Automatic Software Interference Detection in Parallel Applications

Abstract

We present an automated software interference detection methodology for Single Program, Multiple Data (SPMD) parallel applications. Interference comes from system and unexpected processes. If not detected and corrected such interference may result in performance degradation. Our goal is to provide a reliable metric for software interference that can be used in soft-failure protection and recovery systems. A unique feature of our algorithm is that we measure the relative timing of application events (i.e. time between MPI calls) rather than system level events such as CPU utilization. This approach lets our system automatically accommodate natural variations in application utilization of resources. We use performance irregularities and degradation as sign of software interference. However, instead of relying on temporal changes in performance, we design a system that detects spatial performance degradation across multiple processors. We also show examples on multiple parallel applications that demonstrate our technique's effectiveness, resilience and robustness.

I. Introduction

High-performance computing systems with thousands of processors have been introduced to execute large-scale scientific applications. Management and maintenance of such systems are daunting tasks, and it is crucial to have automated supporting mechanisms. In this paper, we present and automated mechanism for software interference and resource contention detection.

Performance of parallel applications depends on user-level execution of the application code and interfering system level operations that occur simultaneously. Hence, interfering OS activity can cause resource contention and performance degradation for the application [11], [15]. Our objective is to be able to detect and quantify performance irregularities and degradation on individual processors running a parallel application and use it as an indicator and metric for the intensity of system level interference. Such a metric has several applications in autonomous maintenance and management of high-performance computing systems to detect problems such as:

Inconsistent system and OS kernel setup may cause performance degradation in systems. In large systems with thousands of nodes, it is possible that system manager fails to update all nodes kernels properly. If kernel inconsistency results in performance degradation on some of the cluster nodes, our algorithm could detect it.

System software failure is preceded with performance degradation. Failure prediction systems generally rely on machine learning and statistical inference algorithms to predict failures before they occur. These systems require a local mechanism that can monitor degradation of a node attribute over time for early failure detection. Modern hardware devices are equipped with such monitoring mechanisms. For instance, most disk drives follow the SMART protocol, and provide indication of suspicious behaviors. Similarly, motherboards contain temperature sensors, which

can be accessed via interfaces. Network drivers, such as those for Myrinet interface cards, maintain statistics of packet loss and retransmission counts. However, there is no failure detection mechanism for system software. This is despite the fact that most failures are due to software errors. For instance, failure logs for an eight-month period from the Platinum and Titan clusters at NCSA shows that while 0.1 and 5 percent of failures were due to hardware errors, 84 and 60 percent of them were due to software errors respectively [7]. Our proposed mechanism can be used as a failure detection and monitoring mechanism for early prediction of software errors.

Software aging is a phenomenon, usually caused by resource contention that can cause computer systems to hang, panic, or suffer performance degradation. Software rejuvenation is a proactive fault management technique for cleaning up the system and to prevent severe failures or systems performance degradation. Proactive rejuvenation (sometimes called therapeutic reboots) can include re-initialization of components, purging shared-memory pool latches, or node fail-over. In a prediction-based rejuvenation system, our proposed mechanism can be used to activate the rejuvenation mechanism(s) to avert catastrophic failures.

Previously, the general approach for failure prediction and interference detection in the systems is to monitor and collect information on critical resource attributes such as CPU, disk, memory and network. Then, by using statistical, machine learning and curve fitting techniques try to predict and detect software interference or failure [1], [4], [5], [10]. This approach requires multiple sampling of the normal system behavior for every fixed configuration to tune its detection parameters.

In our approach, instead of system resource utilization, we first monitor short term performance of the software application. Then, we compare every processor's performance with a base line formed from average and standard deviation of all processors performance. Hence, instead of relying on absolute performance measurements of each processor separately, we consider relative performance of processors to detect performance degradation on individual processors. If the relative performance of the processors do not change dramatically for different configurations, there is no need for learning phase for every possible configuration.

Solution Overview: Our goal is to develop general interference monitoring algorithms for MPI applications. For each processor we form a 4-tuplet profile sequence. The first entry is the MPI function, the second one specifies the corresponding peer node(s) of the MPI call, the third entry is the time elapsed since the last MPI call (computation time) and the fourth entry is elapsed time in this MPI call (communication time). We abstract the sequence of the first two tuplets to a grammar that describes the behavior patterns. This abstraction enables us to extract *Typical Sequences* (TS) that are common patterns in processor profiles. Then, we monitor computation (communication) time of typical sequences to form comparable samples for all processors. Note that for each processor and typical sequence we have a time series. Further, for each typical sequence we form two additional time series, whose samples are the average and standard deviation of processor samples. Processor typical sequence samples are compared against *dynamic threshold* values computed from the average and standard deviation samples. We form an *Interference Level Signal* (ILS), which is the running average of the number of processor samples that are above their corresponding threshold value. Finally, for each processor, we use all typical sequences interference level signals to compute *interference metric*, a positive number whose value indicates interference level for the program

running on that processor.

Dynamic threshold values are computed *separately* for each processor and typical sequence, even though all of them are functions of the same average and standard deviation functions. Therefore, what we measure is relative deviation of performance with respect to the average and standard deviation rather than absolute performance values. Hence, we are resilient to natural temporal performance variations as long as the relative performance does not change dramatically.

Our main contributions in this paper are:

1) We use MPI[1] profiling to monitor and separate computation and communication components of the performance.
2) We use the Sequitur algorithm [14] to autonomously extract typical profile sequences that are repeated and common between all processor profiles.
3) Our approach takes advantage of spatial and relative performance correlation between processors that usually exists in SPMD parallel applications.

The rest of the paper is structured as follows: In section II, we briefly review MPI profiling and the Sequitur algorithm. In section III, we go over the main challenges in the design of our algorithms to provide the rational for our proposed solutions. In section IV, we introduce the main components of our proposed algorithm. In section V, we study performance and characteristics of the algorithm on NAS [2] benchmarks and a production level scientific application called Parallel Ocean Program (POP) [17]. In section VI, we review and discuss previous related works.

II. Preliminaries and Tools

A. *MPI Profiling*

We use the MPI profiling interface to perform our data gathering. The idea behind the MPI profiling interface is to use the linker to substitute a user-written *wrapper* function for the MPI function called by the application. Every MPI function with the name MPI_Xxx is also callable by the name PMPI_Xxx name. Hence, we write a wrapper function with the same name MPI_Xxx, where we perform our desired profiling and also call the real MPI function by the name PMPI_Xxx.

In the start and end point of wrapper functions, we save the wall time value. The computation time for an MPI call is the difference between the current MPI call starting time and the previous MPI call ending time. We also save the wall time values before and after calling the real MPI function inside the wrapper. The difference between these two values is the communication time of the current MPI call.[2]

For each MPI call, we also specify the peer processor. For instance, the peer processor for an MPI_send is the destination processor. For those functions that do not have an explicit peer processor, we select an appropriate

[1]Even though we have used MPI profiling in our current implementation of the algorithm, similar methodology would work with other parallel programming languages such as OpenMP, UPC and Titanium.

[2]We assume that there is at most one application process per processor, so the wall time is a good proxy for communication time.

parameter or set it to a fixed number. For instance, for MPI_Reduce, we use the operation indicator parameter (i.e. MPI_Sum), for MPI_Wait we use a constant value and for MPI_Barrier we use the communication group.

The profiling data for each MPI function call is a 4-tuplet consisting of MPI function name, peer processor, computation and communication time. The profiling information is monitored and saved for each processor separately.

