DProf: Distributed Profiler with Strong Guarantees

ZACHARY BENAVIDES, University of California, Riverside, USA
KEVAL VORA, Simon Fraser University, British Columbia, Canada
RAJIV GUPTA, University of California, Riverside, USA

Performance analysis of a distributed system is typically achieved by collecting profiles whose underlying events are timestamped with unsynchronized clocks of multiple machines in the system. To allow comparison of timestamps taken at different machines, several timestamp synchronization algorithms have been developed. However, the inaccuracies associated with these algorithms can lead to inaccuracies in the final results of performance analysis. To address this problem, in this paper, we develop a system for constructing distributed performance profiles called DProf. At the core of DProf is a new timestamp synchronization algorithm, FreeZer, that tightly bounds the inaccuracy in a converted timestamp to a time interval. This not only allows timestamps from different machines to be compared, it also enables maintaining strong guarantees throughout the comparison which can be carefully transformed into guarantees for analysis results. To demonstrate the utility of DProf, we use it to implement dCSP and dCOZ that are accuracy bounded distributed versions of Context Sensitive Profiles and Causal Profiles developed for shared memory systems. While dCSP enables user to ascertain existence of a performance bottleneck, dCOZ estimates the expected performance benefit from eliminating that bottleneck. Experiments with three distributed applications on a cluster of heterogeneous machines validate that inferences via dCSP and dCOZ are highly accurate. Moreover, if FreeZer is replaced by two existing timestamp algorithms (linear regression & convex hull), the inferences provided by dCSP and dCOZ are severely degraded.

CCS Concepts: • **Software and its engineering** → **Software performance**; • **Computing methodologies** → *Distributed algorithms.*

Additional Key Words and Phrases: Performance Debugging, Distributed Systems, Timestamp Synchronization

ACM Reference Format:
Zachary Benavides, Keval Vora, and Rajiv Gupta. 2019. DProf: Distributed Profiler with Strong Guarantees. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 156 (October 2019), 24 pages. https://doi.org/10.1145/3360582

1 INTRODUCTION

Improving the performance of a distributed program requires careful understanding of the relative behavior of its components. This behavior can be captured by timestamping important events (e.g., arrival at a barrier etc.). Once recorded, these timestamps can be analyzed to gain insight into the behavior of the system. While timing information can be easily captured on any single machine, efficiently synchronizing local timestamps to enable their comparison is a challenging task. The local clocks differ not only in their offset but also in frequency, making direct comparison of their timestamps a problem.

Authors' addresses: Zachary Benavides, Department of Computer Science and Engineering, University of California, Riverside, California, USA, zbena001@ucr.edu; Keval Vora, School of Computing Science, Simon Fraser University, British Columbia, British Columbia, Canada, keval@cs.sfu.ca; Rajiv Gupta, Department of Computer Science and Engineering, University of California, Riverside, California, USA, gupta@cs.ucr.edu.

2475-1421/2019/10-ART156
https://doi.org/10.1145/3360582

The problem of meaningfully comparing local timestamps generated in a distributed system is known as *timestamp synchronization*. The earliest timestamp synchronization algorithms were proposed by Duda et al. [Duda et al. 1987]. These algorithms assume the existence of a linear conversion function between each pair of clocks, and use statistical methods to estimate the parameters of these functions. Others have developed improvements on these basic techniques to increase their accuracy or efficiency [Becker 2010; Dunigan 1992; Hofman and Hilgers 1998; Maillet and Tron 1995; Rabenseifner 1997]. Poirier et al. [Poirier et al. 2010] showed that by viewing the problem through the lens of linear programming, it is possible to strongly bound the error in the conversion process though at the cost of considerable inefficiency.

In this paper, we develop a tool for performance profiling of distributed systems called DProf. Critical to the utility of any tool are its accuracy and cost. Table 1 summarizes our findings when using existing timestamp synchronization techniques (detailed results in §3.5). We observe that most existing techniques provide low accuracy with no bounds on the error in their results, hence providing no insights into how accurate their results may be. While Huygens [Geng et al. 2018] provides high accuracy, it synchronizes clocks (e.g. the PTP Hardware Clocks) in the NICs which limits its applicability for programmable performance profiling. Even though AR-Convex Hull [Poirier et al. 2010] reports bounds on synchronized timestamps, its accuracy is sensitive to the volume of events timestamped during tracing, and furthermore the overhead for synchronizing large number of timestamps makes it impractically expensive.

At the core of DProf is a new error-bounded timestamp synchronization algorithm, FreeZer in Figure 1, which bridges the performance gap between the error-bounded but inefficient Poirier's algorithm [Poirier et al. 2010], and the family of efficient but unbounded algorithms inspired by Duda et al. [Duda et al. 1987]. Thus, analyses built with DProf as their foundation are both accurate and affordable. We demonstrate this by lifting two recent performance analysis techniques, namely *Context Sensitive Profiling* [Benavides et al. 2017] and *Causal Analysis* [Curtsinger and Berger 2015], originally developed for shared memory systems into the distributed domain. We refer to these distributed versions as dCSP and dCOZ (see Figure 1).

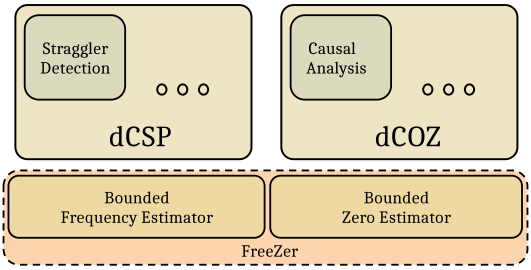

Fig. 1. DProf overview.

While dCSP enables user to ascertain existence of a performance bottleneck, dCOZ estimates the expected performance benefit from eliminating that bottleneck. Experiments with three distributed applications on a cluster of 8 heterogeneous machines validate that inferences via dCSP and dCOZ are highly accurate. Furthermore, we demonstrate the strength of FreeZer by comparing it with two existing timestamp algorithms (linear regression and convex hull [Duda et al. 1987]); our results show that using these techniques, when used instead of FreeZer, provided severely degraded inferences.

Various other performance analysis solutions have been developed for shared memory parallel systems including critical path analysis [Böhme et al. 2012; Hollingsworth 1996; Yang and Miller

Table 1. Key characteristics of existing synchronization solutions and our Freezer technique.

	Accuracy	Error Bounds	Synchronization Overhead
Linear Regression [Duda et al. 1987] [Maillet and Tron 1995] [Rabenseifner 1997] [Becker 2010] [Dunigan 1992] [Hofman and Hilgers 1998]	Low	No	Low
Two-point Alg. [Ashton 1995]	Low	No	Low
Controlled Logical Clock [Rabenseifner 1997]	Low	No	Low
Convex Hull [Duda et al. 1987]	Low	No	Medium
Huygens [Geng et al. 2018]	High	No	Low
AR-Convex Hull [Poirier et al. 2010]	Variable	Yes	High
Freezer	**High**	**Yes**	**Low**

1988], logical zeroing [Hollingsworth and Miller 1994], TaskProf [Yoga and Nagarakatte 2017], Quartz [Anderson and Lazowska 1990], Parashares [Kambadur et al. [n.d.]], IPS-2 [Miller et al. 1990], Bottle Graphs [Bois et al. 2013], etc. that can be used to develop different insights into parallel program behavior. DProf can be used to implement these capabilities for analyzing distributed executions. Performance analyses like HPCToolkit [Adhianto et al. 2008], Scalasca [Geimer et al. 2010], VAMPIR [Nagel et al. 1996], Log2 [Ding et al. 2015], [Liu and Mellor-Crummey 2011], [Shende and Malony 2006], etc. can similarly be enhanced to provide error bounds for computed results.

The remainder of the paper is organized as follows. We introduce the problem statement and notation in §2. Our synchronization algorithm called FreeZer and its evaluation is presented in §3. The use of FreeZer to build the DProf distributed profiling framework and further its use to implement dCSP and dCOZ analyses is presented in §4. In §6 we present case studies to validate the effectiveness of dCSP and dCOZ. Conclusions are presented in §7.

2 TIMESTAMP CONVERSION PROBLEM

Performance analysis of distributed programs involves capturing relevant program events along with their timing information to deduce underlying performance bottlenecks. While timing information can be easily gathered across events within a single machine, absence of a global clock in any distributed setting makes it challenging to accurately capture the timing information for events spanning across multiple machines.

To facilitate comparison across event timings that are captured locally on different machines, we first convert them to a common timeline and then directly compare these converted times. We first formally define the timestamp conversion process, and then discuss the challenges involved in accurately converting event timestamps.

