TANDEM

A Taxonomy and a Dataset of Real-World Performance Bugs

Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo and Sergio Segura

Li-SCIS17

BugID: PB1	PublicationDOI: 10.1007/s11432-015-1015-5	Year: 2017

Performance bug data

Figure	Effect	Cause	Context	Language
Fig. 1	Energy	Condition, Unnecessary computation	Communication	Java (Android)

Description

Figure 1 gives a simplified example of energy inefficiency caused by sensory data under-utilization. The app listens to GPS updates and defines a handler *onLocationChanged* to handle location changes, calculating the distance between the new and a target location. If the distance is larger than *NOT FAR DISTANCE*, the handler will do nothing. All location data that indicate locations far away from the target location will always be discarded. If an app frequently encounters such cases, the location data utilization would be very low compared to location data that are close to the target location, causing significant energy waste.

Source code

```
1  public void onLocationChanged(Location loc) {
2      double dis = calDistance(loc, goalLoc);
3      if (dis < NEAR_DISTANCE)
4          doLightWork();
5      else if (dis < NOT_FAR_DISTANCE)
6          doHeavyWork();
7      else
8          ; //do nothing
9  }
```

Li-SCIS17

BugID: PB2	PublicationDOI: 10.1007/s11432-015-1015-5	Year: 2017

Performance bug data

Figure	Effect	Cause	Context	Language
Fig. 8	Energy	Condition, Unnecessary computation	Communication	Java (Android)

Description

Omnidroid's code snippets. (a) OmniArea.java; (b) LocationManager.java. If there are rules concerning location data, Omnidroid will listen to location update events. Every time when there is an update, it will construct an *OmniArea* object with the received location data. However, the *longitude* and *latitude* fields of the object are mistakenly switched. Therefore, if *longitude* is greater than 90 or less than −90, the constructor of OmniArea will throw an exception (the range of latitude is −90 to 90) and just returns. This exception will be caught but ignored, and therefore the location data will not be utilized.

Source code

```
1  public void onLocationChanged(Location location) {
2      OmniArea newLocation;
3      try {
4          // BUG: the longitude and latitude are mistakenly switched
5          newLocation = new OmniArea(null, location.getLatitude(),
6              location.getLongitude(), location.getAccuracy());
7      } catch (DataTypeValidationException e) {
8          newLocation = null;
9      }
10     //follow-up processing
11 }
```

(a)

```
1  public OmniArea(String userInput, double longitude, double latitude,
2      double proximityDistance) throws DataTypeValidationException {
3      if (latitude < MIN_LATITUDE || latitude > MAX_LATITUDE)
4          throw new DataTypeValidationException("Latitude must be between "
5              + MIN_LATITUDE + " and " + MAX_LATITUDE);
6      if (longitude < MIN_LONGITUDE || longitude > MAX_LONGITUDE)
7          throw new DataTypeValidationException("Longitude must be between "
8              + MIN_LONGITUDE + " and " + MAX_LONGITUDE);
9      if (proximityDistance < 0)
10         throw new DataTypeValidationException("Distance cannot be negative");
11     //follow-up processing
12 }
```

(b)

Pathak-MobiSys12

BugID: PB3	PublicationDOI: 10.1145/2307636.2307661	Year: 2012

Performance bug data

Figure	Effect	Cause	Context	Language
Fig. 3	Energy	Missing operation	Communication	Android

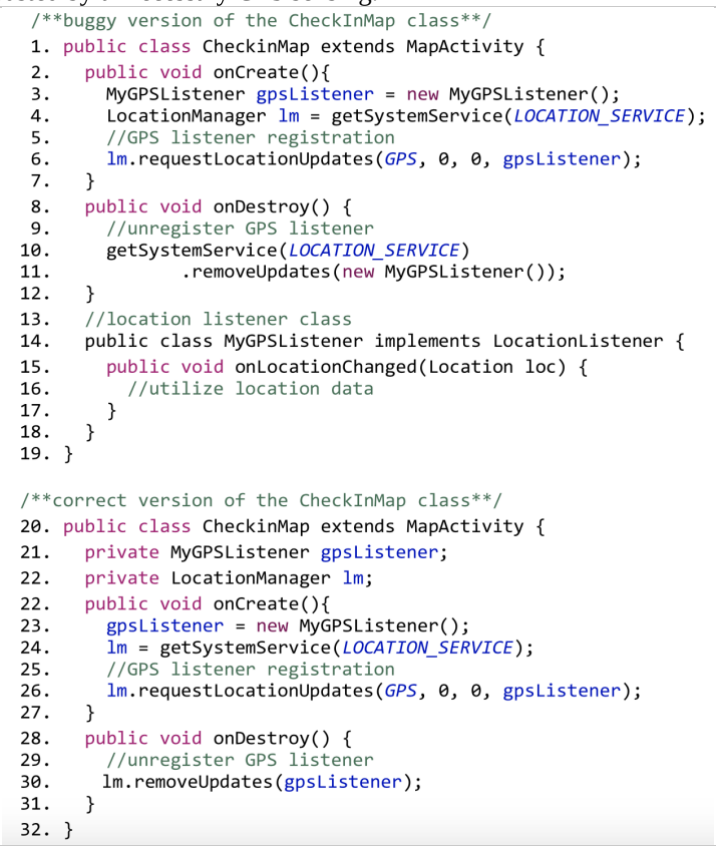
Description

This code shows an example of the complexity involved in power-encumbered programming. This is a code-snippet from the Android framework's class *CdmaConnection*. This class uses a *PARTIAL_WAKE_LOCK* for managing the connection and releases the *wakelock* when the connection is disconnected. However, there are many different possible program paths arising from different patterns of user interactions, hardware states dependent on the external environment, etc. The developer included *releaseWakeLock* in *finalize* as an additional safety measure, even though *finalize* is not guaranteed to be called.

Source code

```
1  @Override protected void finalize(){
2      /**
3       * It is understood that This finializer is not
4       * guaranteed to be called and the release lock
5       * call is here just in case there is some path
6       * that doesn't call onDisconnect and or
7       * onConnectedInOrOut.
8       */
9      if (mPartialWakeLock.isHeld()) {
10     Log.e(LOG_TAG, "[CdmaConn] UNEXPECTED; mPartialWakeLock
           is held when finalizing.");
11     }
12     releaseWakeLock();
13 }
```

Liu-TSE14

BugID: PB4	PublicationDOI: 10.1109/TSE.2014.2323982	Year: 2014		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 9	Energy	Missing operation	Communication	Android (Java)
Description	This code represents an energy problem in Ushahidi application due to developers wrongly unregistered a GPS listener. We observe in the buggy version that, developers registered a GPS listener *gpsListener* in the *onCreate()* handler of the *CheckInMap* activity (lines 3-6), and then tried to unregister the listener in the *onDestroy()* handler of *CheckInMap* (lines 10-11). However, instead of passing previous registered *gpsListener* to the sensor listener unregistration API *removeUpdate()*, developers wrongly created a new GPS listener instance and passed its reference to *removeUpdate()*. The consequence is that the previously registered sensor listener *gpsListener* was not properly unregistered. Therefore, in the buggy version, the *gpsListener* instance would remain in memory for a long time even if the activity it belongs to has been destroyed. The activity instance could also remain in memory after its *onDestroy()* handler is called. As a result, valuable battery energy can be wasted by unnecessary GPS sensing.				
Source code				

```
/**buggy version of the CheckInMap class**/
1. public class CheckinMap extends MapActivity {
2.   public void onCreate(){
3.     MyGPSListener gpsListener = new MyGPSListener();
4.     LocationManager lm = getSystemService(LOCATION_SERVICE);
5.     //GPS listener registration
6.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
7.   }
8.   public void onDestroy() {
9.     //unregister GPS listener
10.    getSystemService(LOCATION_SERVICE)
11.           .removeUpdates(new MyGPSListener());
12.  }
13.  //location listener class
14.  public class MyGPSListener implements LocationListener {
15.    public void onLocationChanged(Location loc) {
16.      //utilize location data
17.    }
18.  }
19. }

/**correct version of the CheckInMap class**/
20. public class CheckinMap extends MapActivity {
21.   private MyGPSListener gpsListener;
22.   private LocationManager lm;
22.   public void onCreate(){
23.     gpsListener = new MyGPSListener();
24.     lm = getSystemService(LOCATION_SERVICE);
25.     //GPS listener registration
26.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
27.   }
28.   public void onDestroy() {
29.     //unregister GPS listener
30.     lm.removeUpdates(gpsListener);
31.   }
32. }
```

Liu-Internetware17

BugID: PB5	PublicationDOI: 10.1007/s11432-017-9400-y	Year: 2017		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Energy	Missing operation	Communication	Android (Java)
Description	The code shows two energy inefficiency problems. Users can click the play/pause button (Lines 12–15, 48–51). The *PlaybackActivity* starts a *PlaybackService* in the background (Lines 17–19), and binds to the service for interaction (Lines 22–24). In order to prevent the media playing from being interrupted, *PlaybackService* uses a			

	wake lock to keep the CPU on during the media playing (Lines 34–35). When the media player gets prepared, the wake lock will be acquired (Lines 43–46). The service checks if the wake lock is held before releasing it (Lines 61, 66), but acquires the wake lock directly without checking (Line 55), so the wake lock can be acquired more than once. If a user clicks the play/pause button twice after the media player gets prepared, the wake lock will be acquired twice but released only once and the battery power will continue to be consumed without user benefits. The second problem occurs when the activity and the service are killed by the Android system. *PlaybackService* stops the playing of media and releases the wake lock in the *onDestroy()* callback (Lines 40–41). When a user presses the back button and exits the activity, the service will also be destroyed and the wake lock will be released. However, the Android system can kill processes when the memory is insufficient, in which case the *onDestroy()* callback of killed activities and services will not be invoked. Thus, the wake lock will not be released, affecting the battery life.
Source code	``` 1 public class PlaybackActivity extends Activity { 2 private ImageButton mPlayPauseButton; 3 private PlaybackService mPlaybackService; 4 private ServiceConnection mConnection = new ServiceConnection() { 5 public void onServiceConnected(6 ComponentName name, IBinder binder) { 7 mPlaybackService = ((MyBinder) binder).getService(); 8 } 9 }; 10 public void onCreate(Bundle state) { 11 mPlayPauseButton.setOnClickListener(new OnClickListener() { 12 public void onClick(View v) { 13 // toggle play/pause state 14 mPlaybackService.playPause(); 15 } 16 }); 17 Intent i = new Intent(this, PlaybackService.class); 18 // start PlaybackService 19 startService(i); 20 } 21 public void onResume() { 22 Intent i = new Intent(this, PlaybackService.class); 23 // bind PlaybackService 24 bindService(i, mConnection, Context.BIND_ABOVE_CLIENT); 25 } 26 } 27 28 public class PlaybackService extends Service 29 implements MediaPlayer.OnPreparedListener { 30 private static String TAG = PlaybackService.class.getName(); 31 private WakeLock mWakeLock; 32 private MediaPlayer mMediaPlayer; 33 public void onCreate() { 34 mWakeLock = getPowerManager().newWakeLock(35 PowerManager.PARTIAL_WAKE_LOCK, TAG); 36 mMediaPlayer = createAndSetupMediaPlayer(); 37 mMediaPlayer.setOnPreparedListener(this); 38 } ```

```
39      public void onDestroy() {
40          // stop media when the service is destroyed
41          stop();
42      }
43      public void onPrepared() {
44          // start media when the media player get prepared
45          start();
46      }
47      public void playPause() {
48          if (mMediaPlayer.isPlaying())
49              pause();
50          else
51              start();
52      }
53      public void start() {
54          // play media and acquire wake lock
55          mWakeLock.acquire();
56          mMediaPlayer.start();
57      }
58      public void stop() {
59          // stop media and release wake lock
60          mMediaPlayer.stop();
61          if (mWakeLock.isHeld()) mWakeLock.release();
62      }
63      public void pause() {
64          // pause media and release wake lock
65          mMediaPlayer.pause();
66          if (mWakeLock.isHeld()) mWakeLock.release();
67      }
68      public class MyBinder extends Binder {
69          PlaybackService getService() {
70              return PlaybackService.this;
71          }
72      }
73      public IBinder onBind(Intent intent) {
74          return new MyBinder();
75      }
76  }
```

Liu-ICSE14

BugID: PB6	PublicationDOI: 10.1145/2568225.2568229	Year: 2014		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Energy	Missing operation, Synchronization, Unnecessary computation	GUI, Language-specific, Multiple processes	Android (Java)
Description	Code and corresponding bug-fixing patch for an energy leak in Zmanim. When *ZmanimActivity* launches, it registers a location listener to receive location changes for updating its GUI (Lines 5–14). The location listener is normally unregistered when the activity is destroyed (Line 27). However, if a user launches Zmanim and then switches it to background (Android OS will call onPause() and onStop() handlers accordingly, but not onDestroy()), the application will keep receiving location changes to update its GUI, which is, however, invisible. The location sensing and GUI refreshing are then useless, but still drain battery power.			

Source code

```
1.   public class ZmanimActivity extends Activity {
2.     private ZmanimLocationManager lm;
3.     private ZmanimLocationManager.Listener locListener;
4.     public void onCreate() {
5.       //get a reference to system location manager
6.       lm = new ZmanimLocationManager(ZmanimActivity.this);
7.       locListener = new ZmanimLocationManager.Listener() {
8.         public void onLocationChanged(ZmanimLocation newLoc) {
9.           //build UI using obtained location in a new thread
10.          rebuildUI(newLoc);
11.        }
12.      };
13.      //register location listener
14.      lm.requestLocationUpdates(GPS, 0, 0, locListener);
15.    }
16.    public void onResume() {
17. +    //register location listener if UI still needs update
18. +    if(buildingUINotFinished)
19. +      lm.requestLocationUpdates(GPS, 0, 0, locListener);
20.    }
21.    public void onPause() {
22. +    //unregister location listener
23. +    lm.removeListener(locListener);
24.    }
25.    public void onDestory() {
26. -    //unregister location listener
27. -    lm.removeListener(locListener);
28.    }
29.  }
```

Liu-TSE14

BugID: PB7	PublicationDOI: 10.1109/TSE.2014.2323982		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4a	Energy	Unnecessary computation	Communication	Android (Java)
Description	This bug gives the concerned code snippet of a location data underutilization problem in an entertainment application Geohash Droid. This application randomly selects a location for users and navigates them there using GPS sensors. As the code shows, Geohash Droid maintains a long running *GeohashService* at background for location sensing. *GeohashService* registers a location listener with Android OS when it starts (lines 7-16), and unregisters the listener when it finishes (lines 22-25). Once it receives location updates, it refreshes the smartphone's notification bar (line 11), which provides users with quick access to their current locations. After that, it notifies remote listeners to use updated location data (lines 12, 27-36). However, there are chances when no remote listeners are alive. When this happens, Geohash Droid would keep receiving the phone's GPS coordinates, simply for updating its notification bar. Such updates do not reflect effective use of newly captured GPS coordinates, while the battery's energy is continuously consumed. Geohash Droid developers received a lot of user complaints for such battery drain.				

Source code

```
1.  public class GeohashService extends Service {
2.    private ArrayList<RemoteListener> mListeners;
3.    private LocationManager lm;
4.    private LocationListener gpsListener;
5.    public void onStart(Intent intent, int StartId){
6.      mListeners = new ArrayList<RemoteListener>();
7.      //get a reference to system location manager
8.      lm = getSystemService(LOCATION_SERVICE);
9.      gpsListener = new LocationListener() {
10.       public void onLocationChanged(Location loc) {
11.         updateNotificationBar(loc);
12.         notifyRemoteListeners(loc);
13.       }
14.     };
15.     //GPS listener registration
16.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
17.   }


21.   //more code from GeohashService
22.   public void onDestroy() {
23.     //GPS listener unregistration
24.     lm.removeUpdates(gpsListener);
25.   }
26.   //notify each alive remote listener for loc change
27.   public void notifyRemoteListeners(Location loc){
28.     final int N = mListeners.size();
29.     for(int i = 0; i < N; i++) {
30.       RemoteListener listener = mListeners.get(i);
31.       if(listener.isAlive()){
32.         //remote listeners consume location data
33.         listener.locationUpdate(loc);
34.       }
35.     }
36.   }
37. }
```

Liu-TSE14

BugID: PB8	PublicationDOI: 10.1109/TSE.2014.2323982		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4b	Energy	Unnecessary computation	Communication	Android (Java)
Description	This bug was found in the Osmdroid application. This application has three components: (1) *MapActivity* for displaying a map to its users, (2) *GPSService* for location sensing and data processing in background, and (3) a broadcast receiver for handling location change messages (lines 7-13). When *MapActivity* is launched, it starts *GPSService* (lines 5-6), and registers its broadcast receiver (lines 15-16). *GPSService* then registers a location listener with the Android OS when it starts (lines 36-47). When the application's users change their locations, *GPSService* would receive and process new location data (line 39), and broadcast a message with the processed data (lines 41-43). The broadcast receiver would then use the new location data to refresh a map (line 10). If the users have enabled location tracking, these location data would also be stored to a database (line 11). If the Android OS plans to destroy *MapActivity* (lines 18-22), *GPSService* would be stopped (line 20), and both the location listener and broadcast receiver would be unregistered (lines 21, 51). However, if Osmdroid's users switch from *MapActivity* to any other activity, *MapActivity* would be put to background (not destroyed), but *GPSService* would still keep running for location sensing. If the location tracking functionality is not enabled, all collected location data would be used to refresh an invisible map. Then, a huge amount of energy would be wasted.				