B. Abstracting MPI event logs

After generating MPI profiles, we focus on the first two entries (MPI function and peer process) and try to find patterns with multiple occurrence in a single profile sequence. From these patterns, we will later extract typical sequences that are common among all (or a subset) of processor profiles and hence their computation time are relatively comparable. Simple programs often have a semi-periodic pattern that can be easily detected. We can detect these patterns by simple signal processing techniques that are based on using the auto-correlation function of the profile sequence. In complex applications, common patterns still exists but they may not be periodic, hence we can not rely on simple signal processing algorithms to detect them.

Instead, we use the Sequitur algorithm to extract the frequent patterns. The Sequitur algorithm is a linear-time, on-line algorithm for producing a context-free grammar from an input string [14]. Sequitur takes a sequence of discrete symbols, which in our problem represent MPI function and peer process in the profile sequence, and produces a set of hierarchical rules that rewrite it as a context-free grammar. Note that there is no generalization in the produced grammar and it can only generate one string, which is the original sequence.

Sequitur produces a grammar based on repeated phrases (patterns) in the input sequence. Each repetition gives rise to a rule in the grammar. Hence, there are high frequency patterns among the rules derived by the Sequitur. At each step, Sequitur appends a new symbol from the input sequence to the end of the produced grammar up to that time. After adding each symbol, Sequitur modifies the grammar to maintain two properties:
1-*Diagram Uniqueness:* no pair of adjacent symbols appears more than once in the grammar.
2-*Rule Utility:* every rule is used more than once.

Table I shows simple examples (adopted from [14]) of the inputs and outputs to the algorithm. The original sequence symbols and Sequitur generated rules are shown with lower-case and capital letters respectively. Rule S generates the whole sequence. In example (a) Rule A corresponds to the pattern bc, which is repeated two times in the original sequence. In example (b), we have appended 6 new letters to the example (a) sequence. Sequitur forms A and B rules, which their corresponding sequences are abcdbc and bc respectively.

III. Requirements, Challenges and Solutions

We now elaborate on some of the problems and challenges in the design of interference detection algorithms. We use these issues to explain the reasoning behind the design of the main components of our interference detection algorithm.

Insensitivity to the network performance: Communication and computation times are the two main components that determine the overall performance of the parallel applications. While software interference increases the computation time it is not clear how it affects the communication time. Software interference may change when a processor

TABLE I

EXAMPLE SEQUENCES AND SEQUITUR GRAMMARS THAT REPRODUCE THEM

	Sequence	Grammar
(a)		
	S→abcdbc	S→aAdA
		A→ bc
(b)		
	S→abcdbcabcdbc	S→AA
		A→ aBdB
		B→ bc

data is ready and/or when it needs new data. This in turn changes the communication pattern between processors and consequently can remove or create network bottlenecks and hence reduce or increase the communication time. Therefore, it is crucial to monitor and profile the communication and computation times separately.

In our instrumentation, we use the wall time to compute the communication and computation time for each MPI call. The communication time is the time elapsed in the main MPI function which is called in the wrapper. The computation time is the time elapsed in the main application program between two MPI calls. In order to measure the computation time, we save the wall time before returning from the MPI wrapper function. The next time that an MPI wrapper is called it first subtracts the wall time from the stored values and saves it in the profile as the computation time for the current MPI call.

Appropriate time scale: In our approach, we monitor software performance to estimate software interference levels. However, software interference is not the sole cause of performance variations and we need to minimize other factors too. Kernel scheduler granularity is another reason for performance variation. Consider that under normal behavior it might be the case that 99 percent of the processing time is allocated to the application and 1 percent is consumed by the OS and demon processes. Suppose that the kernel scheduler granularity is 5ms. Roughly speaking, the processor spends 495ms (99×5) running the application and 5ms running other things. If the monitored computation times are in the kernel granularity order (5ms), even under normal operation we will observe significant variations. In general, the impact of factors other than interference becomes evident and more significant for smaller computation times. This is because the absolute value of performance variations remain the same, so for smaller monitored values the relative impact is higher.[3]

To reduce sensitivity to normal performance variations, we use *aggregation* and *filtering*. To increase the time duration of samples, instead of considering individual samples, we work with the aggregated patterns derived by Sequitur. Further, we only consider those computation time intervals that are above a minimum threshold by filtering

[3]While some applications might be subject to performance degradation due to these small changes in system overhead, we concentrate on identifying larger gained interference.

(dropping) those samples that are lower than threshold.

Another major source of performance variation is application itself. It is quite possible that between different occurrences of the same MPI profile pattern, the application runs different set of instructions which results in temporal variations in the monitored performance. Hence, instead of temporal, we use *spatial* correlations in our analysis.

Note that, we do not require that application performance on all processors be the same, or even they execute same instruction sequence. As long as, there are some common communication patterns among some processors and their relative performance does not change dramatically our algorithm performs well.

Insensitivity to Software Configuration: Software performance may depend on an application's parameters or inputs. This is problematic for interference detection algorithms that are based on statistical and machine learning approaches and rely on similarities in temporal performance patterns. Note that for each new software configuration, these algorithms need to learn the temporal patterns again. The number of possible configurations can be very large, which make it practically impossible to train the detection algorithm for all possible configurations. By using *relative* spatial rather than absolute temporal performance correlation, our algorithm is less sensitive to software configurations. In other words, we assume the patterns of communication remain from run to run, while the timing and repetitions count of the patterns may change.

IV. THE ALGORITHM

In this section, we go over the main modules of the detection system and explain how they interact with each other. The input to the system are processor profiles that contain application MPI calls, peer process, computation time and communication time as explained in section II. Further, we use the Sequitur algorithm to form the frequent patterns for one of the processors. We divide the system into three main modules which are: preprocessing, learning and detection units.

A. Preprocessing Unit

The preprocessing unit generates signals that are inputs to other units. First, we need to select a subset of the Sequitur patterns that are frequently present in all processor outputs. To that end, we sort the Sequitur patterns according to their length. We start from the longest pattern and determine how many matches we find in each profile sequence. To find a match we only check the first entry in the profile, i.e. the MPI function. Recall that the Sequitur patterns are generated based on the MPI functions and peer processor entries in a processor profile. However, since peer processor entry is processor dependent, we can not use it at this stage. We select the first L Sequitur patterns that detected as occurring frequently in all processor outputs and call them the Typical Sequences (TS).

For each TS, we form P processor signals where P is number of processors. A processor signal is computation time of matches with TS sequence in the profile. We also form two additional signals that consist of all processors signals average and standard deviation. Note that average and standard deviation are taken over all processors

samples at each time index. Therefore, we have $L \times (P+2)$ signals. Let $S(p,l,t)$ be the value of processor p and TS l signal at time index t and $AVG(l,t)$, $STDEV(l,t)$ be the average and standard deviation signals.

B. Detection Unit

The outputs of this unit are interference level signals and metrics. For each processor and TS we compute a dynamic threshold,

$$T(p,l,t) = AVG(l,t) + K(p,l) \times STDEV(l,t) \tag{1}$$

Whenever processor p and TS l signal is above the corresponding threshold, contention level assessment value should be increased. Our indicator function $I(p,l,t)$ specifies when signal is above threshold:

$$I(p,l,t) = \begin{cases} 1 & \text{if } S(p,l,t) > T(p,l,t) \\ 0 & \text{if } S(p,l,t) \leq T(p,l,t) \end{cases} \tag{2}$$

In section III, we elaborate on sensitivity to normal variations in application performance, and we explained that we have to filter out those samples that are less than a threshold value. We filter signal samples that are less than $0.5ms$. We take running averages of the filtered indicator functions call them Interference Level Signal ($ILS(p,l,t)$). Presence of interference on a processor increases $ILS(p,l,t)$ signal values. We use the interference level signals to come up with a single interference metric, $m(p)$. In this paper we use weighted average of the signals final values:

$$m(p) = \sum_{l} w(l) ILS(p,l,end), \tag{3}$$

where the weight $w(l)$ of typical sequences is proportional to sequence length. As we will see in the simulation results longer typical sequence samples are more stable and less sensitive to normal performance variations, hence we give them higher weights in the metric computation. Clearly there are alternative ways to define the metric. For instance, we could take a maximum instead of an average over all typical sequences, or instead of last signal value, we can use maximum or average of all signal values.

