker's functionality, we use MVAPICH/MVAPICH2 [43], a popular high performance open-source MPI library over InfiniBand, with two real-world bug cases: one data corruption bug causing incorrect results and one deadlock bug causing program to hang. Both MVAPICH and MVAPICH2 packages are large: each has more than 350,000 lines of C code in more than 1,500 source files.

To evaluate runtime overhead incurred by the online tracking component of DMTracker, we compare the performance difference of High Performance Linpack (HPL) benchmark and NAS Parallel Benchmarks (NPB) with and without DMTracker.

5.1 Case 1: Data Corruption in Communication

The data corruption bug in MVAPICH2 version 0.9.8 was triggered deterministically by executing a communication library for linear algebra, called BLACS (Basic Linear Algebra Communication Subprograms) [1]. The test program for the BLACS package, called *xCbtest*, reports "Invalid element" error in all the 64 processes after executing BSBR (broadcast/send and broadcast/recv) test cases. The data corruption bug happens silently (i.e., no hang or system failures) and is shown at the last stage of result verification.

After being applied for this scenario, DMTracker reports six abnormal chain groups that violate two of five extracted FC-invariants. Out of the reported six abnormal cases, the top two ranked anomalies indicate the real bug. In both cases, a FC-invariant (15075 times) is violated by rarely-occurred similar chains (154 times). Figure 5 shows the frequently-happening chain (FC-invariant) on the left side and the rare cases (a broken chain caused by the data corruption bug) on the right side. The instance context checking confirms that all the 154 chain instances are caused by the data corruption bug.

DMTracker not only detects this bug, but also provides useful diagnostic information to programmers for quickly locating this bug. With detailed information about the ab-

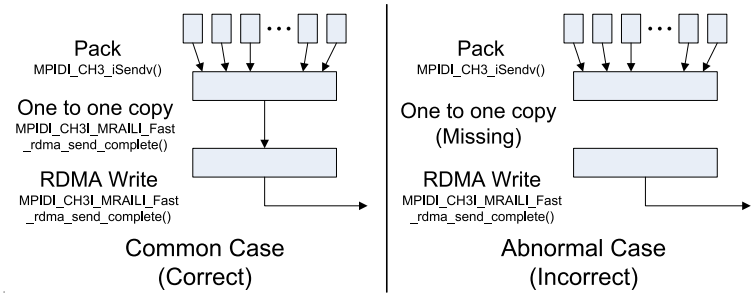


Figure 5: Case 1: Broken Chain

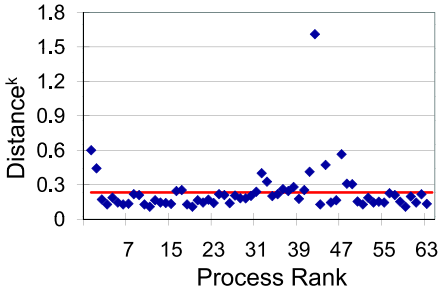


Figure 6: Case 2: $Distance^k$ of Processes

normal chain groups, the root cause of the problem, an optimization called *header caching* for RDMA operations fails to handle a corner case in communication protocols, can be easily identified. This demonstrates that statistical-rule-based tools like DMTracker can be helpful in detecting rarely-happening bugs and locating the root causes, which otherwise would require much more human effort.

Furthermore, this case study shows that DMTracker can also detect software bugs that manifest themselves across all the processes in a similar way. This is because DMTracker exploits temporal similarity within each process as well as spatial similarity across different processes. This data corruption bug has been triggered in all the 64 processes in a similar way, indicating that it is extremely hard, if not impossible, for previous work [41] to detect and diagnose it by only exploring spatial similarity among different processes.

5.2 Case 2: Deadlock in Connection Setup

The deadlock bug was triggered by running FT benchmark when testing an internal version of MVAPICH on 64 processes. The program hang non-deterministically with a very small chance during execution.

After being applied, DMTracker detects an anomaly that violates the extracted CD-based invariant. As shown in Figure 6, process 43 has a very high $Distance^k$ score, 1.61, which is significantly higher than the average score 0.23. It strongly indicates that the chain distribution in process 43 is an outlier of any cluster of chain distributions. DMTracker detects this non-deterministic bug via exploiting the spatial similarity across different processes.