Distributed Timing Model. Let $N = \{n_0, n_1, \ldots, n_k\}$ be the set of nodes in a distributed system with k compute nodes (machines) and let $C = \{c_0, c_1, \ldots, c_k\}$ be the set of clocks such that $\forall c_i \in C, c_i$ is the clock for n_i. With absence of a global clock, the clocks in C are not synchronized with each other and operate at different frequencies. Today's multicore machines have a separate clock associated with each core, hence, each core becomes a separate node in our distributed system. However, this model can be used for distributed settings where all cores within a socket or all cores within a machine rely on single synchronized clock, by simply modeling each socket or each machine as a node in the distributed system.

Let $TS = \{ts_0, ts_1, ..., ts_k\}$ be the captured timestamps such that $\forall ts_i \in TS, ts_i$ is captured on c_i in units of clock cycles. Let $rt(ts_i)$ be an abstract oracle that maps ts_i to the real time. Since the captured timestamps are relative to their respective local epochs, $\forall ts_i,\ ts_j \in TS, ts_i$ may or may not be equal to ts_j even though $rt(ts_i) = rt(ts_j)$. Timestamp conversion is the process to convert all the local timestamps to a common base so that they become directly comparable. In particular, a function $conv(ts_i)$ that converts any given timestamp to a common base must achieve the following $\forall\ ts_i,\ ts_j \in TS$:

$$\begin{gathered} rt(ts_i) \oplus rt(ts_j) \leftrightarrow conv(ts_i) \oplus conv(ts_j) \\ |rt(ts_i) - rt(ts_j)| = |rt(conv(ts_i)) - rt(conv(ts_j))| \end{gathered} \tag{1}$$

where $\oplus \in \{=, \neq, >, <, \geq, \leq\}$. The first relation ensures that the ordering of timestamps becomes comparable after conversion while the second relation ensures that the timing information is preserved across the converted timestamps. For simplicity, we assume that our common base is provided by n_0, i.e., $\forall ts_i \in TS \setminus \{t_0\}, conv(ts_i)$ converts the ts_i in terms of c_0 such that $conv(ts_i)$ is directly comparable to ts_0. To convert ts_i in terms of c_0, we need two measures:

(1) Frequencies of c_i and c_0, referred to as $fr(c_i)$ and $fr(c_0)$ respectively.
(2) Special timestamps, z_i and z_0 that are captured on c_i and c_0 respectively at *the same moment of real time*. We call these special timestamps as *zeros*.

With the above measures, ts_i can be converted in terms of c_0 using $conv(ts_i)$ defined as:

$$conv(ts_i) = (ts_i - z_i) \times \frac{fr(c_0)}{fr(c_i)} + z_0 \tag{2}$$

Problem. The main challenge is to compute $fr(c_i)$ and z_i measures with high precision to ensure that the relations in Eq. 1 are maintained correctly. While accurate estimates can be computed when low-precision frequency and zero measures are adequate, the difficulty increases quickly with rising precision requirement.

Our Approach. Since in practice we cannot fully control the accuracy of frequency and zero measures to a precision high enough that guarantees correct comparison of timestamps, we tightly bound the inaccuracies in these estimates so that the timestamp conversion can provide strong guarantees for converted values. This means, $fr(c_i)$ and z_i are now in interval domains with strong lower and upper bounds and $conv(ts_i)$ converts ts_i from time domain to interval domain to preserve strong bounds over the converted timestamp value, hence capturing conversion inaccuracies.

As summarized in Table 1, our solution (Freezer) is aimed to provide high accuracy with tight error bounds and low synchronization overheads. In §3.5, we will compare Freezer with existing solutions to showcase its high error-bounded accuracy.

3 FREEZER: BOUNDED FreQUENCY & ZerO ESTIMATION

In this section, we analyze the inaccuracies involved in estimating frequency and zero measures, and develop solutions to tightly bound the estimates for strong profiling guarantees.

Algorithm 1: Computing absolute frequency of c_i

```
$ts_1 \leftarrow$ timestamp()   // e.g., using x86 TSC
sleep($t$)
$ts_2 \leftarrow$ timestamp()
$f_i(t) \leftarrow (ts_2 - ts_1)/t$
```

3.1 Estimating Relative Frequencies

A straightforward way to compute frequency estimates is to individually determine the absolute frequency at which each clock is operating by counting the number of cycles elapsed for a predetermined amount of time, as shown in Algorithm 1. Here counting cycles can be achieved using local x86 timestamp counters (TSC) – the use of the invariant TSC, widely available on modern x86 Intel and AMD systems, provides a constant frequency time source even in presence of dynamic frequency scaling. While this solution may suffice for capturing less accurate frequency estimates, computing highly accurate frequencies requires precisely measuring a given amount of time. Unfortunately, exact measurements are impossible due to the inherent non-determinism in today's computing systems. For instance, the `sleep` commands only guarantee suspension for *at least* (*and not exactly*) the amount of time provided by the user. Furthermore, there is no guarantee of the closeness of the amount of time that is actually slept to the amount of time that is requested. Although increasing t to a high enough value in Algorithm 1 can arbitrarily mitigate these inaccuracies, determining a sufficient value of t for some user defined tolerance is infeasible for reasons already discussed.

Since we want to compare the timestamps of distributed events (i.e., timestamps captured across on clocks), we aim to compute a relative measure between multiple clocks. Comparing frequencies of multiple clocks can be achieved by directly computing *relative frequencies* between those clocks. Furthermore, the computation for relative frequencies can be carefully controlled to provide stronger guarantees which become useful to capture the deterministic error bound. By carefully computing relative instead of absolute frequencies, we can not only arbitrarily mitigate the sources of inaccuracy, but we can do so in a way that bounds the accuracy of our estimate.

Similar to computing absolute frequencies, a simple way to compute relative frequencies across a pair of clocks is to measure the number of cycles spent for the same amount of real time t on those clocks (see Algorithm 2). In this case, $f_{j \to i}(t)$ (computed on line 5) is the relative frequency to convert durations captured on c_j to the units of c_i. We can capture the inaccuracies introduced by the non-deterministic factors as follows:

$$f_{j \to i}(t) = \frac{fr(c_i) \times (t + a)}{fr(c_j) \times (t + b)} \tag{3}$$

where $fr(c_i)$ is the frequency of clock i, and variables a and b represent the inaccuracies introduced by the sleep calls (i.e., a call to `sleep(t)` will introduce a delay of $t + a$ seconds for some positive value of a). We can carefully eliminate the effect of inaccuracies by taking the limit as t tends to ∞ as described next. Let A and B be positive constants such that $a < A$ and $b < B$. Consider

Algorithm 2: Computing relative frequency of c_i and c_j using their absolute frequencies

```
$c_i$
1: $ts_1 \leftarrow$ timestamp()
2: sleep($t$)
3: $ts_2 \leftarrow$ timestamp()
4: $diff_i \leftarrow ts_2 - ts_1$

$c_j$
1: $ts_1 \leftarrow$ timestamp()
2: sleep($t$)
3: $ts_2 \leftarrow$ timestamp()
4: $diff_j \leftarrow ts_2 - ts_1$

5: $f_{j \to i}(t) \leftarrow diff_i \,/\, diff_j$
```

continuous functions $f^{+}_{j\to i}$ and $f^{-}_{j\to i}$ as defined below:

$$f^{+}_{j\to i}(t) = \frac{fr(c_i) \times (t + A)}{fr(c_j)(t)} \quad \Bigg| \quad f^{-}_{j\to i}(t) = \frac{fr(c_i)(t)}{fr(c_j) \times (t + B)}$$

By L'Hopital's rule, we have:

$$\lim_{x\to\infty} f^{+}_{j\to i}(x) = \lim_{x\to\infty} f^{-}_{j\to i}(x) = \frac{fr(c_i)}{fr(c_j)}$$

Since $f^{-}_{j\to i}(t) \leq f_{j\to i}(t) \leq f^{+}_{j\to i}(t)$, we can conclude that:

$$\lim_{t\to\infty} f_{j\to i}(t) = \frac{fr(c_i)}{fr(c_j)}$$

Hence, the impact of inaccuracies when calculating relative frequencies can be eliminated by increasing t to a very high value (as is the case with absolute frequencies). While exact relative frequencies cannot be practically achieved, we can determine the rate at which relative frequencies become accurate as t increases. The error in a frequency estimate can be computed based on Eq. 3 as:

$$\text{Error} = \frac{fr(c_i) \times (t + a)}{fr(c_j) \times (t + b)} - \frac{fr(c_i)}{fr(c_j)}$$

To achieve frequency estimates that are accurate within a tolerance T, we have:

$$\frac{fr(c_i) \times (t + a)}{fr(c_j) \times (t + b)} - \frac{fr(c_i)}{fr(c_j)} < T$$
$$\implies t > \frac{1}{T}\left(\frac{fr(c_i)}{fr(c_j)}(a - b)\right) - b$$

In other words, the length of time that is required to achieve a desired accuracy grows linearly with desired accuracy.