Source code

```
1.  public class MapActivity extends Activity {
2.    private Intent gpsIntent;
3.    private BroadcastReceiver myReceiver;
4.    public void onCreate(){
5.      gpsIntent = new Intent(GPSService.class);
6.      startService(gpsIntent); //start GPSService
7.      myReceiver = new BroadcastReceiver() {
8.        public void onReceive(Intent intent) {
9.          LocData loc = intent.getExtra();
10.         updateMap(loc);
11.         if(trackingModeOn) persistToDatabase(loc);
12.       }
13.     }
14.     //register receiver for handling location change messages
15.     IntentFilter filter = new IntentFilter("loc_change");
16.     registerReceiver(myReceiver, filter);
17.   }
18.   public void onDestroy() {
19.     //stop GPSService and unregister broadcast receiver
20.     stopService(gpsIntent);
21.     unregisterReceiver(myReceiver);
22.   }
23. }
```

```
31.  public class GPSService extends Service {
32.    private LocationManager lm;
33.    private LocationListener gpsListener;
34.    public void onCreate(){
35.      //get a reference to system location manager
36.      lm = getSystemService(LOCATION_SERVICE);
37.      gpsListener = new LocationListener() {
38.        public void onLocationChanged(Location loc) {
39.          LocData formattedLoc = processLocation(loc);
40.          //create and send a location change message
41.          Intent intent = new Intent("loc_change");
42.          intent.putExtra("data", formattedLoc);
43.          sendBroadcast(intent);
44.        }
45.      };
46.      //GPS listener registration
47.      lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
48.    }
49.    public void onDestroy() {
50.      //GPS listener unregistration
51.      lm.removeUpdates(gpsListener);
52.    }
53.  }
```

Liu-IEEESoftware15

BugID: PB9	PublicationDOI: 10.1109/MS.2015.4		Year: 2015	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2a	Energy	Unnecessary computation	Communication, GUI	Android
Description	The code can cause problems when users enter an area with weak GPS signals, the application might continue discarding noisy location data. This continual but used less location sensing can quickly drain the phone battery. Another problematic situation arises when users switch *MapActivity* to the background without enabling location tracking. In such cases, even if *GPSService* obtains precise location data, the data will be used only to render the navigation map, which is completely invisible. Again, this wastes battery energy.			

Source code	1. public class MapActivity extends Activity { 2. private Intent gpsIntent; 3. private BroadcastReceiver myReceiver; 4. public void onCreate(){ 5. gpsIntent = new Intent(GPSService.class); 6. startService(gpsIntent); //start GPSService 7. myReceiver = new BroadcastReceiver() { 8. public void onReceive(Intent intent) { 9. LocData loc= intent.getExtra(); 10. updateMap(loc); 11. if(trackingModeOn) { 12. persistToDatabase(loc); 13. } 14. } 15. } 16. //register receiver for handling location changes 17. IntentFilter filter = new IntentFilter("loc_change"); 18. registerReceiver(myReceiver, filter); 19 } 20. public void onDestroy() { 21. //stop GPSService and unregister broadcast receiver 22. stopService(gpsIntent); 23. unregisterReceiver(myReceiver); 24. } 25. }

Selakovic-ICSE15

BugID: PB10	PublicationDOI: 10.1109/ICSE.2015.260		Year: 2015	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1-1	Execution time	Call to API Method	Language-specific	Javascript
Description	Underscore issue based on lower built-in function			
Source code	Underscore issue 39	`str.split("'").join("\\'");`		

Jin-PLDI12

BugID: PB11	PublicationDOI: 10.1145/2254064.2254075		Year: 2012	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 1	Execution time	Call to API method	Server	C
Description	Apache HTTPD developers forgot to change a parameter of API *apr_stat* after an API upgrade. This mistake caused more than ten times slowdown in Apache servers.			
Source code	**Patch for Apache Bug 45464** modules/dav/fs/repos.c status = apr_stat (fscontext->info, - APR_DEFAULT); + APR_TYPE); **What is this bug** An Apache-API upgrade causes **apr_stat** to retrieve more information from the file system and take longer time. Now, APR_TYPE retrieves exactly what developers originally needed through APR_DEFAULT. **Impact**: causes httpd server **10+ times slower** in file listing			

Jin-PLDI12

BugID: PB12	PublicationDOI: 10.1145/2254064.2254075	Year: 2012

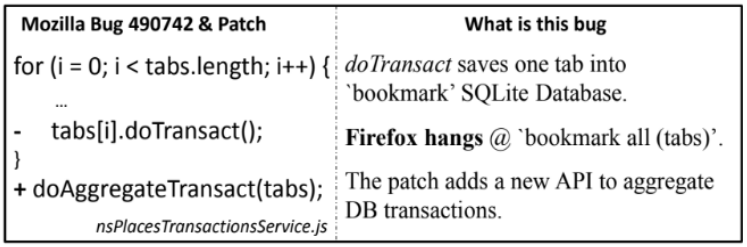
Performance bug data

Figure	Effect	Cause	Context	Language
Figure 4	Execution time	Call to API method	Web	JavaScript

Description

Users reported that Firefox hung when they clicked 'bookmark all (tabs)' with 20 open tabs. Investigation revealed that Firefox used N database transactions to bookmark N tabs, which is very time consuming comparing with batching all bookmark tasks into a single transaction. The addition of batchable functionalities such as 'bookmark all (tabs)' exposed this inefficiency problem.

Source code



Jin-PLDI12

BugID: PB13	PublicationDOI: 10.1145/2254064.2254075	Year: 2012

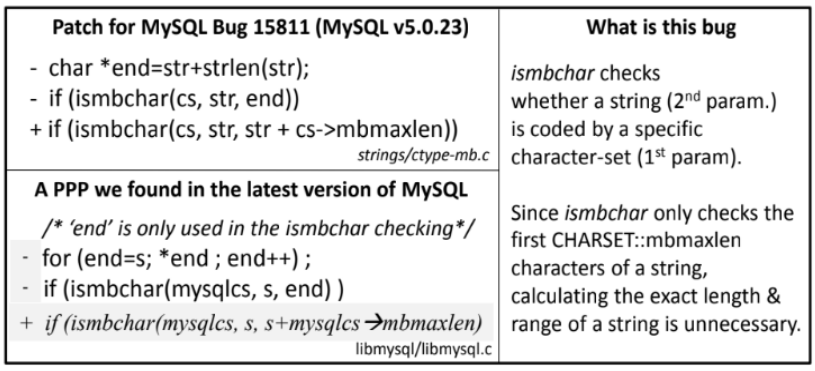
Performance bug data

Figure	Effect	Cause	Context	Language
Figure 6 (right)	Execution time	Call to API method, Condition, Loop	Server	Java

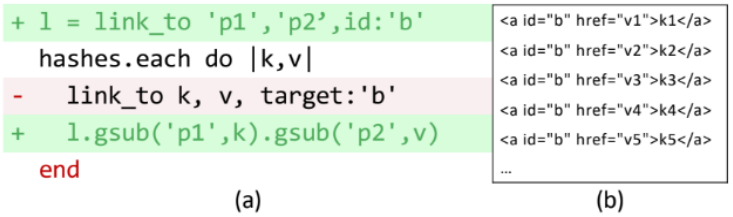
Description

See the figure.

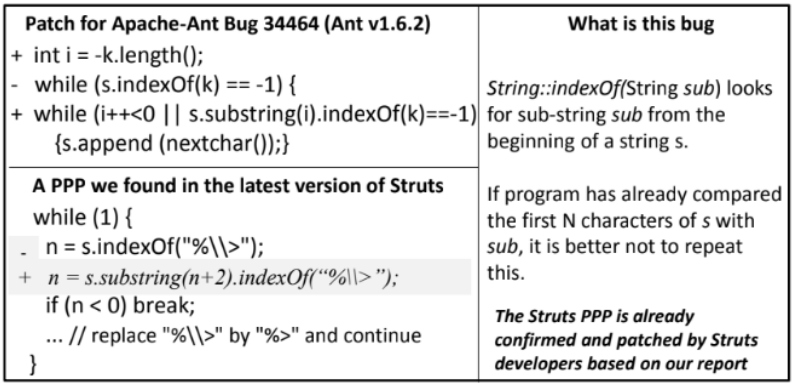
Source code



Yang-ESEC/FSE18

BugID: PB14	PublicationDOI: 10.1145/3236024.3264589		Year: 2018	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Execution time	Call to API Method, Loop, Query, Redundancy	Database, Language-specific, Web	SQL/Ruby
Description	While rendering a list of objects, helper functions are often used to render a partial view for one object at a time, with much redundant computation repeated for every object. For example, the HTML in the figure b is generated line by line by repeated invocations of *link_to* with much redundancy across lines. Such inefficiency is particularly severe when there are many objects to render.			
Source code	 (a) (b)			

Jin-PLDI12

BugID: PB15	PublicationDOI: 10.1145/2254064.2254075		Year: 2012	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 6 (left)	Execution time	Call to API method, Loop, Unnecessary computation	Database	C
Description	See the figure.			
Source code				

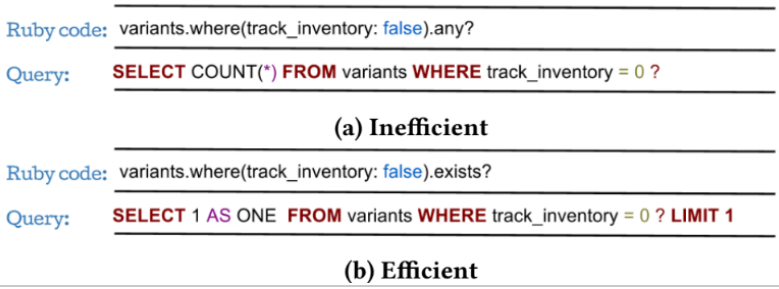
Yang-ESEC/FSE18

BugID: PB16	PublicationDOI: 10.1145/3236024.3264589			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Call to API Method, Query	Database, Language-specific, Web	SQL/Ruby
Description	API misuse from Onebody (the upper code is less efficient than the lower code) The two Rails code snippets both check if a *user* owns any blog posts. However, they use different APIs, *count* versus *exist*, that are translated to different SQL queries by Rails: *select count* vs. *select limit 1*. The former query scans all records in the blogs table with specified *user_id*, counts the number of records, and checks (in the Ruby application) if the count is greater than 0. The latter query returns immediately when it finds one record with the specific *user_id*.			
Source code	`Ruby: if user.blogs.count > 0` `Sql: select count(*) from blogs where user_id = ?` `Ruby: if user.blogs.exist?` `Sql: select 1 from blogs where user_id = ? limit 1`			

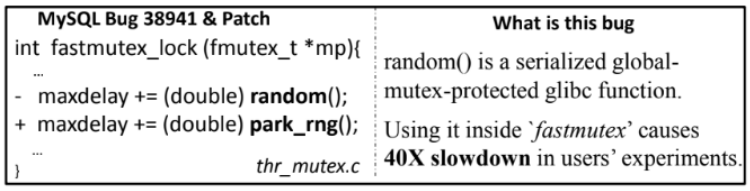
Song-OOPSLA14

BugID: PB17	PublicationDOI: 10.1145/2660193.2660234			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Execution time	Call to API method, Query	Database	C++
Description	The code shows a real-world performance problem in MySQL. MySQL users noticed surprisingly poor performance for queries on certain type of tables. Profiling could not provide any useful information, as the top ranked functions are either low-level library functions. After thorough code inspection, developers finally figured out that the problem is in function *start_bulk_insert*, which does not even get ranked by the profiler. The developer who implemented this function assumed that *parameter-0* indicates no need of cache, while the developers who wrote the caller functions thought that *parameter-0* indicates the allocation of a large buffer. This mis-communication led to unexpected cache-less execution, which is extremely slow. The final patch simply removes the unnecessary branch in the code, but it took developers a lot of effort to figure out.			
Source code	`void start_bulk_insert(ha_rows rows)` `{` `  ...` `-  if (!rows)` `-  { //slow path where caching is not used` `-  DBUG_VOID_RETURN;` `-  }` `-  rows = rows/m_tot_parts + 1;` `+  rows = rows ? (rows/m_tot_parts + 1) : 0;` `  ...` `  //fast path where caching is used` `  DBUG_VOID_RETURN;` `}`			

Yang-ICSE18

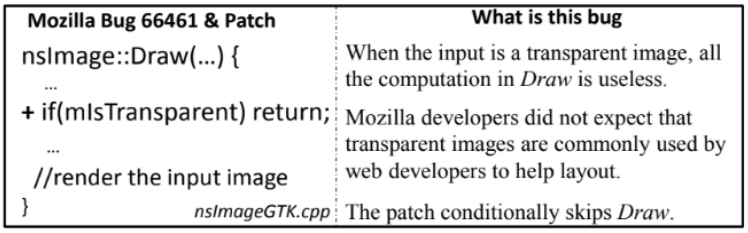
BugID: PB18	PublicationDOI: 10.1145/3180155.3180194			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 4	Execution time	Call to API method, Query	Database, Web	SQL/Ruby
Description	Figure 4 shows two ways that an online shopping system checks if there are product variants whose inventory are not tracked. The Ruby code differs only in the use of *any?* vs *exists?*. However, the performance of the queries differs substantially: the generated query in Figure 4(a) scans all records in the variants table to compute the count if no index exists, but that in Figure 4(b) only needs to scan and locate the first variant record where the predicate evaluates to true.			
Source code				

Jin-PLDI12

BugID: PB19	PublicationDOI: 10.1145/2254064.2254075			Year: 2012
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 5	Execution time	Call to API method, Synchronization	Database, Multiple processes, Language-specific	C
Description	MySQL synchronization-library developers implemented a *fastmutex* lock for fast locking. Unfortunately, users' unit test showed that *fastmutex* lock could be 40 times slower than normal locks. It turns out that library function *random()* actually contains a lock. This lock serializes every threads that invoke *random()*.			
Source code				

Jin-PLDI12

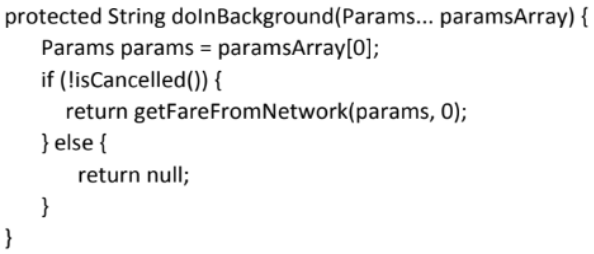
BugID: PB20	PublicationDOI: 10.1145/2254064.2254075			Year: 2012
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 2	Execution time	Call to API method, Unnecessary computation	Web	C++
Description	Mozilla developers implemented a procedure *nsImage::Draw* for figure scaling, compositing, and rendering, which is a waste of time for transparent figures.			

	This problem did not catch developers' attention until two years later when 1 pixel by 1 pixel transparent GIFs became general purpose spacers widely used by Web developers to work around certain idiosyncrasies in HTML 4.
Source code	

Selakovic-ICSE15

BugID: PB21	PublicationDOI: 10.1109/ICSE.2015.260			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1-3	Execution time	Condition	Language-specific	Javascript
Description	Mocha issue based on the non-use of new language features			
Source code	Mocha issue 701	`if (toString.call(err) === "[object Error]") { .. }`		

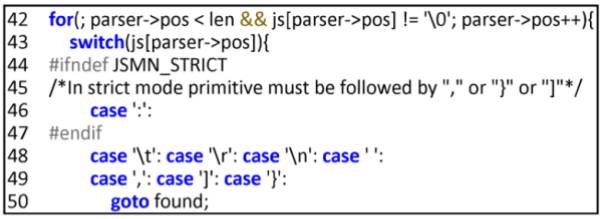
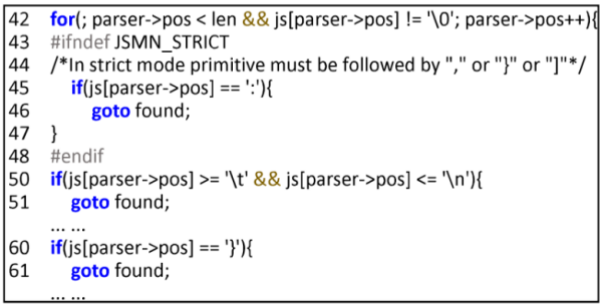
Kang-FSE16

BugID: PB22	PublicationDOI: 10.1145/2950290.2950316			Year: 2016
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Execution time	Condition, Missing operation	GUI, Multiple processes	Java (Android)
Description	Performance bug in BartRunnerAndroid. The cancelation of a time-consuming task is necessary when the task is no longer required. In the figure, the cancelation checking is done before the task begins, thus the task will not be canceled during execution when it is obsolete.			
Source code				

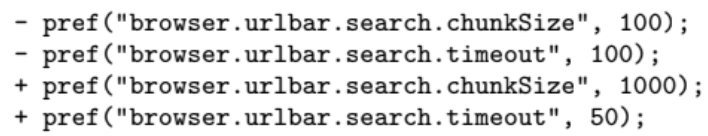
```
protected String doInBackground(Params... paramsArray) {
    Params params = paramsArray[0];
    if (!isCancelled()) {
        return getFareFromNetwork(params, 0);
    } else {
        return null;
    }
}
```

Chen-COMPSAC18

BugID: PB23	PublicationDOI: 10.1109/COMPSAC.2018.00044			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Execution time	Condition, Others (branch misprediction)	System software	C
Description	The swith statement at line 43 (figure at the top) is in a loop and incurs 64 branch mispredictions because CPU cannot precisely predict the branch to be taken. Given			

	that '\n' and '\t' are two adjacent characters, the switch statement can be replaced with multiple if statements (figure at the bottom).
Source code	 

Han-ESEM16

BugID: PB24	PublicationDOI: 10.1145/2961111.2962602	Year: 2016		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Execution time	Configuration	Database, Web	C++
Description	This code shows a slow autocomplete feature bug and its fix. The bug is due to misconfigurations of *browser.urlbar.search.chunkSize* and *browser.urlbar.search.timeout* preferences in Firefox, where the first option defines the number of chunks must be collected from SQLite database, and the second option defines the search timeout value. When the chunk size is set too small, the data query completes fast and waits for the search timeout to return and thus leading to performance degradation.			
Source code				

Kang-FSE16

BugID: PB25	PublicationDOI: 10.1145/2950290.2950316	Year: 2016		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Execution time	Configuration	GUI, Multiple processes	Java (Android)
Description	Performance bug in OpenLaw. Usually, developers wrongly assume the availability of the execution unit during task scheduling. Waiting for the completion of another heavy-weighted task should generally be avoided via proper scheduling. The comment “parallel update” indicates developers intend to execute the two tasks in parallel.			