DMTracker not only detects this bug, but also reports useful diagnostic information to programmers for identifying the root cause. It reports that the chain group with a network send operation in function *cm_post_ud_packet*() of the file *cm.c* is the major contributor to the high $Distance^k$ score. This information quickly narrows down the root cause to a specific function. In that function, accessing of a variable, which is forgotten to be defined as *volatile*, causes an intended benign data race to be harmful by a small chance.

Although the time spent on the function *cm_post_ud_packet*() in bug cases is much longer than that in normal cases, it can not be easily detected by the function time distribution method [41] due to noises caused by other functions. To achieve better performance, most communication libraries built on current main-stream high performance networks including MVAPICH/MVAPICH2 use a polling-based progressing mechanism. In this mechanism, when waiting for an event, the process will be busy waiting in multiple polling functions. The difference of time spent in abnormal functions is easily overshadowed by the time spent in these polling functions. In our experiments, we measured that the total time spent in the abnormal function in the outlier process is less than 0.05% of the total time in progressing functions, which is not reflected in visible difference of function time distribution. To filter the effect of polling functions, non-trivial work is required with respect to each library to determine which functions should count and which should not.

5.3 Runtime Overhead

In this set of experiments, we evaluated the performance impact of DMTracker's online tracking component with HPL benchmark with different problem sizes and NPB benchmark with class C. We ran these benchmarks based on MVAPICH2 version 0.9.8 natively and with tracking using 32 nodes with 2 processes per node (32x2).

Table 5.3 lists the relative performance degradation for HPL when applying DMTracker as compared to native. In general, the runtime overhead is very low, from 2.3% to 3.1%. The low overhead is because DMTracker only tracks a small set of functions related to data movements. In addition, we observe that the overhead decreases as the problem size of HPL benchmark increases. That is because when the problem size becomes larger, the data movements among processes tends to be in larger chunks, resulting in less frequent data movements and thus lower overhead. Therefore, for long-running applications operating on large data sets, the overhead incurred by DMTracker is expected to be low.

Table 1: Runtime Overhead for HPL

Problem Size	40000	50000	60000
Relative Degradation	3.1%	2.7%	2.3%

We also observe that applications with different communication characteristics show different runtime overheads when being tracked by DMTracker. As shown in Table 2, the overhead varies largely for different NPB benchmarks. The ones with very frequent communications, like CG, IS, and LU, show slightly higher overhead (3.9%-6.0%); while others, BT, FT, MG, and SP, only show almost negligible overhead (0.9%-1.6%). These results are expected since the overhead is caused by tracking data communications in applications. As demonstrated by both HPL and NPB, DMTracker incurs very low runtime overhead, which indicates that DMTracker can be deployed in production runs.

Table 2: Runtime Overhead for NPB

Benchmarks	BT	CG	FT	IS
Relative Degradation	1.0%	3.9%	0.9%	4.2%
Benchmarks	LU	MG	SP	
Relative Degradation	6.0%	1.4%	1.6%	

Currently, the trace processing and analysis components of DMTracker are still performed offline. Therefore, this section only discusses the runtime overhead incurred by the online tracking component. In the future, we plan to extend DMTracker to detect software bugs on-the-fly by leveraging stream processing algorithms and multi-core architectures.

5.4 Sensitivity Study and False Positives

We have also conducted a set of experiments to study the parameter sensitivity of DMTracker. As in most static-rule-based tools, parameters and threshold values can affect the balance between effectiveness and false positives of DMTracker. With lower thresholds, DMTracker can report more potential bugs (also more false positives), while using higher thresholds can reduce the number of false positives with the risk of missing the real bug (false negative).

For FC-invariants, we tried several options to lower the thresholds of parameters in case 1: a) uniqueness requirements for a chain group to be FC-invariant, from 5 to 3 (the minimum meaningful value); b) frequency thresholds for a chain group to be FC-invariant, from 10% to 5% (a very low value); and c) abnormality threshold for violation report, from 0.7 to 0.5. Experimental results were sensitive to these parameters to some extent: up to 3 more statistical invariants and up to 15 more violations were reported, but the real bug cases still ranked at the top and highlighted by the context match. Most false positives for FC-invariants we have encountered during the experiments with case 1, NPB, and HPL, can be summarized as follows.