Bounding Relative Frequencies. Now we show how the error in computed frequency estimates can be bounded. To determine the accuracy of computed frequency estimates, we bound the relative frequency to intervals that contain the true relative frequency. The challenge is to make these intervals as small as possible so that the residual inaccuracies do not meaningfully impact the profiling results.

The key insight is that instead of trying to measure the same duration on each clock, we intentionally measure durations which differ in a specific way. In particular, an upper bound can be computed by measuring a larger interval on the first clock compared to that on the second clock, and a lower bound can be computed by reversing this procedure, i.e., by measuring a smaller interval on the first clock compared to that on the second clock. Hence, we can achieve lower bound $f^{L}_{j\to i}(t)$ and upper bound $f^{U}_{j\to i}(t)$ defined as follows:

$$f^{L}_{j\to i}(t) = \frac{fr(c_i) \times t}{fr(c_j) \times (t + \epsilon)} \quad \Bigg| \quad f^{U}_{j\to i}(t) = \frac{fr(c_i) \times (t + \epsilon)}{fr(c_j) \times t}$$

$$f^{L}_{j\to i}(t) < f_{j\to i}(t) < f^{U}_{j\to i}(t)$$

where ϵ is an arbitrary variation to increase time intervals ($\epsilon > 0$). While any positive ϵ value can be used to compute $f^{L}_{j\to i}(t)$ and $f^{U}_{j\to i}(t)$, smaller ϵ values will produce tighter bounds.

Algorithm 3: Bounding relative frequency of c_i and c_j by introducing causal dependencies

```
c_i                                      c_j
$ts_1 \leftarrow$ timestamp()            receive_from($c_i$)
send_to($c_j$)                           $ts_1 \leftarrow$ timestamp()
receive_from($c_j$)                      sleep($t$)
$ts_2 \leftarrow$ timestamp()            $ts_2 \leftarrow$ timestamp()
$diff_i \leftarrow ts_2 - ts_1$          send_to($c_i$)
                                         $diff_j \leftarrow ts_2 - ts_1$

$f^U_{j \to i}(t) \leftarrow diff_i \,/\, diff_j$

receive_from($c_j$)                      $ts_1 \leftarrow$ timestamp()
$ts_1 \leftarrow$ timestamp()            send_to($c_i$)
sleep($t$)                               receive_from($c_i$)
$ts_2 \leftarrow$ timestamp()            $ts_2 \leftarrow$ timestamp()
send_to($c_j$)                           $diff_j \leftarrow ts_2 - ts_1$
$diff_i \leftarrow ts_2 - ts_1$

$f^L_{j \to i}(t) \leftarrow diff_i \,/\, diff_j$
```

In order to ensure the epsilon is applied to the correct machine, we introduce a causal dependency between the appropriate clock measurements using communication primtives (i.e. send/recv). Algorithm 3 shows how we compute bounds $f^L_{j \to i}(t)$ and $f^U_{j \to i}(t)$.

With $f_{j \to i}(t)$ bounded by $f^L_{j \to i}(t)$ and $f^U_{j \to i}(t)$, the error in $f_{j \to i}(t)$ is bounded by $f^U_{j \to i}(t) - f^L_{j \to i}(t)$. If we were intersted in minimizing the maximum error, we could take the midpoint of this interval as a point estimate. However, directly using the midpoint will lose the information about the direction of the error. Hence, instead of using the midpoint value, we directly use the captured bounds ($f^L_{j \to i}(t)$ and $f^U_{j \to i}(t)$) while processing runtime profiles to provide strong end-to-end guarantees over profiling results.

3.2 Estimating Zeros

To analyze timestamps measured via different clocks, we also need to compute comparable timestamps across those clocks that capture the same moment in real time. These comparable timestamps will be used to shift all the remaining timestamps so that the resulting time scale starts from zero. Hence, we refer to these pairs of comparable timestamps as *zeros*.

Algorithm 4: Computing zero for c_i and c_j using causal dependencies

```
c_i                                      c_j
receive_from($c_j$)                      $ts_1 \leftarrow$ timestamp()
$ts_2 \leftarrow$ timestamp()            send_to($c_i$)
send_to($c_j$)                           receive_from($c_i$)
                                         $ts_3 \leftarrow$ timestamp()

$zero_{j \to i} \leftarrow <(ts_1, ts_3), ts_2>$
```

Algorithm 4 shows how we can compute zeros across a pair of clocks. We do so by trapping a timestamp from one clock between a pair of timestamps taken on the other clock. To achieve this, we introduce causal dependencies that ensure that ts_2 on c_i is captured between ts_1 and ts_3 on c_j. Hence, $zero_{j \to i} = \textless (ts_1, ts_3), ts_2 \textgreater$ effectively means:

$$rt(ts_1) < rt(ts_2) < rt(ts_3)$$

where $rt(ts_i)$ maps ts_i to the moment in real time when it was captured. While the zero can be approximated by picking the midpoint so that it becomes $\textless (ts_1 + ts_3)/2, ts_2 \textgreater$, similar to relative frequency bounds, we explicitly maintain the bounds for zeros so that they can be directly used along with relative frequency bounds to provide strong guarantees over profiling results. The lower bound of the zero (i.e., ts_1) is matched with that of relative frequency (i.e., $f^L_{j \to i}(t)$) to achieve a lower bound of the converted timestamp, while the upper bound of zero (i.e., ts_3) is matched with that of relative frequency (i.e., $f^U_{j \to i}(t)$) to achieve an upper bound of the converted timestamp. With both bounded frequencies and zeros, strong conversion bounds can be provided for all timestamps taken in the system.

3.3 Evaluation of Timestamp Synchronization

In this section we evaluate the accuracy and cost of our timestamp synchronization method and compare it with the accuracy and costs of existing methods. The experiments were carried out on a heterogeneous 8-node cluster with a total of 76 cores operating at 800-2,261 MHz and 8-32 GB main memory per node. Each node runs 64-bit Ubuntu 14.04 kernel.

Maximum Error - Relative Frequency and Zeros. In our first experiment, we sought to get an idea for how the error in the relative frequency is affected by the relative location of the clocks in the cluster. To do so, we calculated the relative frequencies of each of the clocks in our cluster. Measurements were performed for each pair of cores (there are 76 cores spread across 8 machines). We measured the frequencies over a duration of 24 hours in order to minimize the effects of variance in communication costs. Figure 2 shows the maximum error in terms of the difference between the bounds for our relative frequency measurements. Since the actual relative frequency can be anywhere within the computed bounds, we measure the worst case error as the size of the range defined by those bounds. As we can see, the maximum error is clustered in two regions: (across machines) the causal dependencies incur a roundtrip network communication that increases the error; and on (same machine) the errors for different cores are lower as there is no network overhead. While frequency error for cores across different sockets have higher error in some samples compared to error for those within the socket, the inter-socket communication does not impact the relative frequency measurements as much.

We validated our claim that length of time required to achieve a desired accuracy grows linearly with desired accuracy by measuring the maximum error of a relative frequency as difference of its bounds, i.e., $error_{j \to i} = f^+_{j \to i}(t) - f^-_{j \to i}(t)$ for varying t. Figure 3 shows that the error drops rapidly and after 12 hours the accuracy increase is extremely small.

Our next experiment measures the distribution of the error sizes among the zero values calculated over the clocks in our cluster. We calculated the zero between each pair of clocks in our system, and took the maximum error as the difference between the upper and lower bounds in the measurement. Figure 4 summarizes the results of this experiment. The plurality of the zero values had an error around 0.7 million cycles, and none of the measurements had zero values greater than 1.6 million cycles. For the gigahertz clocks present on the machines in our system, the error in the zero measurement is in the millisecond range.

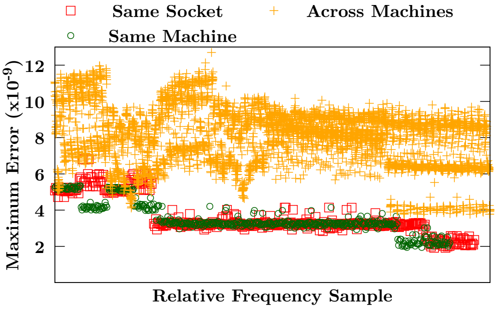

Fig. 2. Distribution of maximum error in relative frequency measurements.

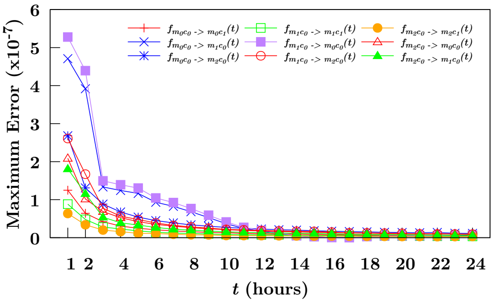

Fig. 3. Increase in relative frequency accuracy with t. Subscript m_ic_j indicates core j on machine i.

