JinsightLive for IBM System z : Binary File Format

A brief explanation of the JinsightLive binary file format to aid understanding and problem determination.

Highlights

- The JinsightLive binary file format is simple, robust and surprisingly flexible. Although designed to describe Java code activity, other languages could be massaged into this format and therefore the visualizer re-tasked.
- After a brief header, the rest of the file contains discrete events, there are only a few event types of interest: Class Define, Method Entry/Exit, Object Alloc/Free, Thread Start/Stop and GC activity.
- Assets need to be described before they are called upon. For example, before a method entry event can be processed, the Class that contains that method has to be described, as well as the Object instance called.
- The unique and highly efficient JinsightLive “Execution View” visualization has never been applied to other function call based languages.

Overview

JinsightLive uses a simple file and wire format, it describes server side JVMPI events plainly, is robust and somewhat extensible. It does have some quirks that need to be accounted for but they are not too much of a hindrance.

The Header

```
struct Header {
   uint8_t binary ;
   uint32_t version;
   uint32_t platform;
   uint32_t numberOfEvents;
   uint32_t maxThreads;
   uint32_t maxClasses;
   uint32_t ticksPerMicrosecond;
   uint64_t startTicks;
   uint32_t vmStartTime;
   uint32_t connectionStartTime;
   uint32_t overhead;
};
```

Fig 1: The JinsightLive header.

The 45 byte header is always little endian figure 1 shows us the form. The first byte would be ASCII 'b' or 0x62 and indicates that this is a binary file. Next we have a 4 byte int for the format version, it should be fixed to 0x08. The next 4 byte int is the platform that the trace was taken on, this is chiefly used to decide the endianism of the rest of the trace after the header. The choices for platform are shown in figure 2.

```
#define JINSIGHT_WIN32x86    11
#define JINSIGHT_AIXPPC      22
```

#define JINSIGHT_OS390 33
#define JINSIGHT_LINUX_x86 44
#define JINSIGHT_LINUX_390 55

Fig 2: JinsightLive platforms.

The following 4 byte int is the number of events in the trace file, this can be 0 as the client does not rely too heavily on it. But if you wish, at the end of a collection you can seek back and write the number of collected events into this position. Following this there are two 4 byte ints that define the maximum number of classes and threads defined in the trace, the JinsightLive client uses these numbers to initialise internal structures. The client will add 20,000 to maxClasses and take the result of Math.min(256, maxThreads) for maxThreads.

The ticksPerMicrosecond is a relatively important value to get right as JinsightLive uses it to calculate most of its timing information. The ticks in question here are of course the values reported by the hardware clock on the platform in question. On Intel this would be rdtsc, on zSeries STCK, etc. startTicks should be set to the HW ticks when the profile started. vmStartTime and connectionStartTime are in epoch seconds and can be set to the same value, for example again the time the profile started. Finally we have a 4 byte int overhead value that can be used to tell the client how many ticks it takes to invoke the profiler to record an event crossing the JNI barrier for example, this can also be set to 0. As mentioned before this header is always little endian but everything that follows is encoded in the defined platforms endianism.

The Events

The events follow the header directly, the general form is a 1 byte event ID followed by an event specific structure. As some events contain strings they can be variable in size. The Major events that we are interested in listed in figure 3 with their IDs.

#define CLASS_DEFINE 0x04
#define EXTENDED_EXTENSIVE_CLASS_LOAD 0x6e
#define WIDE_METHOD_ENTER 05b
#define METHOD_LEAVE 0x1f
#define OBJECT_DEFINED 0x14
#define OBJECT_CREATED 0x15
#define OBJECT_RECLAIMED 0x18
#define ARRAY_DEFINED 0x16
#define ARRAY_CLASS_DEFINED 0x05
#define ARRAY_CREATED 0x17
#define THREAD_DEFINE 0x0a
#define THREAD_START 0x0b
#define THREAD_EXIT 0x0c
#define GC_START 0x46
#define GC_STOP 0x47
#define USER_EVENT 0x4e
#define BEGIN_BURST 0x65
#define END_BURST 0x66

Fig 3: The major events.