Infrequently used communication protocol. Since the flow control algorithm in the MPI library may choose different protocols according to network resource usage, in some cases a fall-back protocol may be used. Then the data movements in the infrequently used protocol will cause a very small group of chains to be very similar but different from FC-invariants, which will be reported as violation by DMTracker. This type contributes more than half of the false positives. To prune them, we can use more sophisticated checking steps by incorporating semantic information.

Buffer reuse for control message. In the MPI library, control messages are passed using the same protocol as small data messages, but the communication buffers for control messages are immediately reused instead of being copied to application buffers. Thus the data movements for control messages may result in a slightly different (usually longer) chain than FC-invariant, which will be considered as a violation.

For CD-invariants, we tried different values of parameter k, such as $n/2$, $n/4$, $n/8$, where n is the number of processes. The result of case 2 is not affected by different values. When applying to HPL and NPB, a few false alarms were given for LU and CG indicating that they have abnormal processes with different data movement statistics. These false alarms can be pruned by checking the data movement statistics about previous known good runs.

Note that to correct suboptimal parameter settings, users only need to rerun the analysis component offline, which is generally much cheaper than rerunning the large-scale parallel program again.

6. RELATED WORK

Our work builds upon many previous studies on software reliability. Due to lack of space, here we only provide a brief survey on related work in three categories: a) bug detection for parallel programs, b) problem diagnosis in large scale systems, and c) general software bug detection technologies.

Many research efforts have been made to help detect bugs in parallel programs, including parallel debuggers [19, 40, 26, 6], technologies to support interactive parallel debugging [10, 28, 49, 4], and automatic bug detection tools [12, 55, 31, 38, 27, 21, 41]. Most of these focus on helping interactive debugging by using automated information collection/aggregation technologies and visualization technologies. While some follow programming-rule-based approaches, the study in [41] uses a statistical-rule-based method by detecting anomaly in function time distribution. Our work is complementary to these works by using a statistical-rule-based method to detect anomaly in data movements.

Problem diagnosis in large scale systems has been studied for many years [13, 59, 9, 7, 50, 3, 41, 30]. These works mainly study how to analyze and locate the root causes of system failures or performance problems. The root causes can be hardware failures, configuration problems, software bugs, operator mistakes, etc. Different from these works, we focus more on capturing the problems in programs' semantics rather than monitoring environments or configurations, since the purpose of our study is on detecting and analyzing software bugs in parallel programs.

There have been many studies on technologies to detect software bugs in general systems. Static bug detection technologies include implementation-level model checking [58, 17], compiler based technologies [15, 16, 11, 14, 20], and data mining technologies [33, 34]. Dynamic bug detection technologies include pure software-based approaches [25, 42, 51, 18, 24, 35, 36, 22] and hardware-supported approaches [61, 60, 37, 45, 57, 46]. Unlike these works, we solely focus on the data movements in parallel programs and propose DM-based invariants to capture the special semantics.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed an innovative statistical-rule-based approach to automatically find the *hard-to-detect* bugs that can cause *severe* problems such as data corruptions and deadlocks in large-scale parallel programs. Our approach extracts the data movement based program invariants at runtime, and detects anomaly based on the extracted invariants. Based on this idea, we have built DMTracker to help programmers locate root causes of software bugs. Our evaluation with two real-world bug cases in MVAPICH/MVAPICH2 shows that DMTracker is effective in detecting them and providing useful diagnosis information. In addition, DMTracker only incurs a very low runtime overhead (0.9%-6.0%) measured by HPL benchmark and NAS Parallel Benchmarks, so that it is possible to be deployed in production runs.

Current DMTracker is only an initial step toward this direction. For future work, we plan to conduct more case studies for more types of programs. We also plan to study more intelligent algorithms in data movement analysis and more models besides the chain to model data movements for detecting bugs more effectively. In addition, we would like to extend DMTracker to detect abnormal behaviors in parallel programs on-the-fly.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for useful feedback. We also thank Dr. Zhenmin Li for discussions on analysis algorithms and Wei Huang for discussions on the bug cases.