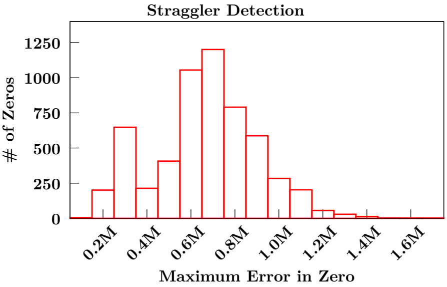

Fig. 4. Distribution of maximum errors in zero measures.

Inversions due to error. If the error in frequency and zeros is larger than the difference between two causally ordered events in the system, then an *inversion* will occur. Our next experiment investigates how far apart these causally related events needed to be before we stopped witnessing inversions. We used the algorithm from Algorithm 5 to generate pairs of timestamps which are causally ordered, then we converted the first timestamp to the base of the second timestamp and compared them to see if there was an inversion. By inserting a sleep between the taking of the first timestamp and the waiting at the barrier, we can separate the timestamps by an arbitrary amount of time. To mitigate the effects of variance in the sleep function we averaged each of our measurements across 1000 trials. Figure 5 shows that the percentage of inversions drops drastically at around the 50 microsecond delay mark and disappeared completely when the delay exceeded 140 microseconds. Thus, if the events in a program are separated by more than about 140 microseconds in real time, we expect the ordering to be correctly inferred.

Algorithm 5: Algorithm for timestamp inversions

```
Process 1                                  | Process 2
$ts_1 \leftarrow$ timestamp()              | barrier()
sleep($d$)                                 | $ts_2 \leftarrow$ timestamp()
barrier()                                  |
```

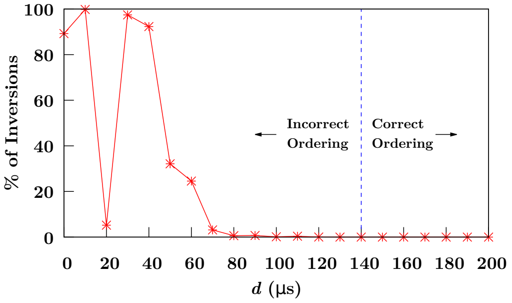

Fig. 5. Timestamp inversions in Algorithm 5 indicating potential $rt(ts_2) \not> rt(ts_1)$ as d increases.

Cost of capturing events. The timing information is captured using the x86 timestamp counters whose relative frequency and zero bounds are captured by FreeZer in the previous step. Querying these counters is a relatively inexpensive operation since it does not need any synchronization across different cores and can be executed in user space. This helps to keep the overall profiling lightweight and cause minimal perturbation to the program. We measured the amount of time that it took to generate and save one hundred thousand to one million events. The results in Figure 6 show that capturing 1M events requires only 197 ms while consuming less than 51MB of main memory.

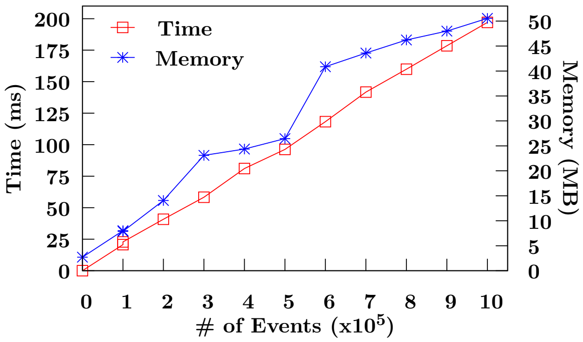

Fig. 6. Time & memory consumption to capture events.

3.4 Impact of Communication Latency

Since FreeZer estimates relative frequencies and zeros using `send_to` and `receive_from` network operations, these measures are affected by communication latency. Specifically, the communication hardware impacts errors in the measures; a faster network (e.g., InfiniBand over traditional TCP/IP/Ethernet) would cause lesser errors compared to a slower network since the collected timestamps capture the network latency as well. These errors get reflected as bounds in FreeZer, and hence, a faster network produces tighter bounds for our relative frequencies and zeros measures. While FreeZer does not (and cannot) calculate the exact relative frequencies and zeros, it also does not estimate (or *approximate*) communication latencies to provide strongly bounded results. As discussed in §4, our profiler performs send-receive operations using UDP over unix sockets instead of using a higher level network programming layer like MPI to eliminate unpredictable delays that impact measurement accuracy.

3.5 Comparison with Other Algorithms

Accuracy comparison. Next we compare FreeZer with two other timestamp synchronization methods, *linear regression* and *convex hull* algorithms from [Duda et al. 1987]. The linear regression algorithm forms a set of points from pairs of timestamps taken from message transmission events during the execution. A linear regression is performed on this set, and the resulting trend line is used as the conversion function estimate. In the convex hull algorithm, the same set is formed as above, and then further split into two subsets based on the message transmission direction. The conversion function is estimated as the line passing through the middle of the corridor formed by the convex hulls of these subsets.

Since most algorithms do not provide explicit error bounds, we must take care to define what is meant by accuracy. We use a popular measure of accuracy for timestamp synchronization algorithms called *fast sends.* A *fast send* is a send which is closer to its corresponding receive than the minimum network transmission time. To measure the number of fast sends we used two trace files of timestamps generated by running a program on a pair of machines that repeatedly exchanges messages in both directions. Each message exchange consists of a send and a receive and causes two events, one before the send and another after the receive, to be generated. Since the intervals reported by the bounded algorithms are not directly comparable to the point estimates of the unbounded algorithms, we use the interval midpoints in the following comparison. The results gathered over 100 trials, presented in Table 2, show that FreeZer performs favorably relative to the other two algorithms.

Efficiency comparison. The primary factor affecting the *synchronization time* is the number of events in the traces. Therefore in Figure 7 we compare the synchronization times of different algorithms with increasing trace size. Traces were generated in the same way as for accuracy measurements.

First let us consider the efficiency of *linear regression* and *convex hull* algorithms from [Duda et al. 1987] who efficiency is nearly the same. FreeZer is only slightly slower than the linear regression algorithm – at 10k messages 0.39× and at 1M messages 0.35×. The small performance gap between FreeZer and these algorithms is due to FreeZer's use of arbitrary precision integer arithmetic to ensure that floating point inaccuracy and finite integer precision do not affect the validity of the bounds. While the two algorithms are slightly more efficient than FreeZer, they do not provide error bounds.

The only previous algorithm that reports error bounds is the *accuracy reporting convex hull* (ar-convex hull) algorithm of Poirier et al. [Poirier et al. 2010]. This works similarly to the regular convex hull algorithm above, but it derives bounds on the conversion error for individual timestamps

Table 2. Accuracy comparison in terms of % *fast sends*.

Algorithm	Min	Average	Max
FreeZer	0.000	0.000	0.000
Convex hull [Duda et al. 1987]	36.855	40.647	49.490
Linear regression [Duda et al. 1987]	0.075	3.142	30.790

by considering the set of all possible lines in the corridor of the convex hulls. We observe that the ar-convex hull algorithm is the slowest. For instance, with 10k messages exchanged in each direction, the speedup of FreeZer over ar-convex hull is 57.25×. The expense of the ar-convex hull algorithm stems from its formulation. Every bound for each timestamp is found by solving a linear programming problem whose size grows with the number of timestamps being converted. However, FreeZer finds each bound for each timestamp by simply using the appropriately bounded relative frequency and zero. Since the calculation of the relative frequency and zero is independent of the timestamps being converted, it can be done offline.

There are two additional algorithms. The *two-point algorithm* [Ashton 1995] first forms the same set of points as the linear regression algorithm. It then uses this set to construct a set of "equivalences", which are estimates of points that lie on the true conversion line. The conversion function estimate is taken as the line connected the two "best equivalences" as defined by a set of rules intended to minimize error. The *controlled logical clock* [Rabenseifner 1997] first runs one of the above algorithms, and then retroactively adjust timestamps which violate pre-determined causality rules. We chose not to further evaluate these two algorithms for the following reasons. The controlled logical clock algorithm is a wrapper algorithm which first requires a pre-synchronization step; it is not a standalone algorithm. The two-point algorithm requires a minimum number of events to operate, and this threshold is not met by the traces encountered in our later experiments with real applications.