You will notice the _DEFINED and _CREATED events, the distinction here is that if the JinsightLive profiling agent begins collecting after the JVM has executed for some time, then it may see, for example, a method entry on an object that it did no "see" being created. In this case it will write an OBJECT_DEFINED just before writing the WIDE_METHOD_ENTER. But if the object is created as JinsightLive is profiling it would write both the _DEFINED and _CREATED events. The instance numbering scheme required by the JinsightLive client should also be noted. JVMPI provides ID for classes, methods, objects and threads, these ID's are simply memory addresses. JinsightLive on the other hand requires a linear numbering scheme starting from 1 for these

assets. Figure 4 is an illustration of how JVMPI events can map to JinsightLive events. The first and most involved events are to do with class loading. It is best to always write and CLASS_DEFINE event followed by an EXTENDED_EXTENSIVE_CLASS_LOAD event (from now on referred to as _CLASS_LOAD).

JVMI_EVENT_CLASS_LOAD :
CLASS_DEFINE
EXTENDED_EXTENSIVE_CLASS_LOAD
JVMPI_EVENT_METHOD_ENTRY2 : WIDE_METHOD_ENTER
JVMPI_EVENT_METHOD_EXIT : METHOD_LEAVE
JVMPI_EVENT_OBJECT_ALLOC :
OBJECT_DEFINED
OBJECT_CREATED
ARRAY_DEFINED
ARRAY_CLASS_DEFINED
ARRAY_CREATED

JVMPI_OBJECT_FREE : OBJECT_RECLAIMED

JVMPI_EVENT_THREAD_START :
THREAD_DEFINE
THREAD_START

JVMPI_EVENT_THREAD_EXIT : THREAD_EXIT

JVMPI_EVENT_GC_START : GC_START

JVMPI_EVENT_GC_STOP : GC_STOP

Fig 4: JVMPI to JinsightLive mapping

```
struct ClassDefine {
  uint8_t eventType;
  uint64_t ticks;
  uint16_t classID; // generated
  uint32_t classObjectID;
};
```

Fig 5: CLASS_DEFINE

Figure 5 describes CLASS_DEFINE. It is fixed length, the classID is a unique identifier created by the profiler from a linear numbering scheme starting from 1; classObjectID can be set to 0. Immediately following the CLASS_DEFINE you should write a _CLASS_LOAD event, which describes the actual class itself. The JinsightLive client then automatically associates the ID set in CLASS_DEFINE with description on _CLASS_LOAD; these two events should be considered to be one. _CLASS_LOAD is described in figure 6, it is of variable length due to the various strings it contains.

```
struct ExtendedExtensiveClassLoad {
  uint8_t eventType;
  uint64_t ticks;
  uint16_t classID; // same as define
  uint16_t size;
  uint16_t classNameLength;
  char[] className;
```

```
  uint16_t numberOfMethods; {
    uint16_t methodNameLength;
    char[] methodName;
    uint16_t methodSignatureLength;
    char[] methodSignature;
    uint16_t accessFlags;
  } // repeated numberOfMethods times

  uint16_t numberOfFields; {
    uint16_t fieldNameLength;
    char[] fieldName;
    uint16_t fieldSignatureLength;
    char[] fieldSignature;
    uint16_t accessFlags;
  }  // repeated numberOfFields times

  uint16_t superClassID; // using the Jinsight ID

  uint16_t numberOfInterfaces; {
    uint16_t interfaceIDs; // using Jisnight IDs
  }  // repeated numberOfInterfaces times
};
```

Fig 6: *_CLASS_LOAD*

The strings defined take the form of a uint16_t length value followed by length bytes containing the string itself, there is no null termination. Class definition is the area of greatest interest when attempting to squeeze a profile from another language into the JinsightLive file format. The class loads define a name space that the JinsightLive client will use for many things. Experimentation should yield the most effective mapping from the source language name space to Java/Jinsight.
As mentioned before, class definitions have to be written before they can be referred to by other events.