Source code	public void onCreate(Bundle savedInstanceState) { ... // Load actual list final LawSectionList sectionDB = new LawSectionList(LawSectionList.TYPE_ALL); ... sectionDB.execute(adapter); // And parallel update the list from network UpdateLawList updater = new UpdateLawList(); updater.execute(adapter); ... }

Kang-FSE16

BugID: PB26	**PublicationDOI:** 10.1145/2950290.2950316	**Year:** 2016		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Execution time	Configuration	GUI, Multiple processes	Java (Android)
Description	Performance bug in FBReader, a popular e-book reader. Improper thread pool size is a common cause of long task queuing since the pool is often busy. In this figure, developers are not sure about the proper pool size, and hence put down a to-do comment in the source.			
Source code	private static final int IMAGE_LOADING_THREADS_NUMBER = 3;//TODO: how many threads? private final ExecutorService myPool = Executors.newFixedThreadPool(IMAGE_LOADING_THREADS_NUMBER, new MinPriorityThreadFactory()); ... void startImageLoading(...){ final ExecutorService pool = image.sourceType() == ZLImageProxy.SourceType.FILE ? mySinglePool : myPool; pool.execute(...); }			

Han-ESEM16

BugID: PB27	**PublicationDOI:** 10.1145/2961111.2962602	**Year:** 2016		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Configuration	Multiple processes, Server	C++
Description	This code shows a bug due to a wrong implementation of the page cleaner feature in MySQL and its fix. This bug led to a slowdown during the shutdown phase of database server.			
Source code	/*storage/innobase/srv/srv0start.cc*/ +for (i = 1; i < srv_n_page_cleaners; ++i){ + os_thread_create(buf_flush_page_ + cleaner_worker, NULL, NULL); + }			

Han-ESEM16

BugID: PB28	PublicationDOI: 10.1145/2961111.2962602		Year: 2016	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Execution time	Configuration	Server	C
Description	This code shows an abnormal termination bug and its fix. This bug happens when a child process receives a software signal that closes the listening sockets before the process starts polling the sockets using the *apr_pollset_poll* function, that is only called when multiple listening ports are present. Since no listening sockets are available (closed by the signal), the calling child process waits indefinitely.			
Source code	`for (;;) {` `+ status = apr_pollset_poll(pollset,` `- -1, ...); /*wait forever*/` `+ apr_time_from_sec(10), ...);` `+ if (APR_STATUS_IS_TIMEUP(status)` `+ continue;` `}`			

Han-ESEM16

BugID: PB29	PublicationDOI: 10.1145/2961111.2962602		Year: 2016	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Execution time	Configuration, Loop	Server	C
Description	This code shows a waste loop computation bug and its fix. To trigger a slowdown during the Apache graceful restart, user needs to start Apache with a relatively larger number for the *StartServers* option. In this case, the server takes more time to restart than usual. When the children servers already exited, there is no need to iterate these servers. By checking the existence of the children server processes first, the time-consuming *dummy_connection* function is skipped if the server no longer exists.			
Source code	`ap_mpm_pod_killpg(ap_pod_t *pod, int num){` `for (i=0;i<num && rv==APR_SUCCESS;i++) {` `+ if(ap_scoreboard_image->` `+ servers[i][0].status!=SERVER_READY` `+ || ap_scoreboard_image->servers[i][0].pid` `+ == 0)` `+ continue;` `+ rv=dummy_connection(pod);` `}` `}`			

He-ESEC/FSE19

BugID: PB30	PublicationDOI: 10.1145/3338906.3342498		Year: 2019	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Execution time	Configuration, Query	Database	SQL
Description	See the figure.			

Source code	**MySQL #21727** **Description**: A query of the type: "`SELECT a, b, (SELECT x FROM t2 WHERE y=b ORDER BY z DESC LIMIT 1) c FROM t1`" will get much slower as `sort_buffer_size` is increased. **Related Configuration**: `sort_buffer_size`(Session that needs to perform a sort will be allocated a buffer with this amount of memory.) **Root Cause**: Allocation of memory for the sort buffer at each evaluation of a subquery may take a significant amount of time if the buffer is rather big. In this case, memory accesses are implemented with complicated address manipulation (e.g. system calls and address calculations), and the bug only manifest under large configuration vales (`sort_buffer_size`).

Han-ESEM16

BugID: PB31	PublicationDOI: 10.1145/2961111.2962602		Year: 2016	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Execution time	Configuration, Synchronization	Multiple processes	C++
Description	This code shows a lock contention bug and its fix. There are two logging functions that use the l*og_sys->mutex* lock to write to a buffer. This inevitably causes lock contention that hurts the application performance.			
Source code	`/*Add one more buffer*/` `byte* buf_pair_ptr[2];` `...` `/** Acquire the log sys mutex. */` `#define log_mutex_enter_all() do {` `+ mutex_enter(&log_sys->w_mutex);` `mutex_enter(&log_sys->mutex);` `} while (0)` `...` `/** Release the log sys mutex. */` `#define log_mutex_exit_all() do {` `+ mutex_exit(&log_sys->w_mutex);` `mutex_exit(&log_sys->mutex);` `} while (0)`			

Lemieux-ISSTA18

BugID: PB32	PublicationDOI: 10.1145/3213846.3213874		Year: 2018	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Execution time	Data structure	General-practice	C
Description	The C program is a simplified version of wf. The main program driver (omitted from the figure for brevity) takes as input a string, splits the string into words at whitespaces, and counts how many times each word occurs in the input. To map words to integer counts, the program uses a simple hashtable (defined at Line 11) with a fixed number of buckets. The program exhibits a performance bottleneck when the input contains very long words (e.g., nucleic acid sequences) because the program will spend most of its time in the *compute_hash* function (it iterates over each character in the word irrespective of its length). Also, if the input contains many distinct words (e.g., e-mail addresses			

	from a server log), the frequency of hash collisions in the fixed-size hashtable increases dramatically. For such an input, the program will spend most of its time in the function *add_word*, traversing the linked list of entries in the loop at lines 28–37.
Source code	``` 1 // Hash-map entry; also a linked list node, 2 // to resolve hash collisions 3 typedef struct entry_t { 4 char* key; 5 int value; 6 struct entry_t* next; 7 } entry; 8 9 // Fixed-size table of hash-map entries. 10 const int TABLE_SIZE = 1001; 11 entry* hashtable[TABLE_SIZE] = {0}; 12 13 // Computes a hash value for a word. 14 unsigned int compute_hash(char* str) { 15 unsigned int hash = 0; 16 for (char* p = str; *p != '\0'; p++) { 17 hash = 31 * hash + (*p); 18 } 19 return hash % TABLE_SIZE; 20 } 21 // Increments word count in the hash-map. 22 void add_word(char* word) { 23 // access the appropriate hashtable bucket 24 int bucket = compute_hash(word); 25 entry* e = hashtable[bucket]; 26 27 // find matching entry 28 while (e != NULL) { 29 if (strcmp(e->key, word) == 0) { 30 // increment count 31 e->value++; 32 return; 33 } else { 34 // traverse linked list 35 e = e->next; 36 } 37 } 38 // If no entry found, create one 39 hashtable[bucket] = new_entry(word, 1, hashtable[bucket]); 40 } ```

Bornholt-OOSPLA18

BugID: PB33	PublicationDOI: 10.1145/3276519	Year: 2018		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Sec. 5.3	Execution time	Data structure	Others (solver-aided program)	Rosette
Description	This code shows a bug that leads to excessive path creation and merging. As the interpreter executes a program, it records each executed instruction in a *trace* - a list of executed instructions - which is used to visualize counterexamples. However, since this call is made with a symbolic path condition, which forces to merge the new and existing values of *trace* when performing the mutation. This leads to excessive path creation and merging. In essence, *trace* has an irregular representation.			

Source code	(define (record-trace msg) (set! trace (append trace (list msg))))

Nistor-ICSE13

BugID: PB34	PublicationDOI: 10.1109/ICSE.2013.6606602			Year: 2013
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Data structure, Loop	General-practice	Java
Description	A Google Core Libraries bug with an inefficient inner loop. In the else branch, the outer loop iterates over each element of this and checks if *c* contains the element (lines 9–13). When *c* is an ArrayList, contains performs a linear search (lines 18–21), which is inefficient, so it would have been better to iterate over *c* and call remove on the set because it has a more efficient inner loop.			
Source code	1 // SetDecorator class in Google Core Libraries contained this method call 2 set.removeAll(arrayList); 3 // Simplified from the AbstractSet class in the standard Java library 4 public boolean removeAll(Collection⟨?⟩ c) { 5 if (size() > c.size()) { 6 for (Iterator⟨?⟩ i = c.iterator(); i.hasNext();) 7 remove(i.next()); 8 } else { 9 for (Iterator⟨?⟩ i = iterator(); i.hasNext();) { // Outer Loop 10 if (c.contains(i.next())) { 11 i.remove(); 12 } 13 } 14 } 15 } 16 // Simplified from the ArrayList class in the standard Java library 17 public boolean contains(Object o) { 18 for (int i = 0; i < size; i++) { // Inner Loop 19 if (o.equals(elementData[i])) 20 return true; 21 } 22 return false; 23 }			

Li-Eurosys18

BugID: PB35	PublicationDOI: 10.1145/3190508.3190552			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 9	Execution time	Data structure, Loop	General-practice, Others (distributed systems)	C
Description	The code shows a loop that iterates through data collections. If the loop contains no update operations, it is considered as a synchronization loop (i.e., non-scalable).			
Source code	1 for (Block block: toAdd) { 2 addStoredBlock(block, ...); 3 ... 4 }			

Xu-OOPSLA12

BugID: PB36	PublicationDOI: 10.1145/2398857.2384690		Year: 2012	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Listing 1-a	Execution time	Data structure, Loop, Redundancy	General-practice	Java
Description	This code shows a real-world example that illustrates the problems of excessive object creation. Method *findEquivalentNodes* in class *SSAGraph* identifies expressions of equivalent types in a set of control flow graphs by iteratively visiting program entities in them. It is declared an anonymous visitor class for each type of program entity and created an object of the class to visit each entity object. However, many of these visitor objects are created in nested loops. The numbers of their instances can grow exponentially with the layer of loop nesting.			
Source code	`class SSAGraph{ void findEquivalentNodes() { for(CFG cfg: cfgs){ cfg.visit(new TreeVistior(){ void visitBlock(Block b){b.visitBlockNodes();} }); } ... } ... } class Block{ void visitBlockNodes(){ for(Statement s: statements){ s.visit(new NodeVisitor(){ void visitExpression(Expr e){e.visitChildren();} }); } ... } ... } class Expr{ void visitChildren(){ for(Expr child: children){ child.visit(new ExprVisitor(){...}); } ... } ... }`			

Xu-OOPSLA12

BugID: PB37	PublicationDOI: 10.1145/2398857.2384690		Year: 2012	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Listing 1-b	Execution time	Data structure, Loop, Redundancy	General-practice	Java
Description	This code shows a real-world example that illustrates the problems of frequent construction of data structures with the same shapes and content, where a more aggressive optimization can be applied. In this example, a new *SimpleDataFormat* object is created in each iteration of the loop to parse a string into a *Date* object.			
Source code	`for(String dateStr : dates){ SimpleDateFormat sdf = new SimpleDateFormat(); try{ Date newD = sdf.parse(dateStr); ... }catch(...) {...} }`			

Dhok-FSE16

BugID: PB38	PublicationDOI: 10.1145/2950290.2950360			Year: 2016
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Data structure, Loop, Redundancy	General-practice	Java
Description	The code shows a bug that can cause an execution slowdown because of the redundant computations in the method *datasetsMapped* (lines 13-17). This method traverses over datasets (line 14). During traversal, it invokes another method indexOf which also traverses over the same data structure (line 8). This results in O(n^2) complexity, where n is the number of elements present in datasets.			
Source code	1. public class CategoryPlot{ 2. private Map<Integer, CategoryDataset> datasets; 3. public CategoryPlot() { ... } 4. public void setDataset(int index, CategoryDataset set) { ... 5. this.datasets.put(index, set); ... 6. } 7. public int indexOf(CategoryDataset dataset) { 8. for (Entry entry: this.datasets.entrySet()) 9. if (entry.getValue() == dataset) 10. return entry.getKey(); 11. return -1; 12. } 13. private List<CategoryDataset> datasetsMapped(int idx){ 14. for (CategoryDataset set:this.datasets.values()){ ... 15. int i = indexOf(set); ... 16. } 17. } 18. public List getCategoriesForAxis(CategoryAxis axis) { 20. int axisIndex = getDomainAxisIndex(axis); 21. datasetsMapped(axisIndex)) ... 22. } 24. }			

Song-ICSE17

BugID: PB39	PublicationDOI: 10.1109/ICSE.2017.41			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 1	Execution time	Data structure, Loop, Redundancy	General-practice, System software	C
Description	The figure shows a performance problem in GCC. Recursive function *mult_alg* computes the best algorithm for multiplying *t*, a time-consuming computation. At run time, *mult_alg* is often invoked for many times, and often with the same parameter partly due to its recursive nature. To avoid redundant computation across different instances of *mult_alg*, developers used a hash-table *alg_hash* to remember which parameter t has been processed in the past and what is the result. Unfortunately, a mistake in the type declaration of hash-table entry *hash_entry* makes the memoization useless for large *t*. In many cases, a slow path is taken, when the fast path should have been taken.			

Source code	``` struct hash_entry { - unsigned int t; + HOST_WIDE_UINT t; }; void mult_alg(... HOST_WIDE_UINT t, ...) { hash_index = hash (t); if (alg_hash[hash_index].t == t) { //fast path: reuse previous results }else{ //slow path: expensive recursive computation ... mult_alg (...); } } ```

Olivo-PLDI15

BugID: PB40	PublicationDOI: 10.1145/2813885.2737966	Year: 2015		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Execution time	Data structure, Redundancy	General-practice	Java
Description	The code shows a redundant traversal bug. The method invocation on line 14 is a virtual call with many possible targets, one of which is the *drawItem* method of *CandlestickRenderer* (lines 20–32). The performance problem arises because the *drawItem* method iterates over all points within the series in order to draw a single data point. However, since the data set is not modified between successive calls to *drawItem*, the recomputation of *xxWidth* in each call to *drawItem* is redundant and needlessly traverses a potentially large list of data items many times.			
Source code	``` 1. public boolean render(Graphics2D g2, Rectangle2D 2. dataArea, int index, ...) { 3. ... 4. XYDataset dataset = getDataset(index); 5. XYItemRenderer renderer = getRenderer(index); 6. ... 7. int sCount = dataset.getSeriesCount(); 8. int series; 9. for (series=sCount-1; series >= 0; series--) { 10. int first = 0; 11. int last = dataset.getItemCount(series) - 1; 12. ... 13. for (item=first; item <= last; item++) { 14. renderer.drawItem(dataset, series, item,...); 15. } 16. ... 17. } 18. ... 19. } 20. public void drawItem(XYDataSet dataset, 21. int series, int item, ...) { 22. ... 23. OHLCDataset highLowData = (OHLCDataset)dataset; 24. itemCount = highLowData.getItemCount(series); 25. double xxWidth = dataArea.getWidth(); 26. for(int i=0; i< itemCount; i++) { 27. ... 28. if(last != -1) { 29. xxWidth=Math.min(xxWidth,Math.abs(pos-last)); 30. } 31. } 32. } ```			

Yang-ICPP12

BugID: PB41	PublicationDOI: 10.1109/ICPP.2012.30			Year: 2012
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 10	Execution time	Data structure, Synchronization	Multiple processes, System software	C++
Description	Code fragment of an optimized (but buggy) kernel of transpose-matrix-vector multiplication (tmv). The shared memory is used to exploit the reuse of the vector *B* and all the global memory accesses are coalesced. There are two main issues with the code shown below. First, the 'float' data type is used. The 'float' data type is well supported on GTX480 but HD5870 prefers the 'float2' type. Second, different GPUs favor different ILP vs. TLP tradeoffs. Between GTX480/285 and HD5870, GTX480/285 prefers a high number of light-weighted threads while HD5870 is more efficient when each thread has enough computations to populate its 5-way VLIW pipeline. For the kernel code in the code, each thread computes a dot product of one column in the matrix *A* and the vector *B* and generates one element in the product vector *C*. For the matrix size of 1k×1k, it means that there are 1k threads and each thread computes the dot product of two 1k-entry vectors. For both GTX480/285 and HD5870, the number of threads is too low to hide the memory latency.				
Source code	`__kernel void tmv(__global float *A, __global float *B, ……){` `__local float shared_1[blockDimX];` `uint y = get_global_id(0);` `floatdotProduct = 0;` `for (uint x = 0; x < height; x = x + blockDimX){` `shared_1[(tidx+0)]=B[(x+tidx)];` `barrier(CLK_LOCAL_MEM_FENCE);` `for(uint i = 0; i < blockDimX; i ++)` `dotProduct += A[y + (x+i)*width] * shared_1[i];` `barrier(CLK_LOCAL_MEM_FENCE);` `}` `C[y] = dotProduct; }`			

Su-LCPC07

BugID: PB42	PublicationDOI: 10.1007/978-3-540-85261-2_16			Year: 2007
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Execution time	Inter-procedural	Communication	C
Description	Fiber distribution code containing a performance bug due to lack of immutable keyword. The programmer meant to add the "*immutable*" keyword to the declaration for the *FiberDescriptor* class. But the keyword was missing. Immutable classes extend the notion of Java primitive type to classes. For this example, if the *FiberDescriptor* were immutable, then the array copy prior to the foreach loop would copy every element in the *globalFibersArray* to the *localFibersArray* including the fields of each element. Without the "*immutable*" keyword, each processor only contains an array of pointers in *localFibersArray* to *FiberDescriptor* objects that live on processor 0. When each processor other than processor 0 accesses the fields of a *FiberDescriptor* object, a request and reply message would occur between the processor accessing the field and processor 0. This performance bug is hard to find manually because the source of the bug and the place where the problem is observed are far from each other.				