We should also note that the Detection unit tasks can be done in real time as the program is running. Since, we are taking average and standard deviation of computation time samples on processors that occur at the same time and we use running average of the (filtered) indicator function, we do not need any future information at time t. In this case, the interference metric is also defined over time as the weighted average of the last computed ILS signals. The only remaining parameter that we have to specify is $K(p,l)$ that we compute during the learning phase.

C. Learning Unit

Learning is performed once per application prior to production runs. The objective of this step is to select a good set of TS for each processor and to set parameters $K(p,l)$ properly. To that end, for each processor we gather two set of signals. The first set is under normal run of the application without any background activity. In the second set, we run a simple background interfering job on the node which we intend to tune its processor parameters. We have to set the parameters $K(p,l)$ so that we have maximum possible separation in the outcomes of two runs of the detection unit.

To that end, let $\overline{I^k_{Nor}}(p, l, end)$ and $\overline{I^k_{Int}}(p, l, end)$ be the final value of the running average filtered Indicator functions of two runs (without and with backgrounds respectively). Note that these values are also a function of threshold value and parameter k defined in (1). Our goal is to set k such that the difference between two run corresponding values is maximized:

$$K(p, l) = \arg\max_k \left(\overline{I^k_{Int}}(p, l, end) - \overline{I^k_{Nor}}(p, l, end)\right) \tag{4}$$

In practice, to find the optimal k value, in the learning phase, we run the detection unit algorithm on both learning runs profiles for different values of k, say $k = 0, 0.5, 1, \cdots, 10$. For each value of k, we count how many times signals corresponding to pure and BG runs are outliers, i.e. they are larger than the dynamic threshold given in equation 1. We select the value of k that maximizes the difference between outlier counts of two runs. This is the value for k that we use in the detecting phase. It is possible that for a particular processor p and a typical sequence l, for all values of k outlier counts for both runs are close to each other. In that case, we do not use typical sequence l in detection unit of processor p. These discarded sequences represent intervals that are not sensitive to interference.

Finally, we compute performance metric for multiple runs with the selected $K(p, l)$ values to determine what is the normal range of performance metric defined in (3) when there is no interfering program and how much increase we expect when there is interference.

V. Performance Evaluation

In this section, we evaluate and study performance of the proposed interference detection algorithm for 3 parallel applications. We run each application without interfering jobs (No BG), with continuously interfering back ground job (Full BG) on a single node (full background run) and with an on-off interfering job (On-off BG) on a single node that runs for around 10 seconds and then becomes idle (sleeps) for 5 seconds. We run applications on 32 nodes (64 processors) of a 64 node cluster. Each node is equipped with dual Intel Xeon 2.66 GHz processors. Nodes are connected via Myrinet network, and use the PBS batch scheduler to admit at most one application per node at a time. To run the application and the interfering job simultaneously, we reserve nodes in interactive mode, so we can login to a desired node and run the background job, while we are running the parallel application.

A. NAS Benchmarks

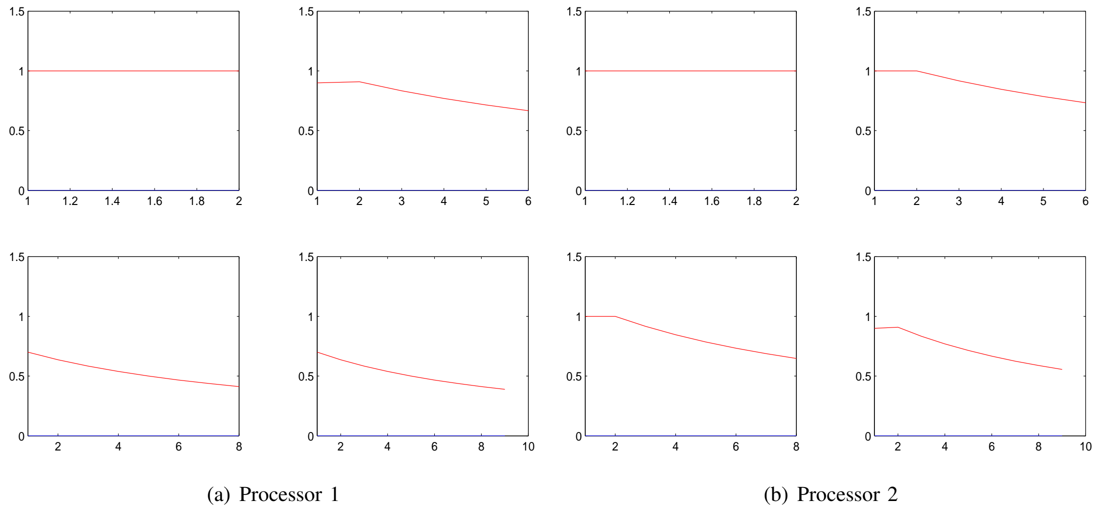
The NAS parallel benchmarks (NPB) [2] were developed at the NASA Ames research center to evaluate the performance of parallel and distributed systems. In this section, we evaluate performance of our system for interference detection on FT and CG. FT and CG are two of the NAS parallel kernels derived from computational fluid dynamics (CFD).

1) FT Benchmark: NAS FT benchmark is a three-dimensional heat equation solver that solves a certain partial differential equation using forward and inverse FFTs. This kernel performs the essence of many spectral codes. It is a rigorous test of long-distance communication performance. We run FT class C on 64 processors in our

TABLE II

LENGTH OF FT MPI PROFILE TYPICAL SEQUENCES

TS	1	2	3	4
lenght	16	8	4	2



(a) Processor 1 (b) Processor 2

Fig. 1. Interference level signal for FT kernel with full, and no background jobs

experiments. Structure of the FT MPI profile is very simple. It consists of 47 MPI calls and after a few initializing calls it periodically calls MPI_Alltoall and MPI_Reduce functions. From the Sequitur grammar we get 4 sequences, whose length are given in table II. Typical sequences are in fact 1, 2, 4 and 8 repetitions of the periodic pattern (MPI_Alltoall, MPI_Reduce). Even though the output pattern is very simple, this experiment confirms that Sequitur can detect highly regular periodic patterns.

We study performance of the interference detection algorithm on node 1. We first use two runs with continuous and no background jobs for the learning phase. Figure 1 shows the ILS signals for two other than learning runs. One of the two runs is with continuous background and the other one with no background job. *In all ILS plots, the x-axis is TS sample number and y-axis is computation time of samples.* As expected, the ILS signal is positive when there is interference and is zero when there is no interference. We can also see when there is a back ground job, the ILS signal associated with longer typical sequences are larger, and hence they are better indicators. This is due to the fact that for longer sequences, we aggregate a larger number of MPI profile samples and thus lessen the sensitivity to natural temporal performance variations.