Thus our results show that FreeZer **not only exhibits high efficiency, it also provides error bounds.** Finally, we would like to point out that recently an accurate and efficient approach for ‘clock synchronization’ was proposed called Huygens [Geng et al. 2018]. Huygens synchronizes

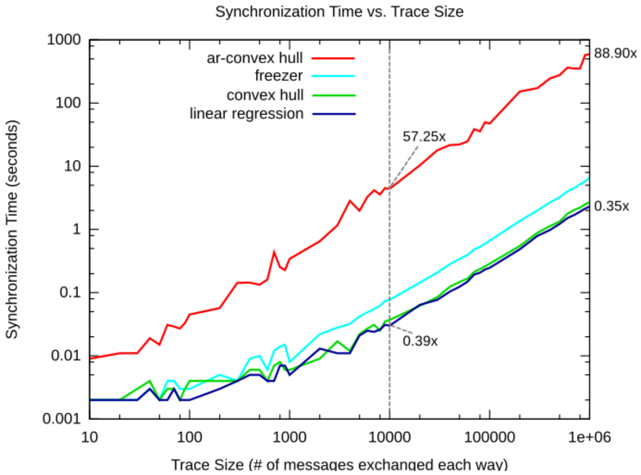

Fig. 7. Comparison of the synchronization times for different algorithms – note that both axes are on a log-scale; Annotations show speedup relative to FreeZer.

clocks (e.g. the PTP Hardware Clocks) in the NICs attached to servers, and is aimed to improve accuracy of timestamps for data, protocol messages and other transactions between different servers. Since it solves a similar problem but at a lower-level than timestamp synchronization, it can be incorporated along-side Freezer to further improve the overall accuracy.

4 PUTTING IT ALL TOGETHER: DPROF AND ITS USES

Figure 8 shows how DProf uses the bounded frequency and zero estimates generated by FreeZer to perform distributed profiling. DProf performs the following steps: first, FreeZer collects bounded frequency and zero estimates based on techniques just discussed. Then, DProf profiles the distributed execution using local x86 timestamp counters to generate process local profiles. These local profiles are then consolidated using the bounded frequency and zero estimates to generate a unified profile with global event ordering and timing information. The consolidated profile is then queried and analyzed based on the kind of profiling being performed by the end user. Next, we discuss each of the steps in detail.

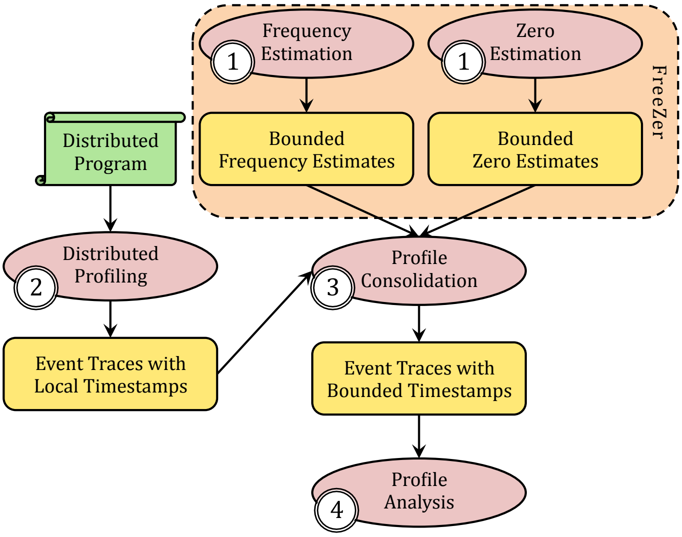

Fig. 8. DProf workflow.

4.1 Step 1: Computing Relative Frequency & Zero Bounds

FreeZer computes relative frequency and zero bounds between pairs of clocks. To compare all the timestamps in the system a base core is chosen, and all the relative frequencies and zeros involving that base core are computed. Then during the profile consolidation phase, the timestamps are all converted to the units of the base core. While causal dependences across cores may require network operations, it is important to ensure that they don't introduce unpredictable delays that impact the overall accuracy. For this, DProf performs send-receive operations using UDP over unix sockets instead of using a higher level network programming layer like MPI.

Computing at Scale. As the number of cores in the cluster increases, computing relative frequency and zero bounds relative to a single base becomes time consuming. Hence, DProf parallelizes this computation by selecting multiple base nodes relative to which our estimates get computed. Since relative frequency calculation primarily includes waiting times, parallelization is trivially achieved by having background processes that are pinned to each core and are responsible for maintaining relative frequency measures against the base core.

Parallelizing zero calculations, on the other hand, requires careful co-ordination since they do not rely on waiting times. For this case, a tree-based parallelization strategy enables computing multiple pairs of zeros in parallel. The key idea is to maintain a logical tree where nodes correspond to the cores in the cluster, and zero calculations occur between nodes to their immediate parents. By doing so, the zero calculations for nodes at the same level can be performed concurrently. Timestamp synchronization (*event consolidation* in step 3) is achieved by converting timestamps multiple times (based on transitive relation) until they reach the root of the tree. The branching factor of the tree (and hence, the height of the tree) impacts the tradeoff between performance via parallelization and synchronization accuracy (each conversion introduces some amount of error). With b being the branching factor, $b \times log_b(n)$ zero calculations are performed serially.

4.2 Step 2: Capturing Distributed Event Profiles

DProf profiles the distributed execution by capturing events and timing information about *marked code regions* that are of interest for profiling (we support annotations introduced in CSP [Benavides et al. 2017]). DProf provides an API to mark entry and exit points for code regions that need to be profiled. During execution, these entry and exit points translate into events whose timing information must be captured. As mentioned earlier, the timing information is captured using the x86 timestamp counters whose relative frequency and zero bounds are captured by FreeZer.

DProf also enables enriching profiles by dynamically capturing context sensitive information that is necessary for performing different types of analyses. For example, straggler detection (case study presented in next section) needs information about distributed barriers; hence, the entry and exit events for distributed barriers are annotated with information about the barrier, like its unique identifier and number of expected threads. Such context sensitive enrichment of profiles enables our profiler to be generalized across various profile analyses.

Note that the collected distributed profiles have event and timing information that is local to individual processes in the distributed system. Next, we discuss how they are consolidated so that profile information across processes can be collectively used to perform a global analysis.

4.3 Step 3: Consolidating Distributed Event Profiles

Consolidation of event profiles mainly includes converting the captured timestamps across all processes into a globally consistent time scale. DProf picks one of the core's clock as a reference and converts all the timestamps from remaining clocks to that in terms of this reference clock. Since FreeZer captures relative frequency and zero information in terms of bounds, the timestamps are converted into interval of timestamps, i.e., the profiling information is transformed from point domain to interval domain.

Consider a timestamp k captured with $core_j$'s clock c_j which needs to be converted in terms of the reference clock c_i of $core_i$. Assuming the zero for $core_j$ with respect to $core_i$ to be $zero_{j \to i} = <(z_j^L, z_j^U), z_i>$, k is converted to $<k^L, k^U>$ as:

$$k^L = (ts_1 - z_i) \times f_{j \to i}^L(t) + z_j^L$$
$$k^U = (ts_1 - z_i) \times f_{j \to i}^U(t) + z_j^U$$

Converting all the timestamps in this manner with respect to the same reference core makes them globally consistent and enables distributed profile analysis.

4.4 Step 4: Analyzing Profiles

The consolidated profiles represent directly comparable event and timing information along with context sensitive information that is dynamically captured during runtime. Hence, different kinds of analyses can be performed over these profiles to gain useful insights about the program, ranging from simply determining how much time was spent in different regions of the program, to understanding complicated relationships like expected impact of speeding up a code region on the application's performance.

4.5 Discussion

Strong Guarantees. Since the timing information in consolidated profiles are in terms of bounded intervals, the profiling strategy can leverage these intervals to provide strongly bounded results. This requires carefully maintaining the interval bounds throughout the analysis by designing interval-aware strategies like interval arithmetic and translation of time intervals into domain-specific intervals based on the kind of profiling that is being performed. For example, in next section we will carefully transform the time intervals to strongly bounded straggler scores and speedup ranges to provide domain-specific guaranteed results.

Eliminating Inverted Orderings. With bounded time intervals, ordering of events occurring nearly at the same time can sometimes become difficult if the intervals are overlapping. In such cases, causality ordering across such events can be used to enforce event ordering in the consolidated profiles and hence, eliminate inverted orderings. For instance, a synchronous send-receive operation with end of send occurring at $<t_{sd}^L, t_{sd}^U>$ and end of receive occurring at $<t_{rv}^L, t_{rv}^U>$ has the causality ordering $t_{sd}^L < t_{rv}^L$ and $t_{sd}^U < t_{rv}^U$. This causality ordering can be explicitly incorporated in the profile to eliminate inversion between the end of send and the end of receive events. However, it is important to note that such enhancement of consolidated profiles must be done without adjusting the timing information in bounded intervals in order to maintain strong guarantees.