Source code	`// missing immutable keyword` `class FiberDescriptor{` `  public long filepos;` `  public double minx, maxx, miny, maxy, minz, maxz;` `  ...` `}` `/* globalFibersArray and the elements in it live on processor 0 */` `FiberDescriptor [1d] globalFibersArray;` `FiberDescriptor [1d] local localFibersArray;` `...` `localFibersArray.copy(globalFibersArray);` `foreach (p in localFibersArray.domain()){` `  FiberDescriptor fd = localFibersArray[p];` `  /* Determine if fd belongs to this processor by examining the fields of fd */` `  ...` `}`

Xiao-APLAS15

BugID: PB43	PublicationDOI: 10.1007/978-3-319-26529-2 18	Year: 2015		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 5	Execution time	Inter-procedural, Others (inconsistent field ordering)	General-practice	JavaScript
Description	The program splay.js contains a performance issue caused by the underscored statements in function *insert*. There is an instance of the typical inconsistent field ordering problem, where the fields “left” and “right” are added to the “SplayTree.Node” objects in different orders. As a consequence, when these “SplayTree.Node” objects are accessed, they would generate PICs and incur additional type checking overhead. These objects would deoptimize the *remove* function through the IC site at Line 16. The performance degradation would be prominent, because splay.js frequently inserts and removes nodes from the tree.			
Source code	`1 SplayTree.prototype.insert =` `2 function(key, value) {` `3  // ...` `4  var node =` `5  new SplayTree.Node(key, value);` `6  if (key > this.root_.key) {` `7  node.left = this.root_;` `8  node.right = this.root_.right;` `9  this.root_.right = null;` `10  } else {` `11  node.right = this.root_;` `12  node.left = this.root_.left;` `13  this.root_.left = null;` `14  }` `15  this.root_ = node;` `16 };` `16 SplayTree.prototype.remove =` `17 function(key) {` `18  // ...` `19  if (!this.root_.left) {` `20  this.root_ =` `21  this.root_.right;` `22  } else {` `23  // ...` `24  }` `25 };`			

Xiao-APLAS15

BugID: PB44	PublicationDOI: 10.1007/978-3-319-26529-2 18	Year: 2015		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 6	Execution time	Inter-procedural, Others (dynamic typing)	Language-specific	JavaScript
Description	A performance bug in the program box2d.js incurs deoptimizations for seven functions by adding a field “m toi”. This field addition operation is performed in			

	function *h.SolveTOI*. We show a simplified version of *h.SolveTOI* (left), where we highlight the two access sites for field “m toi”: Line 7 is a read site and Line 10 is a write site. Line 10 changes the type of the objects referenced by “b”, which deoptimize quite a few functions, such as those in the figure at the right.
Source code	1 h.prototype.SolveTOI = 2 function(a) { 3 // ... 4 for (; ;) { 5 // ... 6 if (b.m_flags & 1. e_toiFlag) 7 c = b.m_toi; 8 else { 9 // ... 10 b.m_toi = c; 11 } 12 } 13 }; 1 A.prototype.GetNext = 2 function () { 3 return this.m_next 4 }; 5 A.prototype.GetFixtureA = 6 function () { 7 return this.m_fixtureA 8 }; 9 A.prototype.GetFixtureB = 10 function () { 11 return this.m_fixtureB 12 };

Xiao-APLAS15

BugID: PB45	PublicationDOI: 10.1007/978-3-319-26529-2 18	Year: 2015

Performance bug data

Figure	Effect	Cause	Context	Language
Figure 8	Execution time	Inter-procedural, Others (dynamic typing)	Language-specific	JavaScript

Description	Gbemu.js uses a big monolithic data structure named *gameboy* to store the virtual machine states. In this flat design, almost all the fields of *gameboy* are added by the constructor *GameBoyCore* and most of these fields are integers. One representative issue is caused by the integer overflow of two fields: *CPUCyclesTotal* and *CPUCyclesTotalCurrent*. Taking *CPUCyclesTotal* as an example, its value can exceed 2_{30} at Line 15 which is the upper bound for the small integer representation used by V8. The integer overflow triggers a representation change to use double value for *CPUCyclesTotal*. As a consequence, all fields in the object “gameboy” are lifted to double representations, and all operations related to these fields are impacted.
Source code	1 GameBoyCore.prototype.initializeTiming = function () { 2 // ... 3 this.CPUCyclesTotal = (this.baseCPUCyclesPerIteration – this. CPUCyclesTotalRoundoff) | 0; 4 } 5 GameBoyCore.prototype.audioUnderrunAdjustment = function () { 6 // ... 7 this.CPUCyclesTotalCurrent += (underrunAmount >> 1)*this.machineOut; 8 } 9 GameBoyCore.prototype.iterationEndRoutine = function () { 10 // ... 11 this.CPUCyclesTotalCurrent += this.CPUCyclesTotalRoundoff; 12 } 13 GameBoyCore.prototype.recalculateIterationClockLimit = function () { 14 // ... 15 this.CPUCyclesTotal = this.CPUCyclesTotalBase + this. CPUCyclesTotalCurrent – endModulus; 16 this.CPUCyclesTotalCurrent = endModulus; 17 }

Li-Eurosys18

BugID: PB46	PublicationDOI: 10.1145/3190508.3190552			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Execution time	Loop	General-practice, Others (distributed systems)	C
Description	The code shows a loop with a workload-sensitive bound. The return value of the I/O operation at line 1 affects the loop bound at line 21.			
Source code				

```
1  File[] blockFiles = dir.listFiles();  // Java I/O API
2  for (i=0; i < blockFiles.length; i++) {
3    ...
4    blockSet.add(new Block(..)); //augmentation
5  }
6  ...
7  BlockListAsLongs(blockSet.toArray(alist));
8  ...
9  BlockListAsLongs(long[] list) {
10     blockList = list ? list : new long [0];
11 }
12 ...
13 NumBlocks() {
14     return blockList.length/LONGS_PER_BLOCK;
15 }
16 ...
17 for (i=0; i < report.NumBlocks(); i++) {
18     toAdd.add(block);           //augmentation
19 }
20 ...
21 for (Block block: toAdd) {
22     ...
23 }              //toAdd has type Collection<Block>
```

Li-Eurosys18

BugID: PB47	PublicationDOI: 10.1145/3190508.3190552			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Execution time	Loop	General-practice, Others (distributed systems)	C
Description	The code shows a loop with a workload-sensitive index. Return values of I/O operations define the loop index variable and hence determine the loop count.			
Source code				

```
1 while (bytesRead >= 0) {
2     ...
3     bytesRead = in.read(buf);
4 }
```

Li-Eurosys18

BugID: PB48	PublicationDOI: 10.1145/3190508.3190552			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 8	Execution time	Loop	General-practice, Others (distributed systems)	C
Description	The code shows a loop with constant stride. *idx* is updated with a constant increment in every iteration of the loop, then *volumes.length* is considered as an approximation for the loop-count's upper bound and the loop is considered as a possibly type-3 non-scalable loop.			
Source code	`1 for (int idx = 0; idx < volumes.length; idx++) {` `2 volumes[idx].getBlockInfo(blockSet);` `3 }`			

Song-ICSE17

BugID: PB49	PublicationDOI: 10.1109/ICSE.2017.41			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 3	Execution time	Loop, Missing operation	General-practice, System software	C
Description	A resultless [0|1]* bug in GCC. [0|1]* loops may or may not produce results in each iteration. As the algorithm behind the loop has quadratic complexity in the number of operands in an expression, programs with long expressions suffer severe slow-downs. After further diagnosis, developers observed that this loop rarely had any side-effects, as the if condition was rarely satisfied.			
Source code	`+ if(warning_candidate_p(add->expr)) {` `for (tmp = *to; tmp; tmp = tmp->next)` `if (candidate_equal_p (tmp->expr, add->expr)` `&& !tmp->writer)` `{` `...` `tmp->writer = add->writer;` `}` `+ }`			

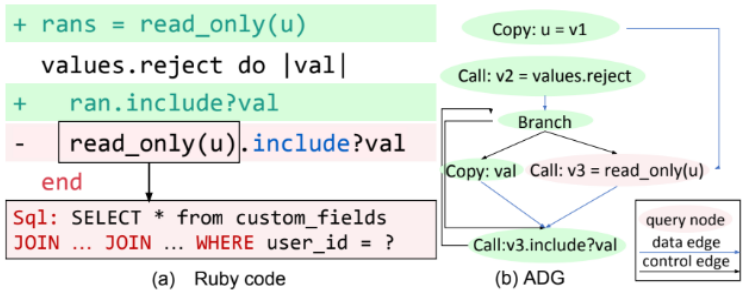
Xiao-ISSTA13

BugID: PB50	PublicationDOI: 10.1145/2483760.2483784			Year: 2013
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1 (second bug)	Execution time	Loop, Others (non-trivial operation)	GUI	C++
Description	This code shows a workload-dependent performance bottlenecks is caused by the method *GetSelectedItemsIndices*, which contains a workload-dependent loop (Lines 11-12).			
Source code	`9 void GetSelectedItemsIndices(CRecordVector<UInt32> &indices) {` `10 indices.Clear();` `11 for (int i = 0; i < _selectedStatusVector.Size(); i++) // PB_2` `12 if (_selectedStatusVector[i]) indices.Add(i);` `13 ... }`			

Yang-ESEC/FSE18

BugID: PB51	PublicationDOI: 10.1145/3236024.3264589	Year: 2018

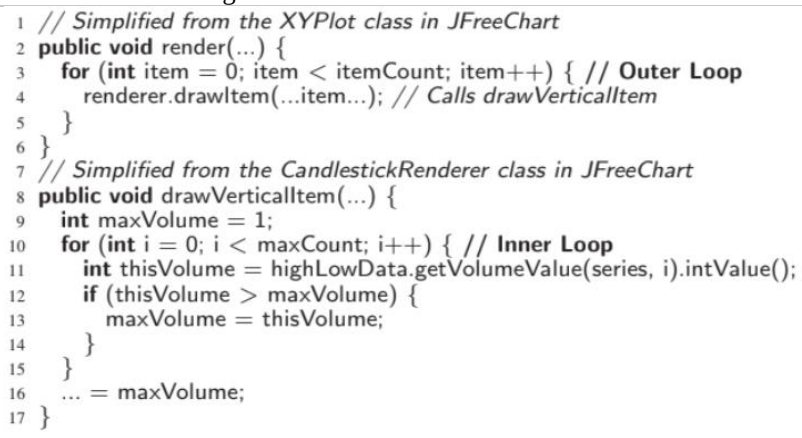
Performance bug data

Figure	Effect	Cause	Context	Language
Fig. 1	Execution time	Loop, Query, Redundancy	Database, Web	SQL/Ruby
Description	Loop invariant query from Redmine (the code checks which *val* in values list belongs to user *u*'s read-only fields). In loop invariant queries, a query is repeatedly issued in every iteration of a loop to load the same database contents. In the real-world example shown in a, hoisting the query out of the loop can speed up the application by more than 10×			
Source code				

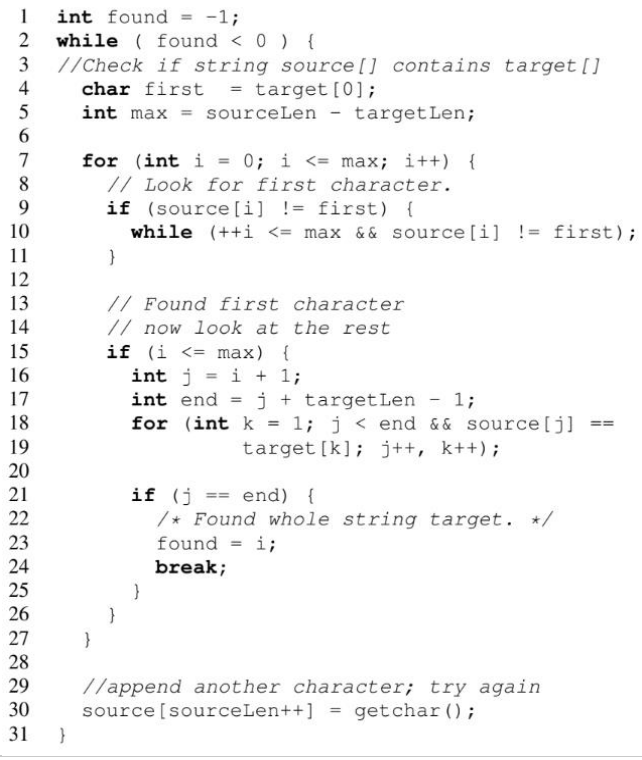
Nistor-ICSE13

BugID: PB52	PublicationDOI: 10.1109/ICSE.2013.6606602	Year: 2013

Performance bug data

Figure	Effect	Cause	Context	Language
Fig. 1	Execution time	Loop, Redundancy	General-practice	Java
Description	A JFreeChart bug with a redundancy in the outer loop. This bug is particularly severe, because it causes the chart display to freeze. The inner loop (line 10) in *drawVerticalItem* computes the maximum volume (line 12) of all the items in the data set. The repeated computation of maximum is redundant, because the volumes of the items do not change between calls.			
Source code				

Song-ICSE17

BugID: PB53	PublicationDOI: 10.1109/ICSE.2017.41			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 5	Execution time	Loop, Redundancy	General-practice, Server	C
Description	A cross-loop redundant bug in Apache (the *for* loop on line 7 searches a string source for a target sub-string target. Since the outer loop on line 2 appends one character to source in every iteration (line 30), the *for* loop is always working on a similar source from its previous execution, with a lot of redundancy.)			
Source code				

```
1  int found = -1;
2  while ( found < 0 ) {
3  //Check if string source[] contains target[]
4    char first  = target[0];
5    int max = sourceLen - targetLen;
6
7    for (int i = 0; i <= max; i++) {
8      // Look for first character.
9      if (source[i] != first) {
10       while (++i <= max && source[i] != first);
11     }
12
13     // Found first character
14     // now look at the rest
15     if (i <= max) {
16       int j = i + 1;
17       int end = j + targetLen - 1;
18       for (int k = 1; j < end && source[j] ==
19                target[k]; j++, k++);
20
21       if (j == end) {
22         /* Found whole string target. */
23         found = i;
24         break;
25       }
26     }
27   }
28
29   //append another character; try again
30   source[sourceLen++] = getchar();
31 }
```

Song-ICSE17

BugID: PB54	PublicationDOI: 10.1109/ICSE.2017.41			Year: 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 4	Execution time	Loop, Redundancy	General-practice, Web	C
Description	A cross-loop redundant bug in Mozilla. Cross-loop redundancy occurs when one dynamic instance of a loop spends a big chunk of its computation in repeating the work already done by an earlier instance of the same loop. The buggy loop in this figure counts how many previous siblings of the input *aNode* have the same name and URI. There is an outer loop, not shown in the figure, that repeatedly updates *aNode* to be its next sibling and calls *sss_xph_generate* with the new *aNode*.			

Source code	`char * sss_xph_generate(node_t* aNode)` `{` `  int count=0;` `  for (n = aNode; n ; n = aNode->prev)` `  if (n->localName == aNode->localName` `  && n->namespaceURI == aNode->namespaceURI)` `  count++;` `  ...` `}//called for every node in a list`

Li-Eurosys18

BugID: PB55	**PublicationDOI:** 10.1145/3190508.3190552			**Year:** 2017
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Execution time	Loop, Redundancy, Synchronization	Others (distributed systems)	C
Description	The code shows a loop with a loop-invariant index that cannot exit until another thread updates a shared variable with a loop-terminating value. The durations of these synchronization loops are non-deterministic and could be affected by workloads.			
Source code	`1 while (rjob.localing) {` `2  rjob.wait();` `3 }`			

Sakalis-ISPASS16

BugID: PB56	**PublicationDOI:** 10.1109/ISPASS.2016.7482078			**Year:** 2016
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Execution time	Loop, Synchronization	Multiple processes	C
Description	This code shows data races that happen when the barrier counter is checked in two places without acquiring any locks, while other threads might be updating it.			

Source code

```
1  void process_tasks(...) {
2    ...
3    t = DEQUEUE_TASK(...);
4  retry_entry:
5    while(t) {
6      switch(t->task_type) {
7        ...
8      }
9      t = DEQUEUE_TASK(...);
10   }
11   ...
12   while(global->pbar_count < n_processors) {
13     if(_process_task_wait_loop())
14       break ;
15     t = DEQUEUE_TASK(...)
16     if(t) {
17       LOCK(global->pbar_lock);
18       global->pbar_count-- ;
19       UNLOCK(global->pbar_lock);
20       goto retry_entry ;
21     }
22   }
23   BARRIER(global->barrier , n_processors);
24 }
```

(a) Barrier loop

```
1  long _process_task_wait_loop() {
2    finished = 0 ;
3    for(i = 0; i < 1000 && !finished ; i++) {
4      if((i & 0xff) == 0)
5        if((volatile long)global->pbar_count
6          >= n_processors)
7            finished = 1 ;
8    }
9    return(finished);
10 }
```

Welton-CCGRID18

BugID: PB57	PublicationDOI: 10.1109/CCGRID.2018.00045	Year: 2018		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Execution time	Loop, Synchronization, Unnecessary computation	Multiple processes,	C
Description	A flat representation of an explicit synchronization error in the main computational loop in Hoomd, which is used to wait for the GPU to update the shared variable *sharedStatus*. *sharedStatus* indicates whether the GPU computation failed because not enough GPU memory was allocated for the operation. The value of *sharedStatus* is true (successful GPU completion) for every iteration of the for-loop except the first iteration when GPU memory is initially allocated by the CPU. Even though the value of *sharedStatus* is false for iterations 2 to N of the for-loop, the application still synchronizes with the GPU on every iteration causing the reduction in performance by delaying the unrelated CPU computation.			

Source code

```
// Status variables shared between CPU and GPU
bool sharedStatus;
int * GPUData;

// Size of GPUData
int size = 0;
cudaMalloc(&GPUData, size);

// Main computational loop of hoomd
for(step = 0; step < nsteps; step++) {
    ...
    do {
        GPUComputation<<< >>>(sharedStatus,
                              GPUData,
                              size,
                              ...);

        // Synchronize to get GPU updates
        // to sharedStatus
        cudaDeviceSynchronize();

        // If sharedStatus is false...
        // allocate more GPU memory and retry
        if(sharedStatus == false) {
            size = len(GPUData) + ...;
            cudaMalloc(&GPUData, size);
        }
    } while(sharedStatus == false);

    // CPU work not dependant on GPU data
    for(i = 0; i < count; i++)
        ...
    ...

    // Existing Implicit Synchronization
    cudaMemcpy(...)
    ...
}
```

Red statements depend on results from GPU

Nistor-ICSE15

BugID: PB58	PublicationDOI: 10.1109/ICSE.2015.100	Year: 2015

Performance bug data

Figure	Effect	Cause	Context	Language
Fig. 10	Execution time	Loop, Unnecessary computation	General-practice	C

Description

Bug from GCC. The RI is executed and its result changes the values used by computation after the loop, but, under the L-Break condition, this result does not affect the perceived outcome of the program.