The next event types are related to creating and collecting objects. The form of these events maps very directly to Java and the JVMPI interface. If writing a profile from another language you may decide to skip these events all together and write all method entry/exits as static. As with other event types there is a distinction between _DEFINE and _CREATE. In the case of Objects you may wish to only write the _DEFINE event if an existing object that the profiler has not seen before is referred to by another event such as method entry. If in doubt it is safer to write a _DEFINE followed by a _CREATE.

In Java/JinsightLive there are 3 object "types", normal objects, primitive arrays and arrays of objects; the sequence of events written depends on the type.

For normal objects you write;

OBJECT_DEFINED
OBJECT_CREATED

For primitive arrays;

ARRAY_DEFINED
ARRAY_CREATED

And For Arrays of Objects;

ARRAY_CLASS_DEFINED
ARRAY_CREATED

These are all fixed length events and the object IDs written within them are again generated linearly by the profiler. For primitive arrays JinsightLive uses its own list of types show in figure 7.

```
#define int_array 1
#define long_array 2
#define float_array 3
#define double_array 4
#define boolean_array 5
#define byte_array 6
#define char_array 7
#define short_array 8
```

Fig 7: JinsightLive primitive array types.

```
struct ObjectDefined {
   uint8_t eventType;
   uint64_t ticks;
   uint32_t objectID; // generated
   uint16_t classID;
};
```

Fig 8: OBJECT_DEFINED.

```
struct ObjectCreated {
   uint8_t eventType;
   uint64_t ticks;
   uint32_t objectID; // same as define
   int16_t classID;
   uint32_t threadID;
};
```

Fig 9: OBJECT_CREATED.

```
struct ArrayDefined {
   uint8_t eventType ;
   uint64_t ticks;
   uint8_t arrayType;
   uint32_t objectID; // generated
   uint32_t size;
};
```

Fig 10: ARRAY_DEFINED.

```
struct ArrayClassDefined {
   uint8_t eventType ;
   uint64_t ticks;
   uint32_t objectID; // generated
   uint16_t classID;
   uint32_t size;
};
```

Fig. 11: ARRAY_CLASS_DEFINED.

```
struct ArrayCreated {
   uint8_t eventType;
   uint64_t ticks;
   uint8_t isArray;
   uint32_t objectID; // same as define
   uint32_t size;
   uint32_t threadID;
};
```

Fig 12: ARRAY_CREATED.

```
struct ObjectReclaimed {
   uint8_t eventType;
   uint64_t ticks;
   uint32_t objectID;
};
```

Fig 13: OBJECT_RECLAIMED

Figures 8, 9 10, 11 and 12 show the object alloc related event structures. Figure 13 is the OBJECT_RECLAIMED event structure. In figure 12 you will notice the single byte isArray entry, this is set currently set to the same value provided by the JVMPI OBJECT_ALLOC event, see jvmpi.h for details.

The next events deal with threads, again we have the _DEFINE _CREATE distinction. In the case of the JinsightLive profiling agent it may start profiling during the life of the JVM and would have not been notified of the start of any of the running threads. Therefore the first time it sees a thread it is unaware of, on an object alloc for instance, it will first write a THREAD_DEFINE event. If it was informed of a real thread start by JVMPI it would write a THREAD_DEFINE followed by a THREAD_START, but at a minimum you can get away with just a _DEFINE, naturally choosing a unique ID for this thread. Figures 14,15 and 16 describe these thread events.

```
struct ThreadDefine {
   uint8_t eventType;
   uint64_t ticks;
   uint32_t threadID; // generated
   uint32_t objectID;
   uint16_t classID;
   uint16_t threadNameLength;
   char[] threadName;
};
```

Fig 14: THREAD_DEFINE

```
struct ThreadStart {
   uint8_t eventType;
   uint64_t ticks;
   uint32_t threadID; // same as define
};
```