This research is supported in part by DOE grants #DE-FC02- 06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342 and #CCF-0702675; grants from Intel, Mellanox, Cisco systems, Linux Networx and Sun Microsystems; and equipment donations from Intel, Mellanox, AMD, Apple, Appro, Dell, Microway, PathScale, IBM, SilverStorm and Sun Microsystems.

9. REFERENCES

[1] Basic linear algebra communication subprograms (BLACS). http://www.netlib.org/blacs/.

[2] A. Petitet and R. C. Whaley and J. Dongarra and A. Cleary. High performance linpack. http://www.netlib.org/benchmark/hpl/.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale applications. In *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 3 2007.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM Symposium on Principles of Database Systems (PODS)*, June 2002.

[6] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a traditional debugger to debug massively parallel applications. *J. Parallel Distrib. Comput.*, 64(5):617–628, 2004.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[8] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. Operating system issues for petascale systems. *ACM SIGOP Operating Systems Review*, 40(2):29–33, 2006.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, 2002.

[10] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. on Programming Languages and Systems*, 13(4):491 – 530, 1991.

[11] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2003.

[12] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of mpi programs with intel message checker. In *Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS)*, 2005.

[13] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Intl' Conf. on Software Engineering (ICSE)*, 2001.

[14] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2003.

[15] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[17] D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *Symposium on Network System Design and Implementation (NSDI)*, 2004.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering*, 27(2):99–123, 2001.

[19] Etnus, LLC. TotalView. http://www.etnus.com/TotalView.

[20] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1994.

[21] G. Florez, Z. Liu, S. Bridges, R. Vaughn, and A. Skjellum. Detecting anomalies in high-performance parallel programs. In *Information Technology: Coding and Computing (ITCC)*, 2004.

[22] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *IEEE/ACM Intl' Conference on Automated software engineering (ASE)*, 2005.

[23] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[24] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ACM Intl' Conf. on Software Engineering (ICSE)*, New York, NY, USA, 2002. ACM Press.

[25] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, 1992.

[26] R. Hood. The p2d2 project: building a portable distributed debugger. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, 1996.

[27] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *ACM/IEEE Conf. on Supercomputing (SC)*, 1990.

[28] R. Hood and G. Matthews. Efficient tracing for on-the-fly space-time displays in a debugger for message passing programs. In *Intl. Symposium on Cluster Computing and the Grid (CCGRID)*, 2001.

[29] InfiniBand Trade Association. http://www.infinibandta.org.

[30] K. L. Karavanic and B. P. Miller. Improving online performance diagnosis by the use of historical

performance data. In *ACM/IEEE Conf. on Supercomputing (SC)*, 1999.

[31] B. Krammer, K. Bidmon, M. S. Mller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing (PARCO)*, 2003.

[32] E. F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, 1987.

[33] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[34] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[35] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2003.

[36] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[37] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. In *ACM Intl' Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[38] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.

[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[40] S. S. Lumetta and D. E. Culler. The mantis parallel debugger. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, 1996.

[41] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *ACM/IEEE Conf. on Supercomputing (SC)*, 2006.

[42] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, 2007.

[43] Network-Based Computing Laboratory. MVAPICH: Mpi for infiniband. http://mvapich.cse.ohio-state.edu.

[44] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[45] S. N. G. Pokam and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *International Symposium on Computer Architecture (ISCA)*, 2005.

[46] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *IEEE Intl' Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[47] Quadrics. http://www.Quadrics.com.

[48] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *ACM SIGMOD Intl' Conf. on Management of Data (SIGMOD)*, 2000.

[49] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *ACM/IEEE Conf. on Supercomputing (SC)*, 2003.

[50] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processes. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[51] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[52] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, Washington, DC, 2006.

[53] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *Intl' Symposium on Fault-Tolerant Computing (FTCS)*, 1992.

[54] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

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

[56] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *ACM/IEEE Conf. on Supercomputing (SC)*, 1999.

[57] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[58] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[59] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. *SIGOPS Oper. Syst. Rev.*, 40(4):375–388, 2006.

[60] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *IEEE/ACM Intl' Symposium on Micro-architecture (Micro)*, 12 2004.

[61] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *International Symposium on Computer Architecture (ISCA)*, 2004.