Source code

```
1 bool irred_invalidated = ...
2 FOR_EACH_EDGE (ae, ei, e->src->succs) {
3 + if (irred_invalidated) break; // FIX
4   if (ae != e && ae->dest != EXIT_BLOCK_PTR
5         && !bitmap_bit_p (seen, ae->dest->index)
6         && ae->flags & EDGE_IRREDUCIBLE_LOOP) {
7     irred_invalidated = true; // RI
8 }}
```

Nistor-ICSE15

BugID: PB59	PublicationDOI: 10.1109/ICSE.2015.100			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Loop, Unnecessary computation	General-practice	Java
Description	Bug caused when *argTypes* is initialized to a non-empty array, because the Result instruction (Line 7) cannot execute throughout the loop, the entire loop computation is wasted, and the loop can just be skipped.			
Source code	1 Class[] argTypes = ... 2 for (Iterator i = methods.iterator(); i.hasNext();) { 3 + if (!(argTypes == null) && !(argTypes.length == 0)) break; // FIX 4 MethodNode mn = (MethodNode) i.next(); 5 boolean isZeroArg = (argTypes == null || argTypes.length == 0); 6 boolean match =mn.getName().equals(methodName) && isZeroArg; 7 if (match) return true; // RI 8 }			

Nistor-ICSE15

BugID: PB60	PublicationDOI: 10.1109/ICSE.2015.100			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Execution time	Loop, Unnecessary computation	General-practice	Java
Description	Performance bug from PDFBox, which presents two buggy loops, loop1 and 2. For the loop iterations, once *alreadyPresent* is set to true on line 10, the condition on line 12 remains false for the remainder of the loop, and all the remaining computation in the loop can just be skipped.			
Source code	1 boolean alreadyPresent = false; 2 while (itActualEmbeddedProperties.hasNext()) { // Loop 1 3 + if (alreadyPresent) break; // FIX 4 ... // non-RIs 5 while (itNewValues.hasNext()) { // Loop 2 6 ... // non-RIs 7 while (itOldValues.hasNext() && !alreadyPresent) { // Loop 3 8 oldVal = (TextType) itOldValues.next(); 9 if(oldVal.getStringValue().equals(newVal.getStringValue())){ 10 alreadyPresent = true; // RI 1 11 }} 12 if (!alreadyPresent) { 13 embeddedProp.getContainer().addProperty(newVal);// RI 2 14 }}}}}			

Nistor-ICSE15

BugID: PB61	PublicationDOI: 10.1109/ICSE.2015.100			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Execution time	Loop, Unnecessary computation	General-practice	Java
Description	Performance bug from Tomcat which contains two instructions RI1 and RI2 that set variable *elExp* to true. Once either RI is executed, the remaining computation in the loop is unnecessary, at best setting *elExp* to true again. The I-Break condition for RI1			

	is that *elExp* equals true. RI2 has the same I-Break condition. The L-Break condition is the conjunction of the two I-Break conditions, i.e., *elExp* equals true.
Source code	1 **boolean** elExp = ... 2 **while** (nodes.hasNext()) { 3 + **if** (elExp) **break**; // **FIX** 4 ELNode node = nodes.next(); 5 **if** (node instanceof ELNode.Root) { 6 **if** (((ELNode.Root) node).getType() == '$') { 7 elExp = **true**; // **RI 1** 8 } **else if** (checkDeferred && ((ELNode.Root) node).getType()=='#' 9 && !pageInfo.isDeferredSyntaxAllowedAsLiteral()) { 10 elExp = **true**; // **RI 2** 11 }}} (a) A Type 3 RI in a Tomcat performance bug 1 valid &= child.validate(); // **RI**

Nistor-ICSE15

BugID: PB62	PublicationDOI: 10.1109/ICSE.2015.100			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Execution time	Loop, Unnecessary computation	General-practice	Java
Description	Performance bug from JMeter. The variable *length* keeps getting overwritten by the RI instruction. Consequently, all iterations before the last iteration that writes *length* are wasted.			
Source code	1 **int** length = ... 2 **for** (**int** idx = 0; idx < headerSize; idx++) { 3 @ **for** (**int** idx = headerSize - 1; idx >= 0; idx--) { // **FIX** 4 Header hd = mngr.getHeader(idx); 5 **if** (HTTPConstants.HEADER.equalsIgnoreCase(hd.getName())) { 6 length = Integer.parseInt(hd.getValue()); // **RI** 7 + **break**; // **FIX** 8 }}			

Nistor-ICSE15

BugID: PB63	PublicationDOI: 10.1109/ICSE.2015.100			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Execution time	Loop, Unnecessary computation	General-practice	Java
Description	Bug from PDFBox with 9 bug instructions. RI1 and RI2 is due to once *annotNotFound* is set to false (line 9), *annotNotFound* cannot become true again and the condition on line 6 evaluates to false in the remaining loop iterations. Similar reasoning applies for RI 4–8, *sigFieldNotFound* (line 14), and the condition on line 11. RI3 and RI9 are simultaneously similar to RI 1–2 and RI 4– 8. The L-Break condition is the conjunction of all nine I-Break conditions, i.e., both *annotNotFound* and *sigFieldNotFound* equal false.			

Source code	1 **boolean** annotNotFound = ... 2 **boolean** sigFieldNotFound = ... 3 **for** (COSObject cosObject : cosObjects) { 4 + **if** (!annotNotFound && !sigFieldNotFound) **break**; // **FIX** 5 ... // *some non−RIs* 6 **if** (annotNotFound && COSName.ANNOT.equals(type)) { 7 ... // **RI 1 and some non-RIs** 8 signatureField.getWidget().setRectangle(rect); // **RI 2** 9 annotNotFound = **false**; // **RI 3** 10 } 11 **if** (sigFieldNotFound && COSName.SIG.equals(ft)&&apDict!=**null**){ 12 ... // **RI 4, RI 5, RI 6, RI 7 and some non-RIs** 13 acroFormDict.setItem(COSName.DR, dr); // **RI 8** 14 sigFieldNotFound=**false**; // **RI 9** 15 }}

Song-ICSE17

BugID: PB64	**PublicationDOI:** 10.1109/ICSE.2017.41		**Year:** 2017	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 2	Execution time	Loop, Unnecessary computation	General-practice, Web	C
Description	A resultless 0*1? bug in Mozilla. 0*1? loops only produce results in the last iteration, if any. Whether these loops are efficient or not depends on the workload. In this figure, large JavaScript files often fill the script list with tens of thousands of nodes and cause poor performance.			
Source code	1 **for** (sn=script->start, offset=0; 2 !sn; sn=sn->next){ 3 offset += sn->delta; 4 **if** (offset == target) 5 **return** sn; 6 } *//script is a linked list with one node for* 7 *//each byte-code instruction in a JavaScript file*			

Xiao-ISSTA13

BugID: PB65	**PublicationDOI:** 10.1145/2483760.2483784		**Year:** 2013	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1 (first bug)	Execution time	Others (non-trivial operation)	GUI	C++
Description	This code shows a workload-dependent performance bottlenecks that is caused by the method *CPanel::OnRefreshStatusBar*, which contains a non-trivial UI update operation (Line 4). *PB_1* is invoked when a selection-change event is fired.			
Source code	1 **void** CPanel::OnRefreshStatusBar() { // *PB_1* 2 ... 3 GetOperatedItemIndices(indices); 4 _statusBar.SetText(...); // *UI operation* 5 ... } 6 **void** CPanel::GetOperatedItemIndices(CRecordVector<UInt32> &indices) **const** { 7 GetSelectedItemsIndices(indices); 8 ... }			

Selakovic-ICSE15

BugID: PB66	PublicationDOI: 10.1109/ICSE.2015.260			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1-2	Execution time	Others (functional programming)	Language-specific	Javascript
Description	Chalk issue based on functional programming style			
Source code	Chalk issue 28	`return applyStyle._styles.reduce(function ( str, name) { var code = ansiStyles[name]; return code.open + str.replace(code.closeRe, code.open) + code.close; }, str) ;`		

Welton-CCGRID18

BugID: PB67	PublicationDOI: 10.1109/CCGRID.2018.00045			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Execution time	Others (missed parallelization opportunity).	Multiple processes	C
Description	This code from LAMMPs shows characteristics that are bad for GPUs: unknown number of loop iterations, multiple multi-level pointer reads and writes, and a branch condition. Developers infer that multiple costly memory transfers are necessary to transfer the individual data regions pointed to by *v[i]* and *f[i]* to/from the GPU. The memory access pattern will be poor due to non-sequential accesses of data pointed to by *v* and *f* across loop iterations, and that the amount of work may be too small to overcome the overhead of the multiple data transfers.			
Source code	`for (int i = 0; i < nlocal; i++) { if (mask[i] & groupbit) { double dtfm; dtfm = dtf / mass[type[i]]; v[i][0] += dtfm * f[i][0]; v[i][1] += dtfm * f[i][1]; v[i][2] += dtfm * f[i][2]; } }`			

Bornholt-OOSPLA18

BugID: PB68	PublicationDOI: 10.1145/3276519			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Sec. 5.1	Execution time	Others (missed concretization)	Others (solver-aided program)	Rosette
Description	The code shows a missed concretization problem. This implementation is sufficient to verify crash safety at small block sizes (e.g., 32 bytes). But since many crash consistency bugs rely on boundary conditions around the size of disk blocks, with 4kB block sizes even simple tests cannot be verified (or repaired) in reasonable time.			

Source code	`(define N 2)` `(define-symbolic* crash? boolean?)` `(unless crash? ; If not crashed` `  (match-define (file contents length) F)` `  (define new-contents ; write 0x1 to first N bytes` `  (append (make-list N #x01) (drop contents N)))` `  (set! F (file new-contents (+ length N))))`

Bornholt-OOSPLA18

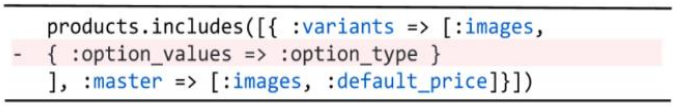
BugID: PB69	PublicationDOI: 10.1145/3276519			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Sec. 5.2	Execution time	Others (missed concretization)	Others (solver-aided program)	Rosette
Description	The problem in this code is caused by a combination of two issues, a missed concretization and an algorithmic mismatch, which manifest as two distinct sources of path explosion. The missed concretization is due to table being a symbolic union, reflecting the table's value along several control-flow paths generated by symbolic execution. The nested use of table thus creates quadratic path explosion. The algorithmic mismatch is due to using filter to create an intermediate list just to sum its contents. The predicate used by filter depends on symbolic state, and so there are O(2^N) paths for the return value of filter. The sum procedure must then run once for each such path.			
Source code	`(map (lambda (t)` `  (sum (filter (lambda (r) (eq? t r)) table)))` `table)`			

Yang-ICSE18

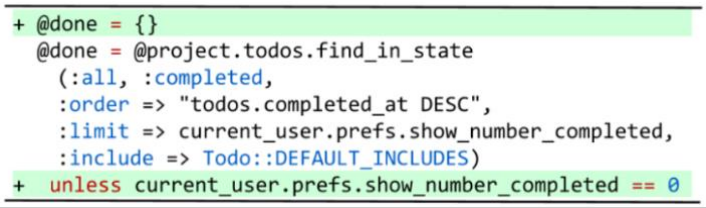
BugID: PB70	PublicationDOI: 10.1145/3180155.3180194			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 7	Execution time	Query	Database, Language-specific, Web	Ruby
Description	This figure shows an example that we find in the latest version of Lobsters, where the deleted code retrieves 50 *mods* objects. Then, for each *mod*, a query is issued to retrieve its associated story. Using eager loading in the added line, all 51 queries (and hence 51 network round-trips) will be combined together. In our experiments, the optimization reduces the end-to-end loading time of the corresponding page from 1.10 seconds to 0.34 seconds.			
Source code	`-  mods = Moderation.limit(50)` `+  mods = Moderation.includes(:story).limit(50)` `  mods.each do |mod|  render mod.story  end`			

Yang-ICSE18

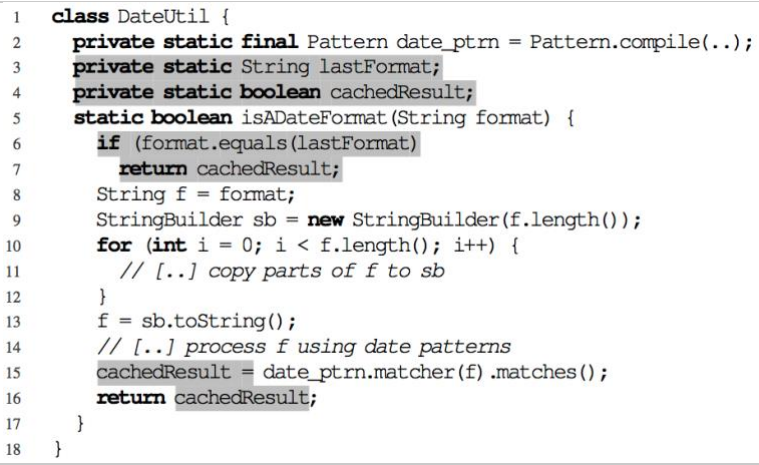
BugID: PB71	PublicationDOI: 10.1145/3180155.3180194			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 8	Execution time	Query	Database, Language-specific, Web	Ruby
Description	Performance bug from Spree. In the user's workload, while loading 405 products to display on the page, eager loading causes 13811 related variants products containing			

	276220 *option_values* to be loaded altogether, making the page freeze. As shown in the figure, the patch delays the loading of option_values fields of variants products.
Source code	

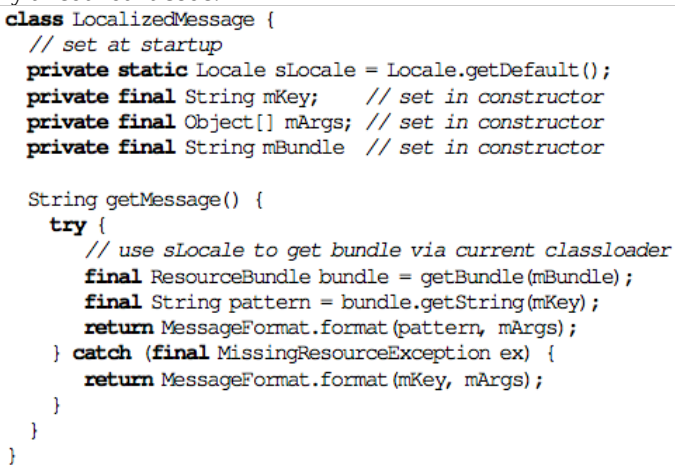
Yang-ICSE18

BugID: PB72	PublicationDOI: 10.1145/3180155.3180194			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 6	Execution time	Query, Unnecessary computation	Database, Web	Ruby
Description	The code from Tracks shows a performance issue due to issuing queries whose results are already known, hence incurring unnecessary network round trips and query processing time. The code originally issues a query to retrieve up to *show_number_completed* completed tasks. When *show_number_completed* is 0, the query always returns an empty set due to limit being 0.			
Source code				

Toffola-OOPSLA15

BugID: PB73	PublicationDOI: 10.1145/2814270.2814290			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Listing 1	Execution time	Redundancy	General-practice	Java
Description	A performance bug in the document converter tool suite Apache POI. The method in the example analyzes a given string and returns a *boolean* that indicates whether the string matches one of several possible date formats. The method gets called frequently in a typical workload, possibly repeating the same computation many times, which slows down the code.			
Source code				

Toffola-OOPSLA15

BugID: PB74	PublicationDOI: 10.1145/2814270.2814290		Year: 2015	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Listing 2	Execution time	Redundancy	General-practice	Java
Description	A performance bug in Checkstyle program. The message string for a particular instance of *LocalizedMessage* is always the same because the message depends only on final fields and on the locale, which does not change while Checkstyle is running. Nevertheless, Checkstyle unnecessarily re-constructs the message at each invocation. This is clearly a redundant code.			
Source code				

```
1  class LocalizedMessage {
2    // set at startup
3    private static Locale sLocale = Locale.getDefault();
4    private final String mKey;     // set in constructor
5    private final Object[] mArgs; // set in constructor
6    private final String mBundle  // set in constructor
7
8    String getMessage() {
9      try {
10       // use sLocale to get bundle via current classloader
11       final ResourceBundle bundle = getBundle(mBundle);
12       final String pattern = bundle.getString(mKey);
13       return MessageFormat.format(pattern, mArgs);
14     } catch (final MissingResourceException ex) {
15       return MessageFormat.format(mKey, mArgs);
16     }
17   }
18 }
```

Liu-ICSE14

BugID: PB75	PublicationDOI: 10.1145/2568225.2568229		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Execution time	Redundancy	GUI	Android (Java)
Description	The first inefficient version conducts two aforementioned operations (Lines 2–9) every time the callback is invoked. The second version applies a “view holder” design pattern suggested by Android documentation. The basic idea is to reuse previously recycled list items. It avoids list item layout inflation when there are recycled items for reuse.			