Figure 2 shows ILS signals for two runs with on-off and no background jobs. ILS signals levels for on-off background job is positive. Signal levels for processor 1 in figure 2(a) is smaller than the full background case shown in figure 1(a) indicating lower interference levels. Table III summarizes FT runs interference metrics, which

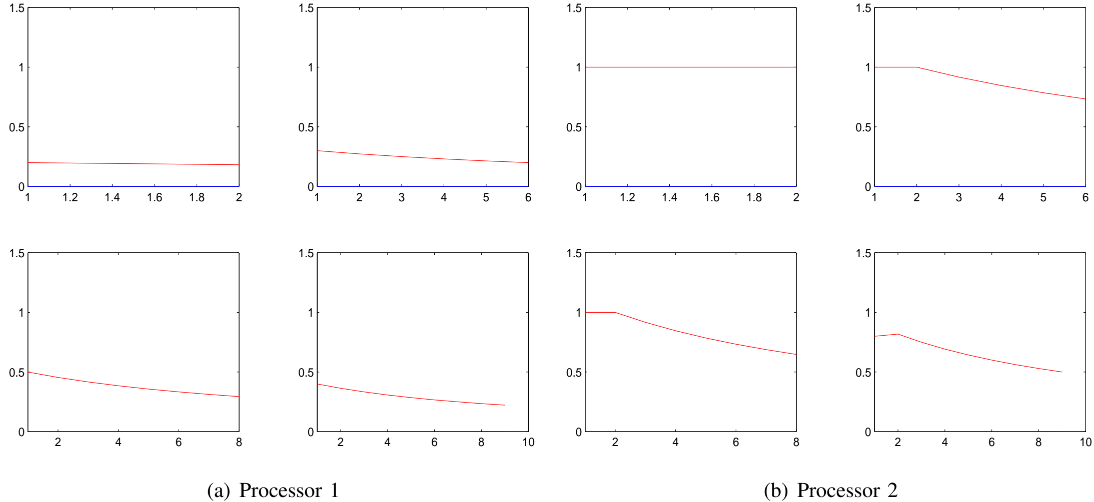


Fig. 2. Interference level signal for FT kernel with on-off, and no background jobs.

TABLE III
INTERFERENCE METRICS FOR FT RUNS

Full BG1	On-off BG1	No BG1
0.324	0.1284	0

confirms relation between interference level on the node and computed metric.

We now study sensitivity of our interference metric to variations in the on-off period of background jobs. Table IV contains the interference metric for different combinations of on and off periods of the background job. Our metric is able to detect interference for all of these cases (i.e. we get non-zero values for the metric). However, when the background job active time period is 1 second and in-active time (off times) period is larger than 20 the metric gets close to zero. In some cases, even though the metric correctly detects the interference, its value does not accurately represent intensity of the background job.

To understand the reason behind this fluctuation of the interference metric, we need to elaborate more on the FT computing pattern. In FT all processors regularly exchange data with each other, which results in significant communication time. In our experiments the FT program spends about twice as much time performing communication as computation. Note that if the background job uses CPU only during FT's communication periods, there will be no interference. Therefore, the interference metric value depends on: (1) intensity of background jobs, and (2) duration of over-laps between on-periods of background job and computation periods of the FT program. The first factor is deterministic, but the second one is in general probabilistic. That is why we observe fluctuations in the performance metric value.

In our experiments a complete run of FT takes about 2 minutes. Now, consider the case where the background

TABLE IV
INTERFERENCE METRICS FOR FT WITH DIFFERENT ON-OFF TIMES

	Off time(s)						
On time(s)	5	8	10	15	20	25	30
10	0.63	0.57	0.84	0.58	0.89	0.7	0.52
3	0.69	0.50	0.44	0.53	0.32	0.36	0.32
1	0.58	0.30	0.10	0.07	0.09	0.03	0.03

TABLE V
LENGTH OF CG MPI PROFILE TYPICAL SEQUENCES

TS	1	2	3	4	5	6	7	8
length	25428	12624	6312	3156	1578	789	720	480

job active period is 1 second (third row) and in-active period is 20 seconds (5th column). Then, there are around 6 on-periods of background job during a complete run of FT (2 minutes). The interference metric value depnds on how much overlap there is between the 6 active periods of the background job and FT's computation times (which are about 40 seconds out of the 120 seconds of FT complete run). The number of overlaps is clearly a random variable and the reason behind fluctuation of the results. Therefore, the important factor is not duty cycle of background jobs, but probability of overlap (interference) between computation time of program and interference.

In our experiments on-periods of background job is independent of the program. In many cases this is not true because the source of the interference could be initiated by the application. For instance, system calls by the application or memory leaks may cause interference. We speculate that for correlated interference, our system can be more effective than non-correlated interference since active periods that overlap has high probability even if the on-periods of the background jobs are small.

2) CG Benchmark: The CG benchmark uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication.

The MPI profile of CG is much more complex than FT. We ran CG class C on 64 processors and the MPI profiles contain around 60,000 MPI calls. Sequitur generates 23 rules (patterns) and we selected 8 longest ones as the typical sequences. Typical sequences lengths are given in table V. It is interesting to look at the autocorrelation function of the MPI profile sequence integer representation in figure 3. We know that auto-correlation of periodic

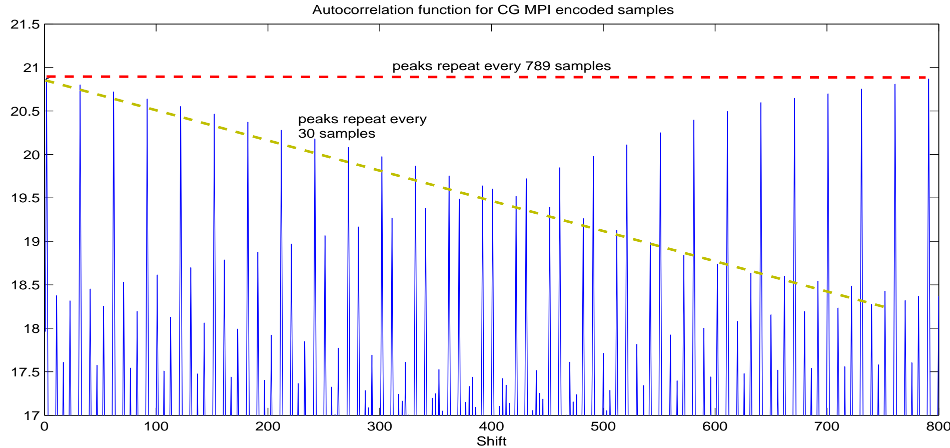


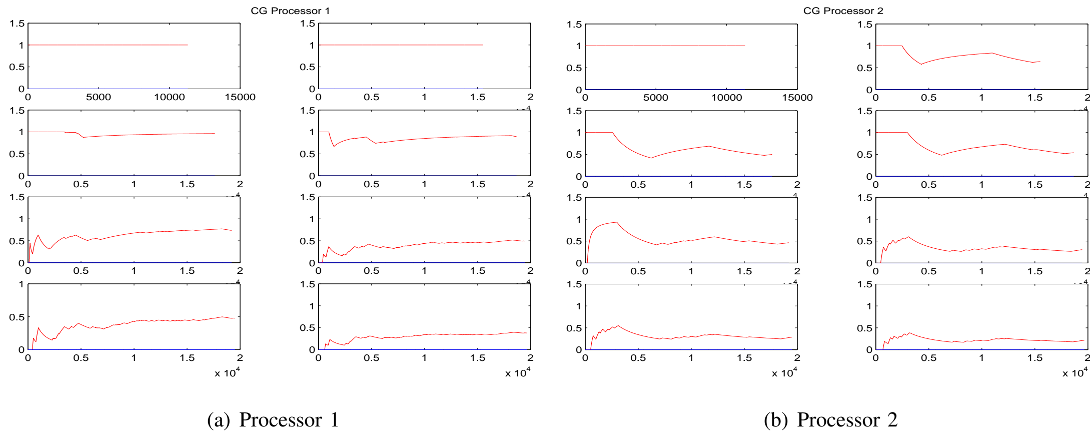
Fig. 3. Cross-correlation function of MPI encoded profile.

signals are periodic and their maximum values are at 0 and multiples of the period. In fact, auto-correlation is commonly used to find period of periodic and semi-periodic patterns in a signal. The autocorrelation function in figure 3 indicates presence of two periodic components. The first one repeats every 30 sample and the second one repeats after 789 samples as shown on the figure. The first six typical sequences are repetitions of the 789 samples pattern and the last two ones are repetitions of the 30 sample pattern. Hence, even in complex situations, where multiple semi-periodic patterns coexist, Sequitur successfully detects them.