Extensibility. Different analyses can be performed by viewing the overall profiles in different ways. In general, since DProf essentially measures the time that the system spends in a user defined global state, it could be used to solve any problem reducible to that. For example, DProf can be used to expose concurrency bugs like deadlocks and livelocks. Furthermore, it can also be used to understand inefficiencies (e.g., retries in leader election) in complex asynchronous systems like Spark [Zaharia et al. 2010], ASPIRE [Vora et al. 2014, 2017b], and GraphBolt [Mariappan and Vora 2019]. In the next section, we demonstrate how DProf is used to develop distributed versions of two recent shared memory performance analysis techniques: *Context Sensitive Profiling* (dCSP) and *Causal Profiling* (dCOZ). These can be effectively used in tandem for performance debugging – dCSP identifies performance bottlenecks and dCOZ estimates how much application performance can be expected to improve if the identified performance bottleneck is removed.

5 DISTRIBUTED ANALYSES USING DPROF

In this section, we develop distributed versions of two recent shared memory performance analysis techniques: *Context Sensitive Profiling* (dCSP) and *Causal Profiling* (dCOZ).

5.1 Context Sensitive Profiling: dCSP

Context Sensitive Profiling (CSP) [Benavides et al. 2017] allows analysis of execution times for different code regions of interest in shared memory parallel programs. The overall execution time is divided into a sequence of time intervals called *frames*, during which no process transitions between code regions. The sequence of frames is then queried to expose different performance insights like bottlenecks at synchronization points, workload imbalance, and many others. Using DProf, we develop CSP to analyze distributed programs by representing our distributed profiles as sequence of frames.

Aggregated Event View. Constructing the frame sequence using DProf profiles requires an ordering across events. Since inter-process events that occur nearly at the same time can have overlapping time intervals, ordering events results in a simple DAG structure that captures multiple possible execution orders. Typical context sensitive analysis like straggler detection requires computation for each different execution path; however, enumerating all possible execution paths can become infeasible due to path explosion.

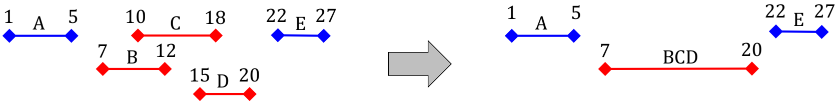

Fig. 9. Aggregating overlapping events B, C and D into a single aggregated event BCD.

To address the above issue, we develop an *Aggregated Event View* that merges overlapping events into combined events so that the resulting sequence of events have total ordering. Availability of such a total ordering enables computation of straggler scores with a linear pass over the entire distributed profile. Since the aggregated event represents multiple underlying events, its timing interval is computed as the largest interval range spanning across all the underlying events' intervals. The interval lower (upper) bound of the aggregated event is the minimum (maximum) of its underlying events. As illustrated in Figure 9, the overlapping events B, C and D are merged together to a single aggregated event BCD that spans from 7 to 20 so that it fully incorporates the three intervals.

5.2 Causal Profiling: dCOZ

Causal profiling [Curtsinger and Berger 2015; Yoga and Nagarakatte 2017] is a technique similar to logical zeroing [Hollingsworth and Miller 1994] that determines the potential impact of optimizing a selected code region on the overall performance of the program. Such kind of profiling enables users to understand different bottlenecks in concurrent programs and to prioritize various optimizations to be incorporated in those programs. While causal profiling systems like COZ [Curtsinger and Berger 2015] enable causal analysis over parallel programs, we develop causal profiling for distributed applications using DProf. Profiles introduced in [Yoga and Nagarakatte 2017] for task parallel programs can be similarly extended using DProf.

Time Weighted DAG View. To perform causal analysis on the captured events, we need to carefully propagate the impact of optimizing a code region to the remaining execution events. Such propagation ensures that the causal dependencies remain consistent while the timing information for events change.

We design a Time Weighted DAG to represent the event and timing information captured by profiling. Vertices in the graph represent captured events while edges connect events that are sequentially executed by the same thread, and inter-thread events that are directly related by causality. The edges are weighted with timestamp intervals that represent the difference between intervals of events represented by source and destination vertices. For COZ analysis, we categorize the edges into two types: *elastic* and *inelastic* edges. An *elastic* edge is one whose weights can be adjusted during COZ analysis, i.e., the duration between their source and destination events

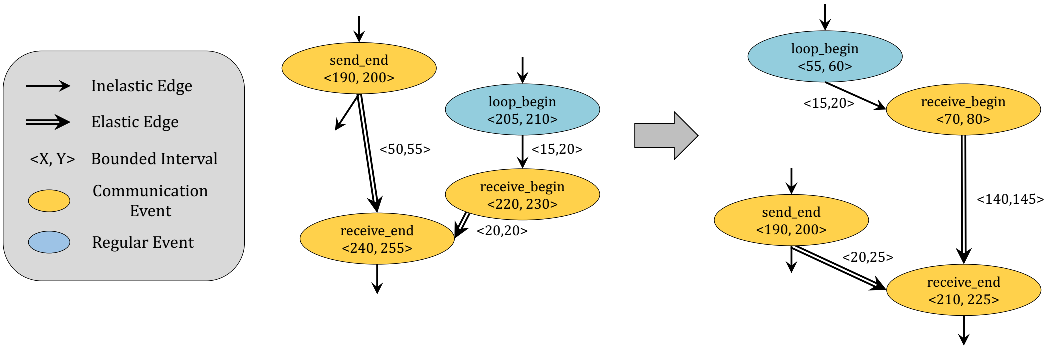

Fig. 10. Adjusting the time weighted DAG on left to the one on the right causes change in weights of elastic edges.

is allowed to grow or shrink during impact propagation. We set all the communication edges to be elastic so that it enables us to carefully adjust the timestamps of communication events while simultaneously ensuring that causal dependencies are never violated. Inelastic edges, on the other hand, are those whose weights remain same throughout the analysis. We set all the non-communication based edges as inelastic. Figure 10 shows two time weighted DAGs with elastic and inelastic edges. We can transform the left time weighted DAG to the one on the right by adjusting the timing information; note that the inelastic edge weights remain the same while elastic edges grow (e.g., between `receive_begin` and `receive_end`) and shrink (e.g., between `send_end` and `receive_end`) based on the adjustments.

Algorithm 6 shows the overall algorithm to perform causal analysis using time weighted DAG. Given a code region r and the amount of reduction, lines 5-18 process the events belonging to r by directly reducing their execution times. Events that occur after r are collected in `global_list` to be processed in lines 20-29. While processing each event in the `global_list`, the amount of reduction to be performed can be limited by causal dependencies that need to be maintained during reduction. For example, while adjusting an end of receive event, the analysis should ensure that the timestamps do not precede end of send events to avoid inversions (as shown in Figure 10). Similarly, any exit event from a given barrier should not precede the latest entry event to the same barrier. Such dependencies are taken care by `adjust()` (line 23) which analyzes the causal dependencies of the given event and its predecessor events, its elastic edges and its inelastic edges to determine the amount of reduction that can be safely performed. This makes the reduction value dynamic, which gets attached to the future events to be adjusted (lines 26-27). The work-list based reduction algorithm terminates when there is no remaining reduction to be performed on any event.

6 PERFORMANCE DEBUGGING WITH dCSP & dCOZ

We present how dCOZ and dCSP can be used for performance debugging in distributed programs. In particular, we show how dCSP can be first used to identify a performance bottleneck, and then how dCOZ can be used to estimate the expected performance improvement resulting from removal of the identified bottleneck. We also show the importance of error bounds by showing that the inferences drawn by dCSP and dCOZ deteriorate significantly if instead of using FreeZer we use either the *linear regression* or *convex hull* algorithms.

To showcase the strength of FreeZer, we choose three popular distributed programs which have varying degrees of performance bottlenecks: Connected Components [Vora et al. 2017a]

Algorithm 6: Causal Analysis using Time Weighted DAG.

```
$r$: code region whose impact is being analyzed
$reduction$: percentage of time reduction for $r$
$region\_list$: $\{v | source(edge(v))$ is an entry event for $r\}$
$global\_list$: $\varnothing$
while $region\_list \neq \varnothing$ do
    $v \leftarrow first(region\_list)$
    $region\_list \leftarrow region\_list \setminus \{v\}$
    if $v$ is not processed then
        $cut \leftarrow weight(v) \times (reduction/100)$
        $weight(v) \leftarrow weight(v) - cut$
        if $v$ is an exit event for $r$ then
            $pair \leftarrow <dest(edge(v)), cut>$
            $global\_list \leftarrow global\_list \cup \{pair\}$
        else
            $region\_list \leftarrow region\_list \cup \{dest(edge(v))\}$
        end if
    end if
end while

while $global\_list \neq \varnothing$ do
    $<v, cut> \leftarrow first(global\_list)$
    $global\_list \leftarrow global\_list \setminus \{<v, cut>\}$
    $new\_cut \leftarrow adjust(v)$
    if $new\_cut > 0$ then
        $weight(v) \leftarrow weight(v) - new\_cut$
        $pair \leftarrow <dest(edge(v)), new\_cut>$
        $global\_list \leftarrow global\_list \cup \{pair\}$
    end if
end while

$new\_time \leftarrow time(program\_exit\_event)$
$speedup \leftarrow (original\_time - new\_time)/original\_time \times 100$
```

(CC), PageRank [Page et al. 1999], and K-Means [Liao [n.d.]]. All three programs iterate over data-elements (graph vertices/edges for CC and PageRank, points for K-Means) to compute results; as an example, the overall structure of K-Means is shown in Algorithm 7. We use [Stisen et al. 2015] for K-Means and LiveJournal [Leskovec et al. 2008] for PageRank and CC to generate important events for performance analysis. These three distributed programs reflect modern real-world distributed programs that also extract multicore performance via threading on each machine; in each iteration, all machines execute their share of the workload in parallel using multiple threads and then they synchronize at a barrier, exchange information, and move to the next iteration.