Source code

```
    //inefficient version
1.  public View getView(int pos, View recycledView, ViewGroup parent) {
2.    //list item layout inflation
3.    View item = mInflater.inflate(R.layout.listItem, null);
4.    //find inner views
5.    TextView txtView = (TextView) item.findViewById(R.id.text);
6.    ImageView imgView = (ImageView) item.findViewById(R.id.icon);
7.    //update inner views
8.    txtView.setText(DATA[pos]);
9.    imgView.setImageBitmap((pos % 2) == 1 ? mIcon1 : mIcon2);
10.   return item;
11. }
12. //apply view holder pattern
13. public View getView(int pos, View recycledView, ViewGroup parent) {
14.   ViewHolder holder;
15.   if(recycledView == null) { //no recycled view to reuse
16.     //list item layout inflation
17.     recycledView = mInflater.inflate(R.layout.listItem, null);
18.     holder = new ViewHolder();
19.     //find inner views and cache their references
20.     holder.text = (TextView) recycledView.findViewById(R.id.text);
21.     holder.icon = (ImageView) recycledView.findViewById(R.id.icon);
22.     recycledView.setTag(holder);
23.   } else {
24.     //reuse the recycled view, retrieve the inner view references
25.     holder = (ViewHolder) recycledView.getTag();
26.   }
27.   //update inner view contents
28.   holder.text.setText(DATA[pos]);
29.   holder.icon.setImageBitmap((pos % 2) == 1 ? mIcon1 : mIcon2);
30.   return recycledView;
31. }
32. //view holder class for caching inner view references
33. public class ViewHolder {
34.   TextView text;
35.   ImageView icon;
36. }
```

Xiao-APLAS15

BugID: PB76	PublicationDOI: 10.1007/978-3-319-26529-2 18			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 10	Execution time	Redundancy	Language-specific	JavaScript
Description	In this case, large volume of closure instances are created and they deoptimize 163 times, where 90.4 % of the deoptimizations are contributed by the field-access site at Line 3 (left).			
Source code				

```
1 this.LINECONTROL[line] =
2 function (parentObj) {
3   if (parentObj.LCDTicks<80) {
4     // ...
5   }
6 }
```

```
1 function entry0(parentObj) {
2   var ticks = parentObj.LCDTicks;
3   processLT143(ticks, parentObj);
4 }
5 function
6 processLT143(ticks, parentObj) {
7   if (ticks < 80) {
8     // ...
9   }
10 }
11 this.LINECONTROL[line] = entry0;
```

Welton-CCGRID18

BugID: PB77	PublicationDOI: 10.1109/CCGRID.2018.00045			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Redundancy	Multiple processes	C
Description	Figure 2 shows an example of an unnecessary transfer from QBox. In the example, QBox is performing a Fourier transform on data starting at location *data[i]* where data is a flat single dimensional array. cuFFT transfers N elements starting at position *data[i]* to the GPU. N is defined by the application on initialization of the FFT library and is stored in the variable plan. The transform is computed on the GPU and the results are transferred back to *data[i]*. Since the values located within data are not modified between each subsequent transform, each transfer after the initial iteration contains duplicated data that does not need to be transferred.			
Source code	```... for(int i = 0; i < n; i++) fftw_execute_dft(plan, &data[i], &data[i]); ...``` A: *Excerpt invoking cuFFT* ```void fftw_execute_dft(plan, in, out){ ... cudaMemcpy(in, dev); [Compute FFT on GPU] cudaMemcpy(dev, out); ... }``` B: *cuFFT library code for function fftw_execute_dft*			

Gu-FSE15

BugID: PB78	PublicationDOI: 10.1145/2786805.2786815			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Execution time	Synchronization	Database, Multiple processes	C
Description	This bug shows how a single-variable atomicity violation was introduced in MySQL791. In one revision, developers decide to update log status to be CLOSED, shown by '+'. This change makes perfect sense for the semantics of the logging thread. However, it could cause the query thread to skip transaction logging, a severe security vulnerability, if the query thread reads log status after it is set to CLOSED and before it is set back to OPEN.			
Source code	```/* Log Thread */ /*log status was OPEN*/ ... //close old log + log_status = CLOSED; ... //open new log log_status = OPEN;``` ```/* Query Thread */ /*log after transaction*/ if(log_status==OPEN){ ... // log update (#) }else{ //Failure:transaction // not logged }```			

Lin-FSE14

BugID: PB79	PublicationDOI: 10.1145/2635868.2635903		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Execution time	Synchronization	GUI	Android (java)
Description	The code shows a synchronization bug from an Android app that displays bus routes and schedules.			
Source code	See below			

```
1  public class RouteselectActivity extends ListActivity {
2      ...
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          ...
6          ListView lv = getListView();
7          final String qry = "select...";
8          final String[] selectargs = {mStopid, datenow,
9              datenow};
10
20
21         Cursor mCsr = DatabaseHelper.ReadableDB()
22             .rawQuery(qry, selectargs);
23         startManagingCursor(mCsr);
24
25
26
27         lv.setOnTouchListener(mGestureListener);
28         if (mCsr.getCount() > 1)
29             tv.setText(R.string.route_fling);
30         else if (mCsr.getCount() == 0)
31             tv.setText(R.string.stop_unused);
32         lv.addFooterView(tv);
33         CursorAdapter adapter =
34             new CursorAdapter(this, mCsr);
35         setListAdapter(adapter);
36     }
37
38 }
```

Liu-ICSE14

BugID: PB80	PublicationDOI: 10.1145/2568225.2568229		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Execution time	Synchronization	GUI, Language-specific, Multiple processes	Android (Java)
Description	Firefox bug 721216 and its bug-fixing patch. This bug caused Firefox to suffer GUI lagging when its “tab strip” button was clicked. The bug occurred because the button’s click event handler transitively called a “refreshThumbnails” method, which produced a thumbnail for each browser tab by iteratively calling heavy-weight Bitmap compression APIs (Lines 3–5). Later to fix this bug, developers moved such heavy operations to a background thread (Lines 6–12), which can asynchronously update Firefox’s GUI.			

Source code	1. public void refreshThumbnails() { 2. //generate a thumbnail for each browser tab 3. - Iterator<Tab> iter = tabs.values().iterator(); 4. - while (iter.hasNext()) 5. - GeckoApp.mAppContext.genThumbnailForTab(iter.next()); 6. + GeckoAppShell.getHandler().post(new Runnable() { 7. + public void run() { 8. + Iterator<Tab> iter = tabs.values().iterator(); 9. + while (iter.hasNext()) 10. + GeckoApp.mAppContext.genThumbnailForTab(iter.next()); 11. + } 12. + }); 13. } Note: the method genThumbnailForTab() compresses a bitmap to produce a thumbnail for a browser tab.

Lin-FSE14

BugID: PB81	PublicationDOI: 10.1145/2635868.2635903			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Execution time	Synchronization	GUI, Multiple processes	Android (java)
Description	This code shows a synchronization bug. At line 3, *onLoadFinished* handler eventually calls *startGroupChat* method, in which an *AsyncTask* is executed. This task writes to field *mRequestedChatId* at line 17. However, this field is read at line 4, which can be executed concurrently with line 17. Thus, there is a race on *mRequestedChatId*.			
Source code	1 class NewChatActivity extends SherlockFragmentActivity { 2 public void onLoadFinished(Loader loader, Cursor cursor){ 3 resolveIntent(); 4 if (mRequestedChatId >= 0) { 5 ... 6 } 7 } 8 private void resolveIntent() { 9 startGroupChat(path, host, listConns.get(0)); 10 ... 11 } 12 private void startGroupChat(...) { 13 ... 14 new AsyncTask<String, Void, String>() { 15 protected String doInBackground(String... params){ 16 ... 17 mRequestedChatId = session.getId(); 18 } 19 }.execute(room, server); 20 } 21 }			

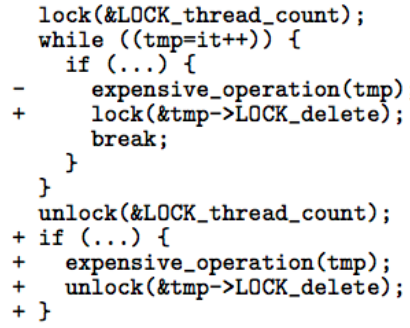
Song-OOPSLA14

BugID: PB82	PublicationDOI: 10.1145/2660193.2660234			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Execution time	Synchronization	Multiple processes	C
Description	Apache performance bug. Users describe that Tomcat could non-deterministically take about five seconds to shut-down, which is usually instantaneous. When applied to Tomcat executions with fast and slow shut-downs, statistical debugging points out that there are strong failure predictors from all three types of predicates: (1) the *if(rc==ETIMEDOUT)* branch on line 5 being taken (branch predicate); (2) the *pthread_cond_timedwait* function returning a positive value (function-return predicate); (3) the value of *rc* on line 3 after the assignment is larger than its original value before the assignment (scalar- pair predicate). These three predicates actually all indicate that *pthread_cond_timedwait* times out without getting any signal. A closer look at that code region shows that developers initialize notified too late. As a result, another thread could set notified to be true and issue a signal even before the notified is initialized to be false on line 1, causing a time-out in *pthread_cond_timedwait*. This problem can be fixed by moving *notified=false*; earlier.				
Source code	1 notified = false; 2 while(!notified) { 3 rc = pthread_cond_timedwait(4 &cond, &lock, &timeToWait); 5 if(rc == ETIMEDOUT) { 6 break; 7 } 8 }			

Gu-FSE15

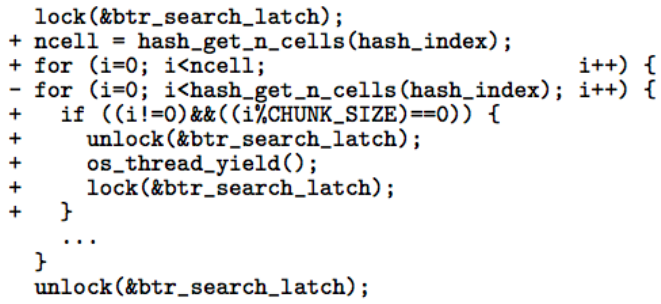
BugID: PB83	PublicationDOI: 10.1145/2786805.2786815			Year: 2015
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Execution time	Synchronization	Multiple processes	C
Description	This buggy revision from HTTrack$_{b20247}$ introduces a new variable accessed by both regions. The lack of synchronization between these two regions leads to concurrency bugs. In this bug, concurrent accesses from two concurrent regions cause a new shared variable to be read before initialization (i.e., order violations);			
Source code	`+ static httrackp *g_opt = NULL;` `/* Main Thread */` `int main() {` `...` `pthread_create(child, ...);` `+ mutexlock(&g_opt->s.l);` `...` `}` `/* Child Thread */` `void child(...) {` `...` `/*Initialize g_opt*/` `+ g_opt = create_opt();` `...` `}`			

Gu-FSE15

BugID: PB84	PublicationDOI: 10.1145/2786805.2786815		Year: 2015	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Execution time	Synchronization	Multiple processes	C++
Description	A performance bug in MySQL$_{r1233}$ with highly contended locks and time-consuming operations in loops. '+' denotes lines added by a revision; '-'denotes lines deleted by a revision.			
Source code				

```
  lock(&LOCK_thread_count);
  while ((tmp=it++)) {
    if (...) {
-     expensive_operation(tmp);
+     lock(&tmp->LOCK_delete);
      break;
    }
  }
  unlock(&LOCK_thread_count);
+ if (...) {
+   expensive_operation(tmp);
+   unlock(&tmp->LOCK_delete);
+ }
```

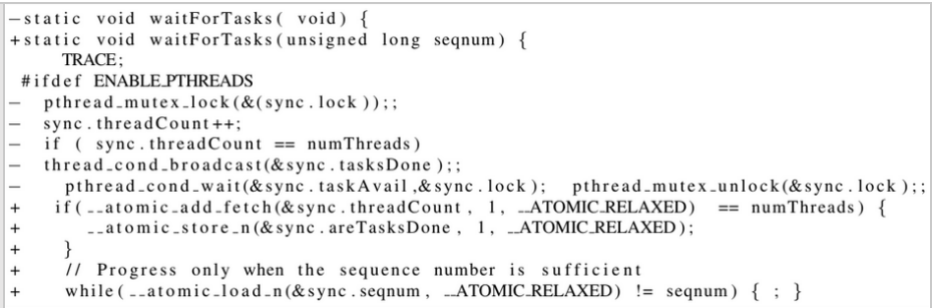
Gu-FSE15

BugID: PB85	PublicationDOI: 10.1145/2786805.2786815		Year: 2015	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Execution time	Synchronization	Multiple processes	C++
Description	A performance bug in MySQL$_{r2121}$ with highly contended locks and time-consuming operations in loops. '+' denotes lines added by a revision; '-'denotes lines deleted by a revision.			
Source code				

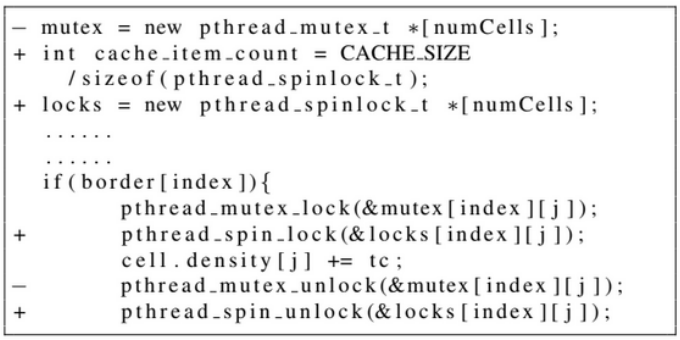
```
  lock(&btr_search_latch);
+ ncell = hash_get_n_cells(hash_index);
+ for (i=0; i<ncell;                              i++) {
- for (i=0; i<hash_get_n_cells(hash_index); i++) {
+   if ((i!=0)&&((i%CHUNK_SIZE)==0)) {
+     unlock(&btr_search_latch);
+     os_thread_yield();
+     lock(&btr_search_latch);
+   }
    ...
  }
  unlock(&btr_search_latch);
```

Alam-EuroSys17

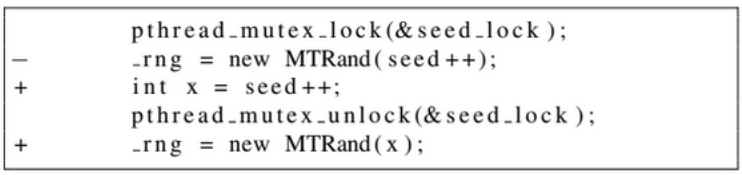
BugID: PB86	PublicationDOI: 10.1145/3064176.3064186		Year: 2017	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Execution time	Synchronization	Multiple processes	C++
Description	The code shows a multiple processes and synchronization bug and its fix. This illustrates one type of locks (ones with the same callsite) has 15288 acquisitions per second but the contention rate is very low.			

Source code	

Alam-EuroSys17

BugID: PB87	PublicationDOI: 10.1145/3064176.3064186		Year: 2017	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 8	Execution time	Synchronization	Multiple processes	C++
Description	The code shows a multiple processes and synchronization bug and its fix. This application uses a two-dimensional array of mutex locks to synchronize concurrent updates of grid data. There are 92K distinct locks, with around 40M acquisitions per second. However, contention rate is almost 0%. In this application, each individual lock has only few thousand acquisitions, but one callsite has a combined acquisitions of 400M.			
Source code				

Alam-EuroSys17

BugID: PB88	PublicationDOI: 10.1145/3064176.3064186		Year: 2017	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 9	Execution time	Synchronization	Multiple processes	C++
Description	The code shows a multiple processes and synchronization bug and its fix. This problem in canneal simulates a cache-aware simulated annealing algorithm to optimize the routing cost of a chip design. Canneal acquires seed lock only 15 times, one for each thread, but lock contention rate is 86%. Also, the total waiting time for this lock is around 0.5 seconds.			
Source code				

Zhang-NOMS14

BugID: PB89	PublicationDOI: 10.1109/NOMS.2014.6838348			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Execution time	Synchronization	Multiple processes, Server	C
Description	Figure shows an example of synchronization code of Apache web server that causes futex kernel events. This is a critical section of inter-thread message passing using shared memory in Apache web server. The producer thread *ap_queue_push* stores a message in the shared queue (*sd* and *p*) and the consumer thread pulls the data in the *ap_queue_pop* function. Two threads are synchronized by using *pthread_mutex_lock* and *pthread_mutex_unlock* calls. In the kernel, this pair of code generate a pair of futex wait and wake events which are used to identify beginning and ending of the slice.			
Source code	`/* producer thread */` `ap_queue_push(fd_queue_t *queue,` `apr_socket_t *sd, apr_pool_t *p) {` `pthread_mutex_lock(&queue->one_big_mutex);` `elem = &queue->data[queue->nelts];` `elem->sd = sd;` `elem->p = p;` `queue->nelts++;` `pthread_mutex_unlock(&queue->one_big_mutex);` `}` `/* consumer thread */` `ap_queue_pop(fd_queue_t *queue` `apr_socket_t **sd, apr_pool_t **p) {` `pthread_mutex_lock(&queue->one_big_mutex);` `elem = &queue->data[--queue->nelts];` `*sd = elem->sd;` `*p = elem->p;` `pthread_mutex_unlock(&queue->one_big_mutex);` `/* caller uses values in sd & p after return */` `}`			

Welton-CCGRID18

BugID: PB90	PublicationDOI: 10.1109/CCGRID.2018.00045			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 5	Execution time	Synchronization, Unnecessary computation	Multiple processes	C
Description	Illustrative example of an early synchronization causing unnecessary delay In the code section shown in the left, a synchronization operation occurs after a memory transfer even though the shared data (stored in *dest*) may not be accessed. If *cond1* is true, the synchronization is unnecessary, blocking the CPU for no reason. In the other cases, there may be enough CPU work performed before the access to *dest* that a delay could be avoided by moving the synchronization closer to the shared data access.			

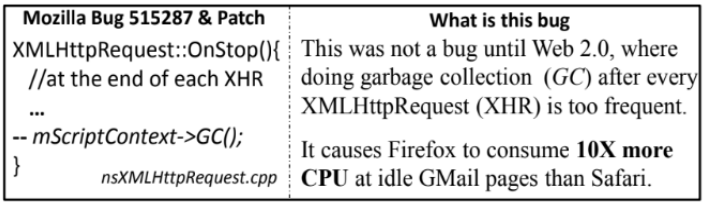
Source code	`cudaMemcpyAsync(dest, src, len, ...);` `...` `cudaDeviceSynchronization();` `v = 1;` `if (cond1) {` `  ... // A lot of CPU computation` `} else if (cond2) {` `  v = dest[0];` `  ... // Any amount of CPU computation` `} else {` `  ... // A lot of CPU computation` `  v = dest[1];` `}` `result = v;`

Song-OOPSLA14

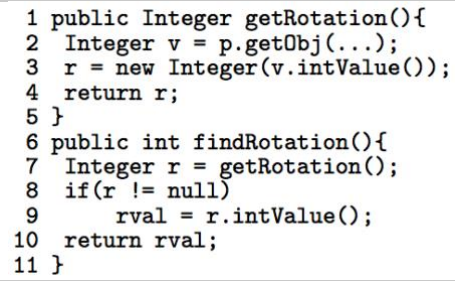
BugID: PB91	PublicationDOI: 10.1145/2660193.2660234			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Execution time	Unnecessary computation	Database	C++
Description	The code illustrates a performance problem reported in MySQL#44723. MySQL44723 is caused by unnecessarily zero-filling the write cache. Users noticed that there is a huge performance difference between inserting 9 rows of data and 11 rows of data. The authors statistical debugging points out that the failure is highly related to taking the (*row* > *MI_MIN_ROWS_TO_USE_WRITE_CACHE*) branch. That is, success runs never take this branch, yet failure runs always take this branch. This is related to the root cause — an inefficient implementation of function *mi_extra*, and the patch makes *mi_extra* more efficient.			
Source code	`1 //ha_myisam.cc` `2 /* don't enable row cache if too few rows */` `3 if (! rows || (rows > MI_MIN_ROWS_TO_USE_WRITE_CACHE) )` `4  mi_extra(...);` `5 //mi_extra() will allocate write cache` `6 //and zero-fill write cache` `7 // fix is to remove zero-fill operation` `8 ....` `9 // in myisamdef.h:` `10 // #define MI_MIN_ROWS_TO_USE_WRITE_CACHE 10`			

Jin-PLDI12

BugID: PB92	PublicationDOI: 10.1145/2254064.2254075			Year: 2012
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Figure 3	Memory	Call to API method	Web	C++
Description	In this bug, Firefox conducted an expensive garbage collection process GC at the end of every *XMLHttpRequest*, which is too frequent. A developer then recalled that GC was added there five years ago when XHRs were infrequent and each XHR replaced substantial portions of the DOM in JavaScript. However, things have changed in modern Web pages. As a primary feature enabling web 2.0, XHRs are much more common than five years ago.			