After using two runs for learning and parameter tuning, we use two separate runs for evaluation of the system. Figure 4 shows ILS signals for two runs with continuous and no background jobs. ILS signals for the no background runs are always zero and for the continuous background they are positive. It is also clear that ILS signals of longer typical sequences are larger and hence more reliable for interference detection. Figure 5 shows the output of the system for two additional runs with the same type of background jobs. Even though the output signal is not the same, but it has the same characteristics i.e., it is zero for the no background run and longer typical sequences result in larger signal when there is a background job. From the FT and CG results, we may conclude that longer typical sequences are better candidates for interference detection. However, for a real time (on-line) version of the interference detection algorithm, longer typical sequences result in higher complexity and larger interference detection delay.

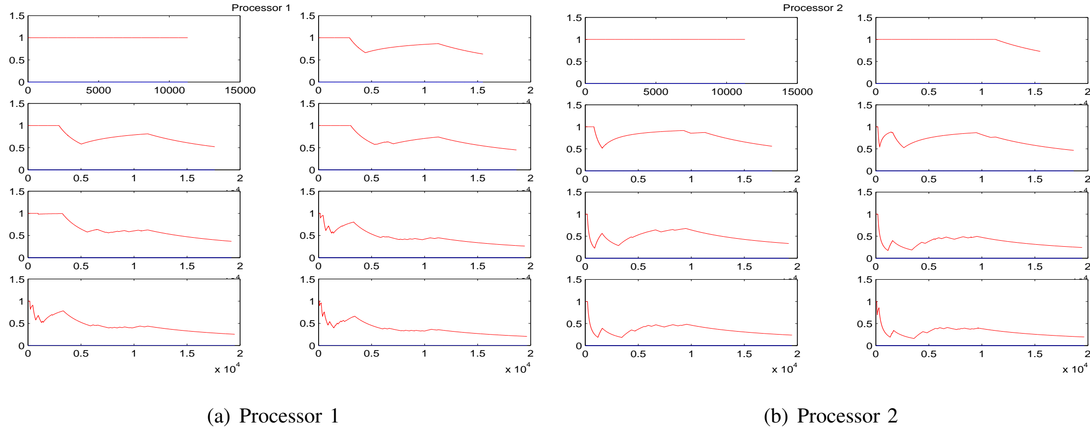
Figure 6 shows performance of the algorithm when there is on-off background jobs. The algorithm can still detect interference but sensitivity of the shorter typical sequences is higher and they are clearly less reliable.

Table VI summarizes CG runs interference metrics, which confirms relation between interference level on the node and computed metric. Namely two runs with full back ground job have greatest interference metric, followed by the run with on-off background and metric is zero for runs without background.



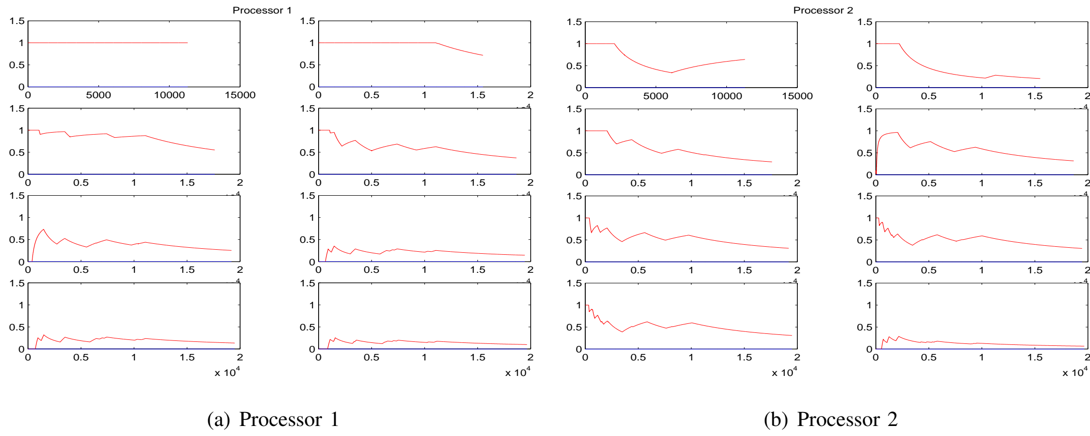
(a) Processor 1 (b) Processor 2

Fig. 4. Interference level signal for CG software with full, and no background job (first run)



(a) Processor 1 (b) Processor 2

Fig. 5. Interference level signal for CG software with full, and no background job (second run)



(a) Processor 1 (b) Processor 2

Fig. 6. Interference level signal for CG software on node 1 processors with on-off, and no Back Ground job.