We consider *stragglers* as performance bottlenecks in our distributed programs. Straggler processes arise when all processes but one finish their assigned task and are left waiting at the barrier while the single remaining process slowly finishes its task. Stragglers are a common reason for slowdown in distributed programs since they can be caused due to several reasons including load imbalance, hardware performance imbalance, network delays, etc. We used 5 machines to perform

Algorithm 7: Distributed K-Means. Region 0 (blue) is the body region, and region 1 (red) is the barrier region for the dCSP and dCOZ analyses.

```
objects ← read_objects( )
Send_Receive(objects)
clusters ← init_clusters(random_centroids)
All_Reduce(clusters)
do
    #Region(0)
    for o ∈ objects do
        c ← compute_closest_cluster(o, clusters)
        o.cluster ← c
        c.objects ← c.objects ∪ {o}
    end for
    # Region(0)
    #Region(1)
    All_Reduce(clusters)
    # Region(1)
    new_centroids ← compute_new_centroids(clusters)
    All_Reduce(new_centroids)
    clusters ← init_clusters(new_centroids)
    change ← compute_change(objects)
    All_Reduce(change)
while change > threshold
```

this case study and the presence of a straggler process was ensured via workload imbalance, i.e., one of the processes was assigned significantly more work compared to other processes. On each machine the workload was executed in parallel by 8 threads. The impact of stragglers is highest in CC and lowest in K-Means which allows us to perform sensitivity analysis of Freezer's strong guarantees.

Algorithm 7 shows the code region annotations used to generate events for detecting stragglers. The blue region is where primary work is performed, and the red region is the immediately following barrier synchronization point. These regions allow us to detect stragglers by searching for dCSP frames in which all but one process/thread are in the red region, while there is exactly one thread/process in the blue region. For concrete analysis, we define *straggler score* (SS) of a thread t to be the percentage of time that the thread was in the blue region while all other threads were in the red region.

We perform our analysis in three steps. First, we use dCSP to calculate the straggler scores for each of the threads in the system. Second, we use the straggler scores calculated by dCSP as input to dCOZ, and calculate the expected speedup when the straggler identified by dCSP is sped up by the value of its straggler score. And finally, we eliminate the straggler by removing workload imbalance and measure the *actual* execution time to compute the overall *prediction error* under each of the three timestamp synchronization schemes: FreeZer, Convex Hull, and Linear Regression.

The relative frequencies measured across our five 8-core machines are between 1.23529−1.40003 with average being 1.34956 and standard deviation being 0.0668561 (detailed numbers are provided in Appendix §A.1). The zeros measured are are TSC numbers which are 17 digits (base 10) with the error being 5-6 digits.

Table 3. Straggler scores for the straggler process. Higher percentage means more severe straggling. The execution times for the programs were: CC – 1799.173 seconds; PageRank – 844.251 seconds; and K-Means – 13.7 seconds.

	CC	PageRank	K-Means
FreeZer	47.19 - 47.21 %	37.17 - 37.19 %	18.9 - 19.9 %
Linear Reg.	23.46 %	18.48 %	8.7 %
Convex Hull	11.75 %	9.73 %	6.9 %

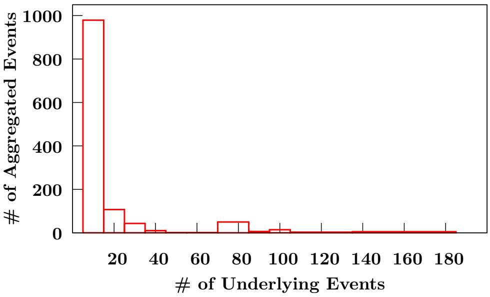

Fig. 11. Size of aggregated events.

Step 1. dCSP: Straggler Detection. With one process chosen to be the straggler, we used dCSP to measure the straggler score of the chosen process. The results of this step are shown in Table 3. Since Freezer provides bounded timestamp information, we calculate bounds for straggler scores by propagating the timestamp bounds throughout the straggler analysis. As we can see, FreeZer's bounds for straggler scores are tight, i.e., at most one percent, which is due to little impact of communication latency on FreeZer's measurements (see §3.4). This shows that FreeZer not only provides tight bounds in theory, but also provides the same in practice. It is interesting to see that linear regression (Linear Reg.) also provides positive straggler scores, however they are not even close to FreeZer's bounds. Furthermore, convex hull (Convex Hull) provides even smaller straggler scores, incorrectly diminishing the imbalance issue among threads.

It is interesting to note that the coarse grained view provided by aggregated events does not impact our profiling results such that the straggler threads go undetected. This is because the aggregated view mostly summarizes small number of events, leaving out majority of events from the original profile. Figure 11 shows the distribution of aggregated event sizes in terms of number of underlying events. As we can see, the size distribution drops sharply; with 1,219 aggregated events, over 89% have at most 20 underlying events and less than 1% have over 100 underlying events.

Step 2. dCOZ: Causal Analysis. Next we use dCOZ to answer the question: if the straggler had finished on time, what reduction in execution time would be observed? Our dCOZ profiler takes two inputs: the code region that will be sped up to eliminate delay caused by straggler, and the amount of speedup to apply to that region. It estimates the expected overall speedup of the program. We again examine the primary work region (blue region in Algorithm 7), except that instead of using a static region identifier of 0, we use a dynamic region identifier corresponding to the rank of the process in the MPI global communicator. This allows us to select the right edge in time weighted DAG view whose loop body corresponds to the process that we've elected as the straggler.

Table 4. Predicted overall speedups via causal profiling given a speedup of the straggler process equal to its straggler score as computed in Step 1. LR w/ Freezer SS and CH w/ Freezer SS indicate Linear Regression and Convex Hull synchronization when using straggler scores calculated by FreeZer.

	CC	**PageRank**	**K-Means**
FreeZer	44.37-45.30 %	32.94-33.72 %	18.1-18.4 %
Linear Regression	21.60 %	15.53 %	7.7 %
LR w/ FreeZer SS	23.46 %	18.48 %	8.1 %
Convex Hull	8.44 %	6.79 %	5.3 %
CH w/ FreeZer SS	11.75 %	9.72 %	7.6 %

Table 5. Percentage error in prediction.

	CC	**PageRank**	**K-Means**
FreeZer	4.28-5.88 %	4.46-5.57 %	1.8-1.8 %
Linear Regression	34.9 %	20.35 %	10.5 %
LR w/ FreeZer SS	31.7 %	16.14 %	10.5 %
Convex Hull	57.54 %	32.8 %	14.0 %
CH w/ FreeZer SS	51.85 %	28.63 %	11.4 %

The results of dCOZ are shown in Table 4. When we calculated the causal profiles using the *linear regression* and *convex hull* synchronizations, we did so using not only the straggler scores corresponding to those methods, but also using the straggler score as calculated with the FreeZer synchronization; this eliminates the errors coming from step 1 and clearly shows errors induced by *linear regression* and *convex hull* in dCOZ alone. As we can again see, FreeZer provides strongly bounded results; in fact, even with smaller execution times for K-Means, FreeZer's bounds are tight. While *Linear Regression* and *Convex Hull* indicate some performance improvement upon removal of stragglers, are again far from FreeZer's results. Even when using FreeZer's straggler scores, *Linear Regression* and *Convex Hull* provide low numbers, again incorrectly estimating that little performance improvement can be achieved.

Step 3. Prediction Error. Next we show the strength of FreeZer over linear regression and convex hull methods by comparing our dCOZ predictions (from Table 4) with execution times when the straggler is eliminated, i.e., using balanced workload across all threads. The execution times with and without straggler are: 1799.2s and 1045.6s for CC, 844.3 and 592.6s for PageRank, and 13.7 and 11.4s for K-Means, and the error in our dCOZ prediction is shown in Table 5. The error values for FreeZer is an order of magnitude lower compared to convex hull and linear regression methods; this is because FreeZer captures and retains tight bounds throughout the analysis instead of approximating the estimates, as done by other techniques. Even when using FreeZer's straggler scores for linear regression and convex hull, their error values remain high which indicates the low accuracy of these methods in dCOZ alone.