Source code	

Mudduluru-ISSTA16

BugID: PB93	PublicationDOI: 10.1145/2931037.2931066		Year: 2016	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2b	Memory	Call to API Method, Inter-procedural, Unnecessary computation	General-practice	Java
Description	Code fragment from pdfbox (benchmark). In this code, the method *findRotation* invokes *getRotation* at line 7 which returns a reference to a newly allocated object. There is a dereference at line 3 which is eventually used for an allocation at the same line. This dereference and allocation is unnecessary because *findRotation* can directly invoke *p.getObj(...)* eliminating the need for the allocation at line 3. A client analysis looking for memory issues can identify that all objects created are only used for creating other objects at other point and therefore could indicate a performance bug.			
Source code				

```
1 public Integer getRotation(){
2  Integer v = p.getObj(...);
3  r = new Integer(v.intValue());
4  return r;
5 }
6 public int findRotation(){
7  Integer r = getRotation();
8  if(r != null)
9      rval = r.intValue();
10 return rval;
11 }
```

Jung-ISSM08

BugID: PB94	PublicationDOI: 10.1145/1375634.1375653		Year: 2008	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 15-2	Memory	Condition, Inter-procedural	General-practice	C
Description	This code shows memory leak at line 276. Some heap locations allocated by the procedure *gl_create_context* are not freed by the procedure *gl_destroy_context*.			

Source code	272: if (!osmesa->gl_visual) { 273: return NULL; 274: } 276: osmesa->gl_ctx = gl_create_context(... ... 279: if (!osmesa->gl_ctx) { 280: gl_destroy_visual(osmesa->gl_visual); 281: free(osmesa); 282: return NULL; 283: } 284: osmesa->gl_buffer = gl_create_framebuffer(... 285: if (!osmesa->gl_buffer) { 286: gl_destroy_visual(osmesa->gl_visual); 287: gl_destroy_context(osmesa->gl_ctx); 288: free(osmesa); 289: return NULL; 290: }

Jung-ISSM08

BugID: PB95	**PublicationDOI:** 10.1145/1375634.1375653	**Year:** 2008		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 15-1	Memory	Condition, Missing operation	General-practice	C
Description	This code shows a real-world reported memory leak. From line 261 to 263, the pointer variable *osmesa* points to an allocated heap structure and *osmesa->gl_visual points* to another allocated heap structure. The procedure *gl_create_visual* returns an allocated heap structure. If the procedure *gl_create_visual* returns a null pointer then the current procedure returns the null pointer at line 273 without freeing the heap structure pointed to by *osmesa*.			
Source code	261: osmesa = (OSMesaContext) calloc(1, sizeof(... 262: if (osmesa) { 263: osmesa->gl_visual = gl_create_visual(rgbmode, ... 272: if (!osmesa->gl_visual) { 273: return NULL; 274: }			

Han-ESEM16

BugID: PB96	**PublicationDOI:** 10.1145/2961111.2962602	**Year:** 2016	
Performance bug data			
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Memory	Configuration	Server	C
Description	This code shows a cached not purged bug and its fix. When the proportion of *LDAPSharedCacheSize* to *LDAPCacheEntries* is too small, the old cache entries will not get purged. Hence, the cache hit rate drops rapidly and degrades the system performance. This performance bug requires both *LDAPSharedCacheSize* and *LDAPCacheEntries* set to specific values to manifest.			

Source code	`node = (util_cache_node_t *)util_ald_alloc(cache, sizeof(util_cache_node_t));` `if (node == NULL) {` `- return NULL;` `+ ap_log_error("LDAPSharedCacheSize is too small...");` `+ if (cache->numentries < cache->fullmark){` `+ cache->marktime = apr_time_now();` `+ }` `+ util_ald_cache_purge(cache);` `+ if (node == NULL){` `+ ap_log_error("Could not allocate memory ...");` `+ return NULL;` `+ }` `}`

Jensen-ESEC/FSE15

BugID: PB97	PublicationDOI: 10.1145/2786805.2786860			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 7	Memory	Data structure	Language-specific	Javascript
Description	Performance bug from Esprima, a widely-used JavaScript parser. In this code, the object literal being returned points to an array of length two throughout its lifetime via the *range* field. As this array is always accessed through the returned object, we can reduce bloat in the data structure by inlining this child array into the object. The authors detected a *consistentlyPointedBy* flag that flagged the site as a candidate for object inlining. A subsequent commit by the developers inlined the objects as in the right code fragment of Figure 7, yielding a roughly 10% bottom-line speedup when parsing large JavaScript files.			
Source code	`return {` `type: type,` `value: id,` `lineNumber: lineNumber,` `lineStart: lineStart,` `range: [start, index]` `};` `return {` `type: type,` `value: id,` `lineNumber: lineNumber,` `lineStart: lineStart,` `start: start,` `end: index` `};`			

Jensen-ESEC/FSE15

BugID: PB98	PublicationDOI: 10.1145/2786805.2786860			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Annex-1	Memory	Data structure	Language-specific	Javascript
Description	The authors exercised annex application by playing in 1-person mode (the computer responding to human moves) for three moves. They identified excessive drag. Most stale objects were being allocated within a function used for move evaluation. These objects were accumulated in a global recursive data structure corresponding to computing a minimax game tree. The code uses the above computation to identify the best (highest-value) computer move from possible moves. The same game tree is retained across multiple calls to *evaluate*, but the code could equivalently start with a fresh tree every time, freeing stale memory earlier. The authors applied this refactoring and found a drastic reduction in the peak counts as well as staleness of objects.			

Source code	`ret = possible[0];` `var value = this.evaluate(ret);` `for (var p=1; p<possible.length; p++) {` `  var v = this.evaluate(possible[p]);` `  if (v > value){` `  value = v;` `  ret = possible[p];` `  }` `}`

Xu-JSS18

BugID: PB99	PublicationDOI: 10.1016/j.jss.2017.03.013		Year: 2018	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Case study 1	Memory	Data structure	Others (map-reduce applications)	Java
Description	This case aims to lemmatize the words in Wikipedia through invoking a third-party library named *StanfordLemmatizer*. This OOM error occurs in Line 5 in the following *map()*, while it is processing the 76th record (R76). Each record denotes a line of the Wikipedia (its text has been compacted). The root cause is that *map()* generates large record-level intermediate results during processing R76, and that R76 is a large single record (a super long line). For tagging each word in the line, *lemmatize()* allocates large temporary data structures that might be 3 orders of magnitude larger than the line.			
Source code	`1 public class Mapper {` `2  StanfordLemmatizer slem = new StanfordLemmatizer();` `3  public void map(Long key, Text value) {` `4  String line = value.toString();` `5  for(String word: slem.lemmatize(line)) => 282 MB` `6  emit(word, 1);` `7  }` `8 }` Case Study 1			

Xu-JSS18

BugID: PB100	PublicationDOI: 10.1016/j.jss.2017.03.013		Year: 2018	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Case study 2	Memory	Data structure	Others (map-reduce applications)	Java
Description	This case aims to compute the word co-occurrence matrix of Wikipedia. As shown, *reduce()* allocates a self-defined data structure named *OHMap* to aggregate the records in each <*k*, *list(v)*> group. Each *key* is a word and *value* is an *OHMap* that holds this word's neighboring words. The OOM error occurs in Line 6 of *reduce()*, while it is processing the 779,551st record in the 16th group. The root cause is large group-level accumulated results, which consist of a 372MB *wordMap:OHMap* referenced by memory-consuming method *plus()* (i.e., *plus()* accumulates too many words in the group-level buffer *wordMap*). The number of records in group G16 has 6 times more records than the other groups, which gives rise to data structure expansion.				

Source code	1 public class Reducer { 2 void reduce(Text key, Iterable<OHMap> values) { 3 Iterator<OHMap> iter = values.iterator(); 4 OHMap wordMap = new OHMap(); 5 while (iter.hasNext()) { // for(V value: values) 6 wordMap.plus(iter.next()); => wordMap:OHMap(372MB) 7 } 8 emit(key, wordMap); 9 } 10 } Case Study 2

Bu-ISMM13

BugID: PB101	PublicationDOI: 10.1145/2464157.2466485			Year: 2013
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Listing (page 3)	Memory	Data structure	Others (big data app)	Java
Description	A performance bug from the open-source Big Data computing systems Giraph. This bug produced a number of complaints on OutOfMemoryError from Giraph's user mailing list, and there were 13 bloat-related threads on Giraph's mailing list during the past 8 months. Part of its data representation implementation is shown below. Graphs handled in Giraph are labeled and their edges are directional. Class *EdgeListVertex* represents a graph vertex. Among its fields, *vertexId* and *vertexValue* store the ID and the value of the vertex, respectively. Field *destEdgeIndexList* and *destEdgeValueList* reference, respectively, a list of IDs and a list of values of its outgoing edges. *msgList* contains incoming messages sent to the vertex from the previous iteration.				
Source code	`public abstract class EdgeListVertex<` `I extends WritableComparable,` `V extends Writable,` `E extends Writable, M extends Writable>` `extends MutableVertex<I, V, E, M> {` `private I vertexId = null;` `private V vertexValue = null;` `/** indices of its outgoing edges */` `private List<I> destEdgeIndexList;` `/** values of its outgoing edges */` `private List<E> destEdgeValueList;` `/** incoming messages from` `the previous iteration */` `private List<M> msgList;` `......` `/** return the edge indices starting from 0 */` `public List<I> getEdegeIndexes(){` `...` `}` `}`				

Yang-ICPP12

BugID: PB102	PublicationDOI: 10.1109/ICPP.2012.30			Year: 2012
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 17	Memory	Data structure	System software	C++
Description	A code sample with an array in shared memory with its size determined by a constant variable. The problem with the buggy implementation is that such parameters are declared as variables, which limits the compiler's capability to optimize the code. The project documentation states that based on the application, the constant variable '*KmerSize'* will be an odd number and is less than 33. The default value is 31. In order to improve the performance of the code, the authors leverage the limited value range of '*KmerSize'* by declaring a template so that they can generate multiple kernels with each having a different fixed value of '*KmerSize'*. Then, they replace the shared memory array with registers and the performance improved by 1.93X on GTX480.			
Source code	`__global__ void globalGetErrorPos(){` `  extern __shared__ uchar space[];` `  //KmerSize is from constant mem` `  for (i = 0; i<KmerSize; ++i) {` `  uchar ch = (off != i) ? space[i*TH_PER_TB+tid] : sub; …` `}…}`			

Xu-PLDI10

BugID: PB103	PublicationDOI: 10.1145/1806596.1806616			Year: 2010
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1a	Memory	Data structure, Redundancy	General-practice	Java
Description	An example of an underutilized container, extracted from program JFlex. Method *makeNL* creates a *Vector* object that by default allocates a 10-element array. The only purpose of this object is to pass the three *Interval* objects into the constructor of *RegExpr1*. This *Vector* object is allocated more than 10,000 times during the execution of a large test case. In fact, there are many locations in the code where *Vector* objects are created solely for this purpose.			
Source code	`class CUP$LexParse$actions {` `  RegExp makeNL(){` `  Vector list = new Vector();` `  list.addElement(new Interval('\n','\r'));` `  list.addElement(new Interval('\u0085','\u0085'));` `  list.addElement(new Interval('\u2028','\u2029'));` `  RegExp1 c = new RegExp1(sym.CCLASS, list);` `  ...` `  }` `}`			

Xu-PLDI10

BugID: PB104	PublicationDOI: 10.1145/1806596.1806616			Year: 2010
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1b	Memory	Data structure, Unnecessary computation	General-practice	Java
Description	An example of an overpopulated container, extracted from program Muffin. Given a request, method *cgi* always decodes the entire request string into Hashtable *attrs*. However, this HashMap is later subjected to at most two lookup operations ("config" and "filter"). Instead of decoding and bookkeeping the entire request string, a specialized version of method *cgi* could declare an additional string parameter representing the requested key, and return the corresponding value immediately when the given key is found.			
Source code	`class Httpd extends HttpConnection{` `  Reply recvReply(Request request){` `  if (request.getPath().equals("/admin/enable")){` `  Hashtable attrs = cgi(request);` `  String config = (String) attrs.get("config");` `  String filter = (String) attrs.get("filter");` `  ...` `  }else if(request.getPath().equals("/admin/createConfig")){` `  Hashtable attrs = cgi(request);` `  String config = (String) attrs.get("config");` `  ...` `  } ...` `  }` `  Hashtable cgi(Request request){` `  Hashtable attrs = new Hashtable(13);` `  String query = request.getQueryString();` `  String data = request.getData();` `  if (query != null){` `  StringTokenizer st = new StringTokenizer(decode(query), "&");` `  while (st.hasMoreTokens()){` `  String token = st.nextToken();` `  String key = token.substring(0, token.indexOf('='));` `  String value = token.substring(token.indexOf('=') + 1);` `  attrs.put(key, value);` `  }` `  } ...` `  return attrs;` `  }` `}`			

Xu-PLDI11

BugID: PB105	PublicationDOI: 10.1145/1993498.1993530			Year: 2011
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2	Memory	Inter-procedural	GUI	Java
Description	This code shows a simplified version of a real-world memory leak. Method *runCompare* implements a comparison operation that is invoked every time the comparison option is chosen. *openCompareEditorOnPage* is invoked to open a compare editor in the workspace GUI that shows the differences between two files. The method body is shown at lines 11–17. In this method, the newly created *ResourceCompareInput* object is cached in a *NavigationHistoryInfo* object retrieved from the current workbench page for future save or restore operations (lines 14–16). Caching this heavyweight input object is actually the cause of the leak: the structures			

	of the two files keep being created and referenced. As a result, the memory footprint grows quickly.
Source code	1 void runCompare(ISelection s) { 2 *transaction(s, INFER){* 3 fInput = new ResourceCompareInput (s); 4 visitedFiles.add(new String(s.left)); 5. visitedFiles.add(new String(s.right)); ... 6 openCompareEditorOnPage 7 (fInput, fWorkbenchPage); 8 fInput= null; ***// don't reuse this input!*** 9 *}* 10} 11 static void openCompareEditorOnPage 12 (CompareEditorInput input, 13 IWorkbenchPage page){ ... 14 NavigationHistoryInfo info = 15 page.getNavInfo(); 16 info.add(input); ... 17 }

Xu-QSIC08

BugID: PB106	**PublicationDOI:** 10.1109/QSIC.2008.12		**Year:** 2008	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Memory	Inter-procedural	Language-specific	C
Description	A simplified example from real code. In *malloc_arg1*, the return value 1/0 indicates the success/failure of memory allocation. In *malloc_arg2*, the return value has no relation to the memory allocation. There is a memory leak when the function foo returns at line L2.			
Source code	`int malloc_arg1(int **p) {` `  int *t = malloc(8);` `  if (!t) return 0;` `  else {` `  *p = t;` `  return 1;` `  }` `}` `int malloc_arg2(int **p) {` `  int *t = malloc(8);` `  if (!t)  assert(0);` `  *p = t;` `  if (...)` `  return 0;` `  else` `  return 1;` `}`		`void foo(void) {` `  int r, *p;` `  r = malloc_arg1(&p);` `  if (!r)` `L1:  return;` `  else {` `  // do something to p` `  free(p);` `  }` `  int q = malloc_arg2(&p)` `  if (!q)` `L2:  return;` `  else {` `  // do something to p` `  free(p);` `  }` `}`	

Xu-QSIC08

BugID: PB107	PublicationDOI: 10.1109/QSIC.2008.12			Year: 2008
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 9	Memory	Inter-procedural	Language-specific	C
Description	Memory leak in which *result* gets allocated a heap object at line M. If the execution of program reaches line L, the statement *free(result)* will be skipped, and *result* will get allocated another heap object. The old one is leaked.			
Source code	`// in file which.c:` `int path_search(...) {` `  if (...) {` `  ...` `  do {` `M:  result = find_command_in_path(...);` `  if (result) {` `  ...` `  if (...) {` `  } else if (in_home) {` `  if (skip_tilde) {` `  next = 1;` `L:  continue;` `  }` `  ...` `  }` `  ...` `  }` `  free(result);` `  } while (...)` `  }` `}`			

Xu-QSIC08

BugID: PB108	PublicationDOI: 10.1109/QSIC.2008.12			Year: 2008
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 10	Memory	Inter-procedural	Language-specific	C
Description	Memory leak in wget: *ftp_response()* allocates a heap object and saves the address to the position pointed to by its second argument. So respline is allocated a heap object at line M, but is not freed when function returns at line L.			
Source code	`// in file ftp-basic.c:` `uerr_t ftp_pasv(...) {` `  ...` `M: err = ftp_response (csock, &respline);` `  ...` `  s = respline;` `  for (s += 4; *s && !ISDIGIT (*s); s++);` `  if (!*s)` `L:  return FTPINVPASV;` `  ...` `}`			

Su-LCPC07

BugID: PB109	PublicationDOI: 10.1007/978-3-540-85261-2_16			Year: 2007
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Memory	Inter-procedural, Loop	Communication	C
Description	The code below has a performance bug in it. In the buggy version, each processor only has a pointer to the array on processor 0. *array.restrict(myPart)* returns a pointer to a subsection of array that contains elements from *startIndex* to *end-Index*. Each dereference in the foreach loop results in communication to processor 0 to retrieve the value at that array index. Processor 0 becomes the communication bottleneck as all other processors are retrieving values from it.			
Source code	1 double [1d] array; 2 if (Ti.thisProc() == 0){ 3 array = new double[0:999]; 4 } 5 array = broadcast array from 0; 6 int workload = 1000 / Ti.numProcs(); 7 if (Ti.thisProc() < 1000 % Ti.numProcs()){ 8 workload++; 9 } 10 int startIndex = Ti.thisProc() * workload; 11 int endIndex = startIndex + workload - 1; 12 RectDomain<1> myPart = [startIndex:endIndex]; 13 double [1d] localArray = array.restrict(myPart); 14 double mySum = 0; 15 double sum; 16 17 foreach (p in localArray.domain()) { 18 mySum += localArray[p]; 19 } 20 sum = Reduce.add(mySum, 0);			

