The mysteries of Gooda
Internal operation description.
This is and will always be a work in progress

## Design Criteria

Gooda is designed for speed and modularity. It consists of three parts, data collection using perf record and predefined scripts with events lists and correct sampling periods, a data file reader and analyzer that creates fixed tables at a variety of SW entity granularities (process, module, function, asm, source etc) and a web based display written in java script.

## Data Structures

The data structures needed to process performance profiling data naturally form a tree hierarchy. There are two dominant parallel trees, the tree associated with the memory layout per process for the processes active during the data collection period and the tree needed for accumulating, organizing and accessing the sampling data. Separating the two trees, while maintaining their interconnections will assist the algorithms.

There are some other structures associated with the data collection session for event lists, the command line, run time environment and so on. These will be discussed later.

In both of the dominant structure cases there is a tree per process and a linked list of the process trees. For the memory maps the processes are identified by the PID. For the data accumulation tree, the “processes” can be identified by any number of attributes. Gooda uses the process path, thereby aggregating the data in cases of process parallel execution or cases where processes are relaunched many times during the collection (ex: bootstrapping GCC).

In Gooda the process structures that anchor the two types of trees are in fact identical. The first PID, of the set of processes with the same path, that gets an event sample becomes the “principal” process structure and all the other processes of that set get their “principal process pointers” set to point at the principal process. The principal process structure points to itself. The ensures the samples are aggregated as only the principal process has a sampling event data tree associated with it, the other PIDs have null pointers for these fields.

The process structures are created on a new process being identified in the stream of records in the perf.data file. This is usually done with a comm or fork record. Processing these records will create a process structure for the first mention of the PID and if the PID has already been observed then these records will indicate either a new thread or a change of the process name. For each process there are a linked list of thread structures.

The mmap records indicate the loading of a module for a process. This triggers the creation of an mmap structure and each PID has a linked list of its mmap structures defining its virtual memory mapping. As modules can be unmapped and remapped and the event samples are not in increasing temporal order, the valid time periods for the mmap structures must be included in their attributes. Thus a given module can have multiple mmap structures even within a single PID.

The principal process trees are used to accumulate the event sample counts. A principal process has a linked list of module structures. All of the mmap structures for the family of PIDs that are associated with a principal process have elements that are the pointers to the appropriate module structure. Thes are matched initially through the paths to the to the module. Each module structure in turn has a linked list of “relative virtual address” (RVA) structures. These are where the samples per instruction pointer are accumulated. The RVA structures can be accessed through a linked list or through a dynamic hash table.

The event sample data is accumulated in many of these structures to easily allow analyses and displays for varying granularities. The raw sample counts are accumulated per PID and TID, per principal process, per module and per RVA.

Each module also has a linked list of function strtuctures. These are created after the perf.data file has been completely read and the sample counts accumulated in the assorted structures. They are only created for modules with sample records. They are defined by the data extracted from piping “readelf -s module” to a file which is then read, sorted and processed to remove duplicate entries and only contain function address ranges.
The “hot” functions will later be disassembled and using that data, linked lists of basic block and asm structures will be created. These will be used for creating the disassembly and control flow graph displays and will therefore get sample counts appropriately created. There will also be a source file structure created which will contain a linked list of source line structures used for accumulating sample counts.

The event samples constitute the bulk of the perf.data file. They come in blocks of records for a single core. When the flow changes from a block for one core to another, the time of the interupt of the first record from the new core will be earlier than the last event from the old core. This lack of temporal ordering must be dealt with.
The event records contain a core number, a PID and TID, a time, some data for evaluating multiplex corrections and a file descriptor. The file descriptor can be matched to a field in a linked list of event attribute structures and from there the event can be identified, but only with a perf event select MSR programming. There is no event name. This is an appallingly bad design. A command line record was added which allows the identification of the event names.
In order to avoid a lot of list walking, an array that directly maps the file descriptors to the event structures and the gooda structures made from the command line data is used. It is dimensioned to a size of largest file descriptor value - smallest FD value. The a given FD - smallest FD value is the lookup index.
The data is recorded in a sample data array of integers. The array is of length num_cores*num_events + num_sockets*num_events + num_events + derived events. Thus it is really the sum of several multidimensional arrays. Each principal structure has such an array, as do the per thread structures.

Analysis