Fig 15: THREAD_START

```
struct ThreadEnd {
   uint8_t eventType;
   uint64_t ticks;
   uint32_t threadID;
};
```

Fig 16: THREAD_END

The bulk of the events in a JinsightLive trace file are method entry and exit. The two events types in question here are WIDE_METHOD_ENTER and METHOD_LEAVE. The thread, class and objects IDs that these events refer to must have already been written. In the class definition event the methods are not explicitly but rather implicitly numbered, the first method defined is ID 0, the next 1 etc. So the client expects the method ID passed to be in the range 0 to n, where n is the number of methods in the class. JVMPI provides only its own methodID, so on the server side the JinsightLive agent holds a lookup table from JVMPI method ID to JinsightLive class and method IDs. The entry and exit events are shown in figures 17 and 18.

```
struct WideMethodEnter {
   uint8_t eventType ;
   uint64_t ticks;
   uint32_t threadID;
   uint16_t classID;
   uint16_t methodID;
   uint32_t ObjectID;
   uint16_t lineNumber;
};
```

Fig 17: WIDE_METHOD_ENTER

```
struct MethodLeave {
   uint8_t eventType ;
   uint64_t ticks;
   uint64_t overhead;
   uint32_t threadID;
};
```

Fig 18: METHOD_LEAVE

If you are attempting at utilise the JinsightLive visualisation client to display the profile of a language other than Java you may find that there is no obvious analogous concept to Objects. In this case you can simply ignore writing object related events and set the objectID on method entry to 0xFFFFFFFF (-1). This indicates to the client that it is a static call.

The overhead value in MethodLeave allows the profiler to indicate to the client the overhead in ticks that this particular entry/exit sequence incurred. This is not the overhead for the child calls. Internally JinsightLive maintains a stack, sets the entry overhead on entry and adds the exit overhead to this and writes the exit event as each level of the stack unwinds.

There are three other events that may be of interest. Firstly there are the GC_START and GC_STOP events, which are self-evident and described in figures 19 and 20.

```
struct GCStart {
   uint8_t eventType ;
   uint64_t ticks;
};
```

Fig 19: GC_START

```
struct GCStop {
   uint8_t eventType ;
   uint64_t ticks;
};
```

Fig 20: GC_STOP

Finally there is a user defined event type, which will take any text and overlay in on the visualized trace; this can be seen in Figure 20.

```
struct UserEvent {
   uint8_t eventType ;
   uint64_t ticks;
   uint16_t messageLength;
   char[] message;
};
```

Fig 21: USER_EVENT

There are numerous client quirks that need to be worked around. One of the most significant is its hard coded expectation to see java/lang/Object at class ID 0 and java/lang/Thread at class ID 1.

Bursts

The simplest JinsightLive trace file would contain a single sequence of execution. However, with the concepts of bursts, a single file can comprise multiple profile sequences from the same profiling session. For example you may wish to externally control when profile data is collected, skip start up execution, profile for a few seconds, skip the next 30 seconds and then profile for a few more seconds.

The concept of bursts within JinsightLive supports this behavior and allows many separate profile sequences to be encapsulated within a single file. Events are wrapped in BEGIN_BURST and END_BURST events. There only needs to be one header, but once bursts have been defined, events should not appear in the file outside of a burst.

```
struct BeginBurst {
   uint8_t eventType ;
   uint64_t ticks;
};
```

Fig 22: BEGIN_BURST

```
struct EndBurst {
   uint8_t eventType ;
   uint64_t ticks;
};
```

Fig 23: END_BURST

Conclusion

Experimentation is the best strategy, creating a minimal trace file with one thread define, one class define/load, one object define/created and one method entry exit. From there, one can iterate through creating more complex traces.

The client has a useful command line option -DVERBOSE_LOAD which prints in detail the event load process allowing format errors to be trapped and corrected .

This document more than likely contains mistakes and may not at the moment cover all events required or all nuances of the client .

Any questions should be sent to paul-anderson@nl.ibm.com so that they can be answered and this document updated.