TABLE VI
INTERFERENCE METRICS FOR CG RUNS

Full BG1	Full BG2	On-off BG1	No BG1	No BG2
0.78	077	0.61	0	0

TABLE VII
LENGTH OF POP MPI PROFILE TYPICAL SEQUENCES

TS	1	2	3	4	5	6	7	8	9	10
length	252	224	192	128	125	112	96	64	64	56

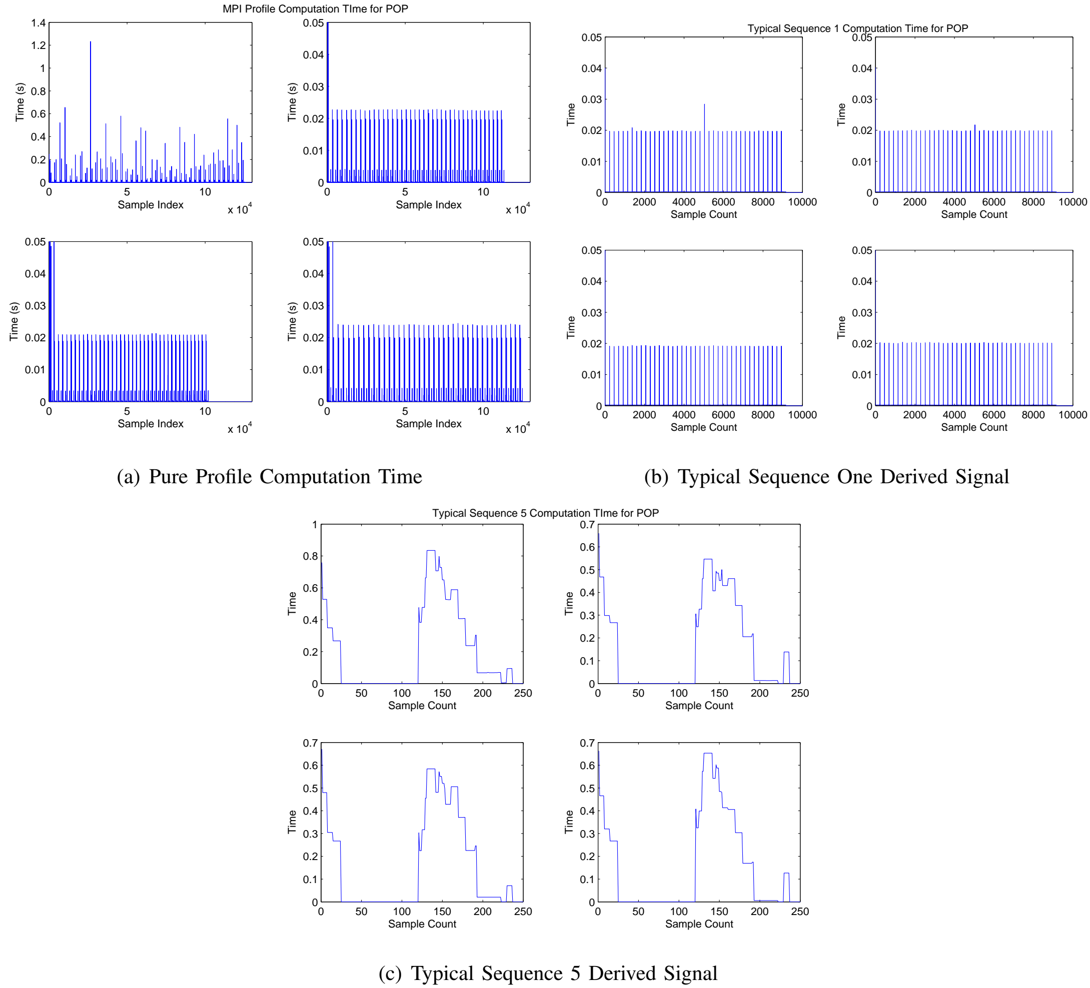
B. POP Program

The Parallel Ocean Program (POP) [9] was developed at Los Alamos National Laboratory and is descendant of the Bryan-Cox-Semnter class of ocean models first developed at the NOAA (National Oceanic and Atmospheric Administration) Geophysical Fluid Dynamics Laboratory in Princeton, NJ in the late 1960s [3]. POP is currently used by the Community Climate System Model (CCSM) as the ocean component. The model solves the three-dimensional primitive equations for fluid motions on a sphere using hydrostatic and Boussinesq approximations. Spatial derivatives are computed using finite-difference discretizations which are formulated to handle any generalized orthogonal grid on a sphere, including dipole and tripole grids.

Figure 7(a) depicts the profile of computation time for 4 processors[4]. Clearly the processor 1 (top-left) profile is different from other processors, hence it is not possible to detect interference by direct comparison of profiles. Figures 7(b) and 7(c) are computation time for two typical sequences occurrences in the same 4 processor profiles. The derived patterns for the 4 processors are comparable, and hence we can use spatial correlation to detect interference. Furthermore, due to profile sample aggregation in a single sequence, and elimination of samples that are not in typical sequences, the total number of samples are greatly reduced compare to the full event trace, which results in reduced processing time.

We started from Sequitur patterns derived from processor 1 profile. Then, we find 10 Typical sequences among them that are also present in other processor profiles and their length were long enough (larger than 50) to be considered. The length of the typical sequences are given in table VII. From each processor profile, we obtain 10 TS derived signals. Eight of the TS derived signals are similar to figure 7(b) and two of them (number 5 and 9) are similar to figure 7(b).

[4]We show results for 4 out of 64 processors due to space limit, however all processors follow the a similar pattern.



(a) Pure Profile Computation Time

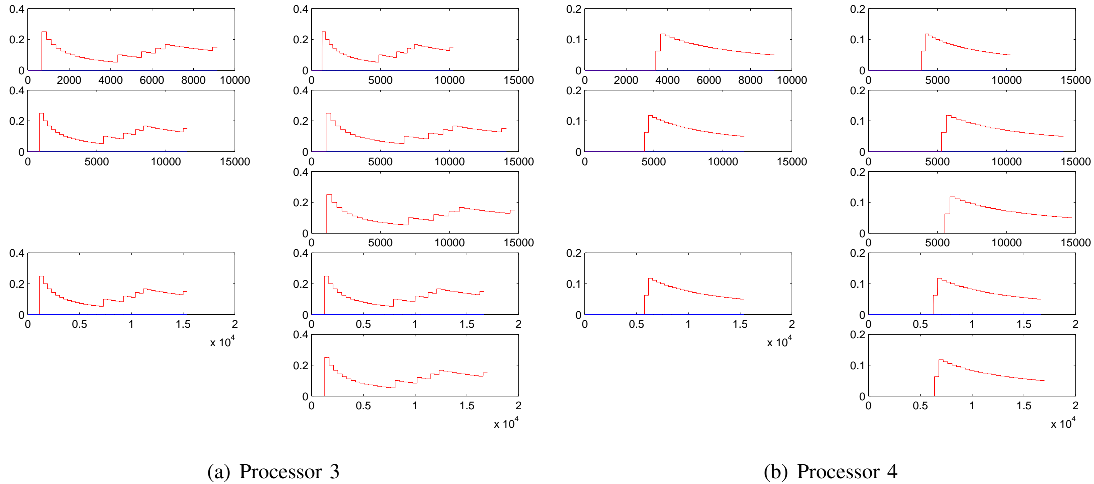
(b) Typical Sequence One Derived Signal

(c) Typical Sequence 5 Derived Signal

Fig. 7. Pure profile and typical sequence samples computation times for first four processors

1) Interference on Node 2: Here, we introduce interfering jobs on node 2. We first run POP, once with no back ground and once with a continuous background job on node 2. We then generate the TS derived signals of the two runs and use them in the learning phase to compute the interference detection system parameters. In the learning phase, if the maximum difference between the filtered indicators of two runs defined in equation 4 is close two zero then that typical sequence will not be used for detection. For node 2, this occurred for typical sequences 5 and 9, which have the output pattern similar to figure 7(b). Therefore, we focus and use the remaining 8 TS for node 2. The discarded sequences are those that show the least sensitivity to interference.

For evaluation, we generate two new profiles with and without background job on node 2. The POP configuration was similar to the first set used in the learning phase. Figure 8 shows the TS interference level signals for two processors (processor 3 and 4) of node 2. Recall from section IV that ILS signals are obtained by taking the running average of the TS indicator functions. The indicator functions specify the time intervals where the metrics for the event sequences are above the dynamic threshold. The ILS signals for the no background run is always zero, and



(a) Processor 3

(b) Processor 4

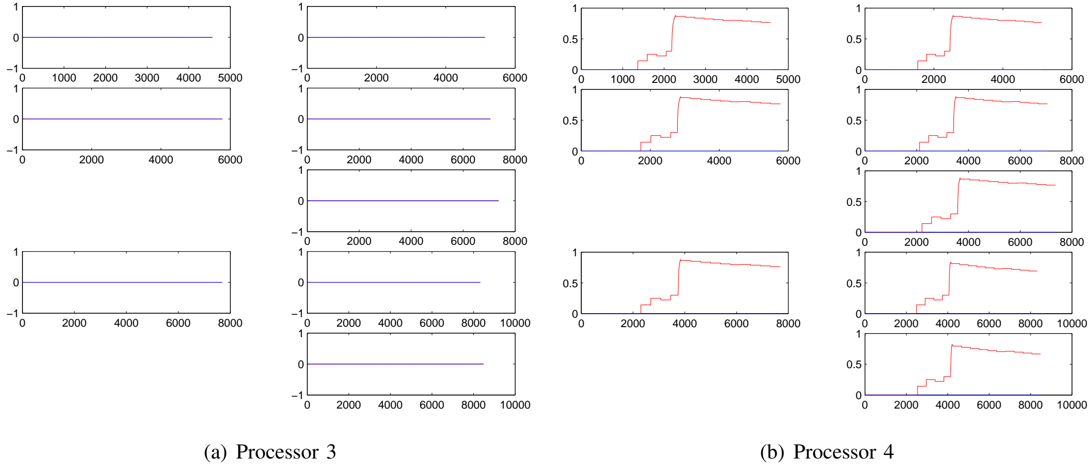
Fig. 8. Interference level signals for POP software when algorithm parameters are tuned to same POP configuration.

hence lies on the x-axis. The signal for the runs with continuous interference is shown in the plots. In general, larger signal values indicate a higher interference level.