Mudduluru-ISSTA16

BugID: PB110	PublicationDOI: 10.1145/2931037.2931066			Year: 2016
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2a	Memory	Inter-procedural, Loop, Unnecessary computation	General-practice	Java
Description	Code fragment from sunflow (benchmark). The object flow profile indicates that the code is a source of creation of a large number of objects. This object flow profile with a large number of objects is indicative of data flow paths that could potentially create memory pressure. On inspecting these hot paths, we understand that the objects created are only used within the loop and their references do not escape to the heap. Therefore, we can optimize this code by reusing objects across iterations.			
Source code	1 public Color getIrradiance(...){ 2 for(Light vpl:vLights[set]){ 3 Ray r1 = new Ray(p, vpl.p); 4 float dotNlD = -(r1.dx*...); 5 float dotND = r1.dx *...; 6 if(dotNlD > 0 && dotND > 0){ 7 float r2 = r1.getMax()...; 8 } 9 }			

Yan-CGO14

BugID: PB111	PublicationDOI: 10.1145/2544137.2544151			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Memory	Loop, Missing operation	General-practice	Java
Description	Figure shows a simple example adapted from the SPECjbb2000 benchmark. *Order* objects are created (line 5) and processed by a *Transaction* (line 6). Each transaction contains *Customers* (lines 12–15), and order processing saves the *Order* in field *curr* of the transaction (line 19) and adds it into one of the *Customers* (line 22). Before an *Order* is processed, the transaction first displays its current order in *this.curr* (lines 26–30), which is set in the previous iteration. At the end of *display*, the current order is removed from *curr* (line 29). While the developer thinks this removal will make the *Order* object unreachable, he/she forgets to clean up references from the *Customer* object. These unnecessary references can lead to a severe memory leak.				
Source code	1 static void main(String[] args) { 2 Transaction t = new Transaction(); 3 for (int i = 0; i < N; i++) { 4 t.display(); 5 Order order = new Order(...); 6 t.process(order); 7 } 8 } 9 class Transaction { 10 Customer[] customers = new Customer[...]; 11 Transaction() { 12 for (int i = 0; i < numCusts; i++) { 13 Customer newCust = new Customer(...); 14 customers[i] = newCust; 15 } 16 } 17 Order curr; 18 void process(Order p) { 19 this.curr = p; 20 Customer[] custs = this.customers; 21 Customer c = custs[p.custId]; 22 c.addOrder(p); 23 ...// process order 24 } 25 void display() { 26 Order o = this.curr; 27 if(o != null) { 28 ... // display o 29 this.curr = null; //remove o 30 } 31 } 32 } 33 class Customer { 34 Order[] orders = new Order[...]; 35 void addOrder(Order y) { 36 Order[] arr = this.orders; 37 arr[...] = y; 38 } 39 }			

Weninger-MPLR19

BugID: PB112	PublicationDOI: 10.1145/3357390.3361025			Year: 2019
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Listing 1	Memory	Loop, Redundancy	Communication, General-practice	Scala
Description	In the loop, a lot of anonymous function objects are created, waiting for an HTTP request to succeed to increment the counter *totalLength*.			
Source code	1 `val response: Future[http.Response] = client(request)` 2 `for (i <- 0 until NUM_REQUESTS) {` 3 `  Await.result(response.onSuccess { rep: http.Response =>` 4 `    totalLength += rep.content.length })}`			

Tonder-ICSE18

BugID: PB113	PublicationDOI: 10.1145/3180155.3180250			Year: 2018
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 1	Memory	Missing operation	General-practice	PHP
Description	This code shows a memory leak. Memory is allocated in line 2 and if the function ends with the return of line 11 it is not released.			
Source code	1 `swHashMap *hmap =` 2 `  sw_malloc(sizeof(swHashMap));` 3 `if (!hmap) {` 4 `  swWarn("malloc[1] failed.");` 5 `  return NULL;` 6 `}` 7 `swHashMap_node *root =` 8 `  sw_malloc(sizeof(swHashMap_node));` 9 `if (!root) {` 10 `  swWarn("malloc[2] failed.");` 11 `  return NULL; // returns, hmap not freed` 12 `}`			

Kothari-ICSMC14

BugID: PB114	PublicationDOI: 10.1109/SMC.2014.6974210			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Memory	Missing operation	Language-specific	C
Description	Function *dswrite*() : writes a block onto a disk device. The highlighted line corresponds to memory allocation. The analyst asks T to show the execution paths where memory is not being freed. T responds with the call graph of function *dswrite*() restricted to the current memory leak scenario.			

Source code	`dswrite(devptr, buff, block)` `struct devsw *devptr;char *buff;DBADDR block; {` `struct dreq *drptr;` `char ps;` `disable(ps);` `drptr = (struct dreq *) getbuf(dskrbp);` `drptr->drbuff = buff;` `drptr->drdba = block;` `drptr->drpid = currpid;` `drptr->drop = DWRITE;` `dskenq(drptr, devptr->dvioblk);` `restore(ps);` `return (OK);` `}`

Lee-ICSE14

BugID: PB115	PublicationDOI: 10.1145/2568225.2568307	Year: 2014		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 11	Memory	Missing operation	Language-specific	C
Description	Bash-4.0 has a memory leak in the read built-in when the number of fields read from an input is not the same as the number of variables passed as arguments to the built-in. At line 5, a temporary memory object is created inside *get_word_from* string, and input string is advanced to the next character in the original input string to check for the end. If the next character is null, the reference to the temporary object is stored in *tofree* (line 8), and the object gets deallocated at line 27. However, if the next word is not null (line 9), the temporary object *t* gets lost.			
Source code	`1 tofree = NULL;` `2 if (*input_string)` `3 {` `4 t1 = input_string;` `5 t = get_word_from_string (` `6 &input_string, ...);` `7 if (*input_string == 0)` `8 tofree = input_string = t;` `9 else {` `10 input_string =` `11 strip_trailing_ifs_whitespace (` `12 t1, ...);` `13 }` `14 }` `15` `16 if (saw_escape)` `17 {` `18 t = dequote_string (` `19 input_string);` `20 var = bind_read_variable (...);` `21 xfree (t);` `22 }` `23 else` `24 var = bind_read_variable (` `25 ..., input_string);` `26 ...` `27 FREE (tofree);`			

Fehnker-ISSE13

BugID: PB116	PublicationDOI: 10.1007/s11334-012-0192-5			Year: 2013
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Table 3	Memory	Missing operation	Language-specific	C++
Description	This code shows a warning for using a non-virtual destructor in an abstract class in firefox. The problem with this construct is that even if a subclass defines its own destructor, it will use the destructor of the abstract class. This might have unexpected consequences, that may include memory leaks.			
Source code	`/*file /obj/dist/include/ xpcom/nsWeakReference.h */` `90 inline` `91 nsSupportsWeakReference:: ~nsSupportsWeakReference()` `92 {` `-> 93 ClearWeakReferences();` `94 }`			

Jensen-ESEC/FSE15

BugID: PB117	PublicationDOI: 10.1145/2786805.2786860			Year: 2014
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 3	Memory	Missing operation	Web	Javascript
Description	This code represents a small jQuery-based example, based on a real drag issue found in a shopping list case study. In function *f()*, a new div DOM node is allocated (into variable *newDiv*) (line 2), made to contain the text "Hello world" (line 3), and attached to an existing DOM node identified by the ID 'contents' (line 7). An event handler function is attached to the newly allocated node (lines 4–6), which changes the background color when a user clicks on the div. In function *g()*, the developer removes this new div from the DOM by assigning an empty string to the *innerHTML* property of the 'contents' node (line 11). Alas, there is a memory problem. The jQuery framework caches event handlers associated with DOM nodes, and clears the cache only when a node is removed using a proper jQuery API call, e.g. (in this case) *$('#contents').empty()*. Since the developer removed the node via *innerHTML* instead, the event handler closure remains in jQuery's cache and becomes stale, as it does not get used again. Since the event handler uses variable *newDiv* (line 5), a reference to the div element remains in its closure, causing the DOM element itself to also drag; this is known in the developer community as leaking of DOM nodes, as it a common issue in web applications.				
Source code	`1 function f() {` `2 var newDiv = $('<div/>');` `3 newDiv.html("Hello world");` `4 newDiv.click(function () {` `5 newDiv.css("backgroundColor", "red");` `6 });` `7 newDiv.appendTo('#contents');` `8 }` `9` `10 function g() {` `11 document.getElementById('contents').innerHTML = '';` `12 }`			

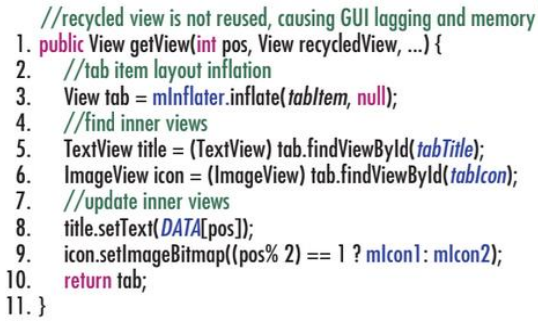
Jensen-ESEC/FSE15

BugID: PB118	PublicationDOI: 10.1145/2786805.2786860	Year: 2014		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Shopping list	Memory	Missing operation	Web	Javascript
Description	Shopping List is a Tizen web application. For this app, were showed leaking objects and DOM nodes deep inside the jQuery library. Call stacks at the allocation points of the leaking objects showed that they were allocated due to application calls for attaching event handlers. Access paths showed that the objects remained reachable due to pointers from an internal jQuery cache. The cause of the leak was that the code attempted to clear a DOM element as shown below. The correct way when using jQuery would be to use jQuery construct *$(ShoppingListApp.listoflists).empty();* in place of an assignment to *innerHTML*. This changes fixed the leak, and a patch was accepted by the developer.			
Source code	`if (self.currentView.resetListOfLists) {` `  ShoppingListApp.listoflists.innerHTML = "";` `}`			

Xu-JSS18

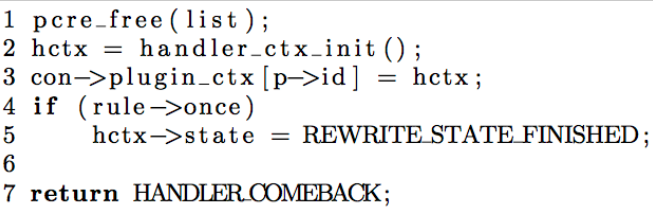
BugID: PB119	PublicationDOI: 10.1016/j.jss.2017.03.013	Year: 2018		
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Case study 3	Memory	Query	Others (map-reduce applications)	SQL—like script
Description	This job is automatically generated by a SQL-like Pig script, which tries to count the number of distinct values in column *pageurl* for each distinct *pagerank* in *tableA*. This script first aggregates the tuples using *GroupBy(pagerank)*. Then, in each group, it uses *count(distinct pageurl*) to deduplicate the tuples that have the same value of *pageurl*. The OOM error occurs in *combine()*, while it is processing the 247,932nd record in 8th group (G8). There is a continuous growth in group G8 and G8 has much more records than the other groups. The root cause is that *combine()* generates large group-level accumulated results during processing 247,932 records in G8 (i.e., a group-level buffer holds too many distinct tuples that have the same *pagerank*). To identify the high-level memory-consuming operator, we further refer to the Pig manual DIS and identify that the *DISTINCT* is the cause. This operator allocates a data structure to accumulate all the input records to sort and deduplicate them.			
Source code	`1 pTable = LOAD tableA as (pagerank, pageurl, duration);` `2 rankTable = GROUP pTable BY pagerank;` `3 urlTable = FOREACH rankTable {` `4  urls = DISTINCT urlTable.pageurl;` `5  GENERATE group, COUNT(urls), SUM(pTable.duration);` `6 };` `7 STORE urlTable into "/output/newTable";  Case Study 3`			

Liu-IEEESoftware15

BugID: PB120	PublicationDOI: 10.1109/MS.2015.4		Year: 2015	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 2c	Memory	Redundancy	GUI	Android
Description	The code shows that item *inflation* (line 3) and inner view update (lines 5–9) occur each time *getView()* is invoked. This hurts the tab tray's scrolling performance. This implementation also fails to reuse items and consumes much memory by continuously inflating list items. Owing to the resulting memory pressure, the garbage collector will run frequently, further degrading the whole system's performance.			
Source code				

```
//recycled view is not reused, causing GUI lagging and memory bloat
1. public View getView(int pos, View recycledView, ...) {
2.    //tab item layout inflation
3.    View tab = mInflater.inflate(tabItem, null);
4.    //find inner views
5.    TextView title = (TextView) tab.findViewById(tabTitle);
6.    ImageView icon = (ImageView) tab.findViewById(tabIcon);
7.    //update inner views
8.    title.setText(DATA[pos]);
9.    icon.setImageBitmap((pos% 2) == 1 ? mIcon1: mIcon2);
10.   return tab;
11. }
```

Lee-ICSE14

BugID: PB121	PublicationDOI: 10.1145/2568225.2568307		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 10	Memory	Redundancy	Language-specific	C
Description	Memory leak in lighttpd *mod_rewrite* method. At line 2, *mod_rewrite* creates a memory object that stores current context information. However, this code region can be visited multiple times during a processing of a request if *url.rewrite-repeat* rule intervenes. When visited again, *mod_rewrite* creates another context object that is redundant at line 2, and overwrites reference to the old context object by a reference to the newly created one. Since the reference to the old context object is lost by the rewriting, this results in a memory leak.			
Source code				

```
1 pcre_free(list);
2 hctx = handler_ctx_init();
3 con->plugin_ctx[p->id] = hctx;
4 if (rule->once)
5     hctx->state = REWRITE_STATE_FINISHED;
6
7 return HANDLER_COMEBACK;
```

Jensen-ESEC/FSE15

BugID: PB122	PublicationDOI: 10.1145/2786805.2786860		Year: 2014	
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Annex-2	Memory	Redundancy	Language-specific	Javascript
Description	The object assigned to *mtable* was identical for each call to *getValue*, and these allocations were causing significant unnecessary churn. The issues could be fixed by replacing the allocations with a single global object. Performing this transformation			

	led to a 10% bottom-line speedup for computer move computation, and the fix was accepted by the developer.
Source code	`getValue: function(place){` `  var i = parseInt(place[0]);` `  var j = parseInt(place[1]);` `  var mtable = {` `  0:{ 0:100, 1:-50, 2:40, 3:30, 4:30, 5:40, ...},` `  1:{ 0:-50, 1:-30, 2:5, 3:1, 4:1, 5:5, ...},` `  ...` `  };` `  return parseInt(mtable[i][j]);` `}`

Yang-ICPP12

BugID: PB123	**PublicationDOI: 10.1109/ICPP.2012.30**			**Year: 2012**
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 4	Memory	Redundancy, Synchronization	System software	C
Description	Buggy code in Kernel invocation with a thread block dimension of 16x16. The problem is that the fixed block dimension like 16x16 does not consider how the data are accessed and reused and therefore is not the optimal choice. The authors examine three different data access and reuse patterns. The first is no data reuse among threads. The second is reuse through (software-managed) shared memory and the third is reuse through hardware-managed caches. Among them, the effect of thread block dimensions on shared memory usage has been observed before and the other two, to our knowledge, have not been reported in the literature.			
Source code	`int main() {` `  dim3 **blkDim(16, 16)**; // Kernel invocation` `  dim3 gridDim(N / blkDim.x, N / blkDim.y);` `  myKernel<<<gridDim, **blkDim**>>>(…); }`			

Yang-ICPP12

BugID: PB124	**PublicationDOI: 10.1109/ICPP.2012.30**			**Year: 2012**
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 6	Memory	Redundancy, Synchronization	System software	C
Description	The tiled matrix multiplication based on the sample code in NVIDIA CUDA SDK. We can see that after the tiles from both *A* and *B* are loaded into the shared memory, they will be used by all the threads in the thread block. Based on the access, *As[ty, k]*, we can see that each element in the tile from matrix *A* is reused by the threads with the same *tx (i.e., thireadIdx.x*). Similarly, each element in the tile from matrix *B* is reused by the threads with the same *ty*, as a result of the access *Bs[k, tx]*. For block dimension of 16x16, each element in either tile is reused 16 times. However, when the block dimension is changed to 32x8, the tile from matrix *A* is of size 32x8 (*As[8][32])* and is reused 32 times while the tile from matrix *B* is of size 32x32 (*Bs[32][32*]) and is used 8 times. Hence, with the block dimension of 32x32, the high data reuse leads to the best energy efficiency although the performance is suboptimal. The block dimension of 16x16, in comparison, has the lowest energy efficiency.				

Source code	__global__ void matrixMul(output* C, input* A, input* B, intwA, in wB){ int tx = threadIdx.x; int ty = threadIdx.y; ….//variable declaration and definition for (a = aBegin, b = bBegin;a <= aEnd;a+=aStep, b+=bStep){ __shared__ float As[blocky][blockx], Bs[blockx][blockx]; …//load a tile of A into shared mem As …//load a tile of B into shared mem Bs for(i= 0; i<Step; i++) Csub += As[ty][i]* Bs[i][tx]; } …//store Csub to matrix C }

Yang-ICPP12

BugID: PB125	PublicationDOI: 10.1109/ICPP.2012.30			Year: 2012
Performance bug data				
Figure	**Effect**	**Cause**	**Context**	**Language**
Fig. 8	Memory	Redundancy, Synchronization	System software	C++
Description	Code fragment for the activation matrix kernel from GPUMLib. The authors show that block dimensions have a significant impact on how effective the cache is utilized. They use the activation matrix kernel from the GPUMLib. For the kernel shown in the code, the authors make no modification and only vary the block dimensions. The GTX480 GPU is configured to have 48kB L1 cache for each SM as the kernel does not use any shared memory and relies on the cache for data reuse. They first keep the same thread block size (256) in a thread block and vary the dimensions from 4x64 to 32x8. Then, they increase the thread block size and use the dimensions of 8x64.			
Source code	__global__ void ActivationMatrix(output *Output, input *A, input *B, intwA, intwB){ … int idnx = blockIdx.x*blockDim.x + threadIdx.x; int idny = blockIdx.y*blockDim.y + threadIdx.y; for(i=0; i<wA;,i++){ a = A[idnx * wA + i]; b = B[idny * wB + i]; temp += pow(a - b , 2); } …}			