Note that while FreeZer provides strong bounds, our dCOZ analysis only captures the impact of eliminating stragglers in terms of speedup on the primary work region. This means, speedups occurring due to other factors upon straggler elimination like caching effects, improved graph locality, etc. are not captured by dCOZ. Since these effects also impact execution time, FreeZer's error in Table 5 cannot be zero.

It is interesting to note that Linear Regression and Convex Hull underestimate straggler scores in all cases; this is because our straggler score is defined based on *all but one* thread being inside the barrier and synchronization inaccuracies offsetting relative timestamps end up causing *multiple* threads to be observed as being outside the barrier.

7 CONCLUSION

We developed DProf, a profiler for distributed programs that provides results with strong guarantees. DProf relies on bounded frequency and zero measures that tightly capture the inaccuracies in the timing information. Using DProf, we developed two performance analysis techniques, dCSP and dCOZ, to demonstrate its versatility and effectiveness in developing tools for distributed performance debugging.

A APPENDIX

A.1 Relative Frequencies for Performance Debugging

The relative frequencies measured across the five 8-core machines used for performance debugging in §6 are between 1.23529–1.40003 with average being 1.34956 and standard deviation being 0.0668561. Table 6 and Table 7 shows the frequencies of all 40 cores (5 machines × 8 cores per machine) relative to a single *host* machine (*host* machine is not part of the 5 machines on which profiles get collected).

Table 6. Frequencies of all 40 cores (5 machines × 8 cores per machine) used for performance debugging in §6 relative to a single *host* machine.

Core	Machine 0	Machine 1
0	1.39998789583816 - 1.399988536446	1.23528657157467 - 1.23528728781227
1	1.39998787495768 - 1.39998857117999	1.23528656912685 - 1.23528724599093
2	1.39998779840999 - 1.39998856764179	1.23528657795941 - 1.23528723807361
3	1.39998409672478 - 1.39998857352618	1.23528658135189 - 1.235287255391
4	1.39998782404304 - 1.3999885785345	1.23528664181965 - 1.23528719264894
5	1.39998789441459 - 1.39998856599002	1.23528658829639 - 1.23528725879004
6	1.39998783306934 - 1.39998857641062	1.23528656905431 - 1.23528723269777
7	1.39998785968398 - 1.39998856312875	1.23528654603035 - 1.23528724815829

Core	Machine 2	Machine 3
0	1.31249258885137 - 1.31249307635736	1.40002774414526 - 1.40002811043533
1	1.31249264001219 - 1.31249305393534	1.40002766683076 - 1.40002814637304
2	1.31249260010262 - 1.31249303995655	1.40002775210883 - 1.40002807406718
3	1.31249260911186 - 1.31249312453983	1.40002774391212 - 1.40002808264091
4	1.31249258708874 - 1.31249308555409	1.40002774091796 - 1.40002806462242
5	1.31249257261324 - 1.31249308696561	1.40002772078684 - 1.40002807291304
6	1.31249255172596 - 1.31249308269315	1.40002772219227 - 1.40002807190336
7	1.31249263731281 - 1.31249300333314	1.40002776298356 - 1.40002809653397

Table 7. Continuation of Table 6: Frequencies of all 40 cores (5 machines × 8 cores per machine) used for performance debugging in §6 relative to a single *host* machine.

Core	Machine 4
0	1.40002789036014 - 1.40002831747008
1	1.4000278943691 - 1.40002822955989
2	1.40002789817165 - 1.40002823701277
3	1.40002790022046 - 1.40002823036416
4	1.400027916588 - 1.40002829188384
5	1.40002792830673 - 1.40002828839855
6	1.40002788726602 - 1.40002822905083
7	1.4000279166127 - 1.40002825884718

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1617424 and Grant No. CCF-1524852. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

L. Adhianto, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. Tallent. 2008. Hpctoolkit: Performance measurement and analysis for supercomputers with node-level parallelism. In *Workshop on Node Level Parallelism for Large Scale Supercomputers, in conjuction with ACM/IEEE SC*.

T.E. Anderson and E.D. Lazowska. 1990. Quartz: A tool for tuning parallel program performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 115–125.

P. Ashton. 1995. Algorithms for off-line clock synchronization. In *Technical Report TR-COSC12/95, Dept. of Computer Science, Univ. of Canterbury*.

D. Becker. 2010. Timestamp Synchronization of Concurrent Events. In *Schriften des Forschungszentrums Julich, IAS Series, Volume 4*.

Z. Benavides, R. Gupta, and X. Zhang. 2017. Annotation guided collection of context-sensitive parallel execution profiles. In *The 17th International Conference on Runtime Verification, LNCS 10548, Springer*. 103–120.

D. Böhme, F. Wolf, B.R. de Supinski, M. Schulz, and M. Geimer. 2012. Scalable critical-path based performance analysis. In *IEEE 26th International Parallel & Distributed Processing Symposium*. 1330–1340.

K. Du Bois, J.B. Sartor, S. Eyerman, and L. Eeckhout. 2013. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. 355–372.

C. Curtsinger and E.D. Berger. 2015. Coz: finding code that counts with causal profiling. In *The 25th Symposium on Operating Systems Principles*. 184–197.

R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *USENIX Annual Technical Conference*. 139–150.

A. Duda, G. Harrus, Y. Haddad, and G. Bernard. 1987. Estimating global time in distributed systems. In *ICDCS, Volume 87*. 299–306.

T.H. Dunigan. 1992. Hypercube clock synchronization. In *Concurrency and Computation: Practice and Experience, 4(3):257-268.*

M. Geimer, F. Wolf, B.J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. 2010. The scalasca performance toolset architecture. In *Concurrency and Computation: Practice and Experience, 22(6):702-719.*

Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*. 81–94.

R. Hofman and U. Hilgers. 1998. Theory and tool for estimating global time in parallel and distributed systems. In *Sixth Euromicro Workshop on Parallel and Distributed Processing*. 173–179.

J.K. Hollingsworth. 1996. An online computation of critical path profiling. In *SIGMETRICS Symposium on Parallel and Distributed Tools*. 11–20.

J.K. Hollingsworth and B.P. Miller. 1994. Slack: a new performance metric for parallel programs. In *Univ. of Maryland and Univ. of Wisconsin-Madison, Tech. Rep.*

M. Kambadur, K. Tang, and M.A. Kim. [n.d.].

J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney. 2008. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. In *CoRR, abs/0810.1355*.

W.K. Liao. [n.d.]. Parallel k-means data clustering.

X. Liu and J. Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *IEEE/ACM 9th Annual International Symposium on Code Generation and Optimization*. 171–180.

E. Maillet and C. Tron. 1995. On efficiently implementing global time for performance evaluation on multiprocessor systems. In *Journal of Parallel and Distributed Computing, 28(1):84–93*.

M. Mariappan and K. Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *European Conference on Computer Systems (EuroSys)*. ACM, Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.S. Lim, and T. Torzewski. 1990. Ips-2: the second generation of a parallel program measurement system. In *IEEE Transactions on Parallel and Distributed Systems, 1(2):206–217*.

W.E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. 1996. Vampir: Visualization and analysis of mpi resources. https://tu-dresden.de/zih/forschung/projekte/vampir?set_language=en

L. Page, S. Brin, and R. Motwani. 1999. The PageRank citation ranking: Bringing order to the web. In *Stanford InfoLab*.

B. Poirier, R. Roy, and M. Dagenais. 2010. Accurate offline synchronization of distributed traces using kernel-level events. In *ACM SIGOPS Operating Systems Review, 44(3):75–87*.

R. Rabenseifner. 1997. The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In *The 5th EUROMICRO Workshop on Parallel and Distributed Processing*. 477–484.

S.S. Shende and A.D. Malony. 2006. The tau parallel performance system. In *International Journal of High Performance Computing Applications, 20(2):287–311*.

A. Stisen, H. Blunck, S. Bhattacharya, T.S. Prentow, M.B. Kjærgaard, A. Dey, T. Sonne, and M.M. Jensen. 2015. Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In *13th ACM Conference on Embedded Networked Sensor Systems*. 127–140.

K. Vora, R. Gupta, and G. Xu. 2017a. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 237–251. https://doi.org/10.1145/3037697.3037748

K. Vora, S.-C. Koduru, and R. Gupta. 2014. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*. 861–878.

K. Vora, C. Tian, R. Gupta, and Z. Hu. 2017b. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/3037697.3037747

C.-Q. Yang and B.P. Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *8th International Conference on Distributed Computing Systems*. 366–373.

A. Yoga and S. Nagarakatte. 2017. Fast Causal Profiler for Task Parallel Programs. In *11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.