For the next set of experiments, we changed the POP configuration to asses our tools ability to detect interference when the workload of an application changed. In particular, we changed the stop_count value from 40 to 20. We ran the algorithm 3 times with no, continuous and on-off background jobs. Note that ILS signals are derived using parameter values that were tuned to the previous POP configuration. Figure 9 shows the ILS signals for the runs with no and continuous back ground jobs. Again, ILS signals for no background runs remain zero, and hence are not visible on the plot. The result for a continuous back ground job on processor 3 is also zero. Note that we did not have any control on how the OS kernel schedules the background job on node processors. In this case, the back ground job is always scheduled on processor 4. Hence, there is no interference on processor 3. This result also indicates that we need to monitor performance of all processors running on multi-processor/multi-core node simultaneously and compute the performance metric for a node based on the ILS signals of both processors.

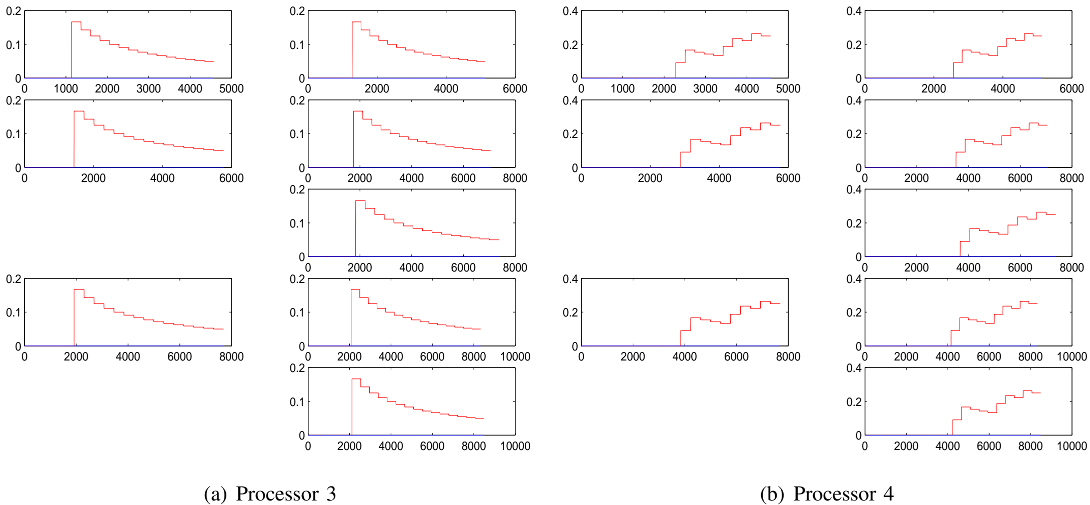
Figure 10 shows the result for the runs with and without on-off back ground jobs. The ILS signal for the no background run is always zero and hence lies on the x-axis. Since the back ground job is on-off and less intense, the interference level is lower than previous experiments. This is reflected in the ILS signal values. Contrary to the previous case, the interference is detectable on both processor outputs. This is a by-product of the kernel scheduler, not a characteristic of the application.

Table VIII provides the interference metric for all runs that we discussed in this section. BG stands for back ground, index 1 is used for runs with the learning phase configuration and index 2 for runs with alternative configuration. The metric correctly detects run with interference, since it value for runs with background job is positive and for runs without background it is zero. Notice that the absolute values of BG1 and BG2 data are not comparable since they are for different configurations. However, even though system is trained with a different configuration, it correctly orders the alternative configuration runs; Full BG2 metric is larger than On-off BG2 and



(a) Processor 3 (b) Processor 4

Fig. 9. Interference level signal for POP software when algorithm parameters are tuned to different POP configuration



(a) Processor 3 (b) Processor 4

Fig. 10. Interference level signal for POP software when algorithm parameters are tuned to different POP configuration

No BG2 metric is zero.

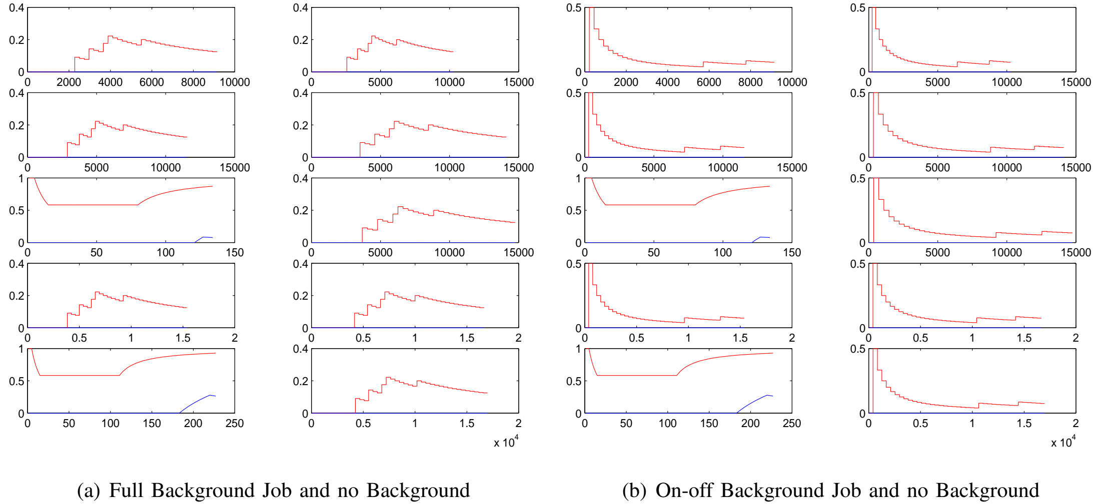
2) Interference on Node 1: We now introduce interfering job on node 1 and study performance and characteristics of the detection system. As shown in figure 7, processor one's profile and computation time signals are very different from other processors. As it is clear from figure 7(a), there is no clear temporal correlation in processor one's profile, nor is there spatial correlation between processor 1 and other processors profiles.

We used two POP runs (one with no background traffic and one with a continuous back ground job) for the learning and parameter tuning. Next, using the same configuration, we ran POP 3 times, with no, on-off and continuous back ground jobs on node 1. The ILS signals for 3 runs are shown in figure 11. Figure 11(a) shows the ILS signal for continuous and no background runs and figure 11(b) shows them for on-off and no background runs. The ILS signals for no background job on typical sequences 5 and 9 become temporary positive, however their

TABLE VIII

INTERFERENCE METRICS FOR POP RUNS WITH BACKGROUND ACTIVITY ON NODE 2. BG1 HAS THE SAME WORKLOAD AS THE TRAINING DATA AND BG2 IS WITH A DIFFERENT WORKLOAD.

Full BG1	Full BG2	On-off BG2	No BG1	No BG2
0.086	0.324	0.1284	0	0



(a) Full Background Job and no Background (b) On-off Background Job and no Background

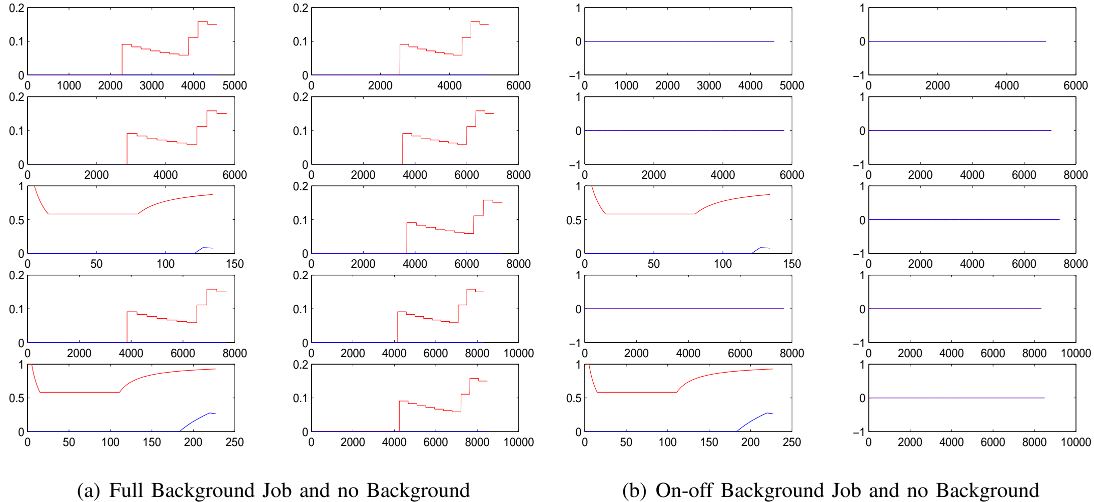
Fig. 11. Interference level signal for POP software when algorithm parameters are tuned for same POP configuration on node 1

level is much smaller and completely distinguishable from signal levels derived for runs with background jobs. It is also interesting that for processor 1 typical sequences 5 and 9 are more effective than other typical sequences, since they have larger signal values. This is exactly in contrast with what we observed for node 2 and reemphasizes that learning and parameter tuning should be done for each node separately.

Figure 12 shows the detection algorithm results for POP runs with a different configuration. For the on-off case shown in figure 12(b), only typical sequences 5 and 9 have positive values and can detect interference. Table IX provides interference metric for all runs with background job on node 1. The notation is similar to the previous case. The metric value for runs with background job is an order of magnitude larger than runs with no background, and can be used to correctly detect interference.

VI. RELATED WORK

Sequitur has been used by Chimbli to find frequent data-access sequences to abstract data reference localities in programs [8]. In a similar application it is used by Shen, et. al. to construct a phase hierarchy in order to identify composite phases and increases the granularity of phase prediction in a system that predicts locality phases in a program [16]. Larus used Sequitur algorithm in whole program paths (WPP), which is an approach to capturing



(a) Full Background Job and no Background

(b) On-off Background Job and no Background

Fig. 12. Interference level signals for POP software when algorithm parameters are not tuned for same POP configuration on node 1

TABLE IX

INTERFERENCE METRICS FOR POP RUNS WITH BACKGROUND ACTIVITY ON NODE 1

Full BG1	Full BG2	On-off BG2	No BG1	No BG2
0.23	0.25	0.19	0.01	0.01

and representing a program's dynamic control flow [12]. Sequitur is used in the second phase of WPP, where the collected traces are transformed into a compact and usable form by finding its regularity (i.e., repeated code).

Combined user and kernel performance analysis tools can be used to detect software interference. For instance KTAU is a system designed for kernel measurement to understand performance influences and the inter-relationship of system and user-level performance factors [13]. Because this system relies on direct instrumentation, there is always concern with measurement overhead and efficiency. Our solution can be used as a light overhead supporting mechanism to detect interference and to enable dynamic instrumentation mechanisms for detailed investigation.

Software aging and failure prediction systems generally require performance metrics that they can monitor and use. Chakravorty et. al. leverage the fact that such metrics for hardware devices exist [6]. Our approach can provide similar metrics for software components. Andrzajak and Silva in paper [1] and Castelli et. al., in paper [5] use system level measurements such as CPU, disk, memory, and network utilization for monitoring. Then they use statistical inference, machine learning or curve fitting based algorithm to predict software aging. However, their analysis is based on temporal behavior analysis of the system, which is subject to change if software configuration changes. Therefore, they need to have a learning phase for each new configuration. Further, since they do not measure software performance directly it is not clear how they can distinguish other factors that may contribute to higher utilizations.

VII. CONCLUSION

We presented an automated software interference detection algorithm for parallel programs. Our approach takes advantage of spatial correlation present between processor communication profiles. We use the Sequitur algorithm to abstract and extract common frequent patterns in processor profiles. We evaluated the performance of the system in detecting interference for different parallel programs and studied its sensitivity to changes in the on-off period degree of interference. We also measured the ability of the algorithm to detect interference when the workload differs from the training workload.

Interference detection algorithms and metrics have many application in autonomous computing systems, including failure prediction, performance optimization, proactive management of software aging and detection of inconsistency in system software. Another, more novel application of interference metrics is to score and evaluate applications on their sensitivity to interference. Interference sensitivity analysis enables us to design applications that are more robust and thus have predictable performance.

REFERENCES

[1] Andrzejak, A., L. M. Silva, "Deterministic Models of Software Aging and Optimal Rejuvenation Schedules," CoreGRID Technical Report TR-0047.

[2] Bailey, D., et. al., "The NAS Parallel Benchmarks," *RNR Technical Report,* RNR-94-007, March 1994.

[3] Bryan, K., "A numerical method for the study of the circulation of the world ocean," *Journal of Computational Physics,* vol. 135, no.2 pp. 154-169, 1997.

[4] Cassidy, K.J., K.C. Gross, A. Malekpour, "Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers" *Proc. Int. Conf. on Dependable Systems and Networks DSN'02,* 2002.

[5] Castelli, V., et. al., "Proactive Management of Software Aging," *IBM Journal of Res. and Dev.,* pp. 311-332, vol. 45, no. 2, 2001.

[6] Chakravorty, S., et. al., "Proactive Fault Tolerance in Large Systems," *HPCRI workshop in conjunction with HPCA'05,* 2005.

[7] Lu, C.-D., "Scalable Diskless Checkpointing for Large Parallel Systems," Ph.D. Dissertation, Univ. of Illinois at Urbana-Champain, 2005.

[8] "Efficient Representation and Abstraction for Quantifying and Exploiting Data Reference Locality," *Proceedings of ACM SIGPLAN Conf. on Programming and Language Design and Implementation,* Snowbid, Utah, June 2001.

[9] Dukowicz, J.K., R. D. Smith, and R. C. Malone, "A Reformulation and Implementation of the Bryan-Cox-Semtner Ocean Model on the Connection Machine," *Journal of Atmospheric and Oceanic Technology,* vol. 10, no. 2, pp. 195-208, Apr. 1993.

[10] Gross, K.C., V. Bhardwaj, R. Bickford, "Proactive Detection of Software Aging Mechanisms in Performance Critical Computers," *Proceedings of 27th Annual NASA Goddard/IEEE Software Engineering Workshop SEW'02,* Dec. 2002.

[11] Jones, T., et. al., "Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System," *Proc. ACM/IEEE conf. on Supercomputing SC'03,* Washington, DC, 2003.

[12] Larus, J. R., "Whole Program Paths," *Proc. of SIGPLAN'99 Conf. on Programming Language Design and Implementation,* Atlanta, GA, May 1999.

[13] Nataraj, A., A. D. Malony, S. Shende, A. Morris, "Kernel-Level Measurements for Integrated Parallel Performance Views: The KTAU Project," *Proc. of IEEE Int. Conf. on Cluster Computing,* pp. 1-12, Sep. 2006.

[14] Nevill-Manning, C.G., I.H. Witten, "Compression and Explanation Using Hierarchical Grammars," *The Computer Journal,* vol. 40, pp. 103-116, 1997.

[15] Petrini, F., D. Kerbyson, S. Pakin, "The Case of Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," *Proc. ACM/IEEE conf. on Supercomputing SC'03,* Washington, DC, 2003.

[16] Shen, X., Y. Zhong, C. Ding, "Locality Phase Prediction," *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS'04,* pp. 165-176, Boston, Oct. 2004.

[17] Parallel Ocean Program (POP), http://climate.lanl.gov/Models/POP/