2 How to use a machine description

A machine description (“MD”) contains both explicit information about resources within a machine - detailed by specific nodes within the MD, and implicit information about the relationship of those resources - detailed by how nodes are interconnected into a relationship graph. We detail the relationship properties later in this section.

2.1 Using the MD as a list

For the simplest of sun4v guest operating environments, details of memory system hierarchy or even cache sizes are of little to no importance. Rather, basic information such as available memory regions and numbers of virtual CPUs are sufficient for the environment to function.

Therefore the MD is designed to enable the extraction of basic information without the need to parse any of the inter-relational information also provided.

For example, a simple guest may wish to simply determine the number of CPUs available in the machine. Within the MD each CPU is represented by a node of type “cpu” (please see section 4 for the definition of node types).

A guest may then, starting at the first node in the MD, simply linearly walk the list of nodes from one to the next in the list looking for nodes of a specific type. As each specific node is found properties may then be read from within that node. Pseudo code for this is illustrated in figure 1 below.

```
int find_node_idx(uint_t *bufferp, char *namep)
{
       struct MD_HEADER *hdrp;
       struct MD_ELEMENT * nodep;
       int i, nelems;
       char *strp;

       hdrp = (void*)bufferp;
       nodep = (void*) (bufferp+16);
       nelems = hdrp->node_blk_sz / 16;
       strp = buffer + 16 + hdrp->node_blk_sz;

       for (i=0; i<nelems; i=nodep[i].d.val) {
              char *sp;
              if (strcmp(strp+node[i].name_offset,
                     namep)==0) return i;
       }

       return (-1); /* failed */
}
```

Figure 1Pseduo C code forwalking the list of nodes

2.2 Accelerating string lookups

To search for specific nodes or properties within a node, list element names need to be matched against known strings. The name for each list element is indirectly referenced in the name block of the machine description.

The basic method of searching for nodes or properties implies that for each tagged element in the machine description list, the name string must be found (using the offset in the element) and then the string compared against the desired string value.

While providing correct results these numerous string compares slow searching of the machine description.

The string match process may be short circuited due to the property of uniqueness of strings in the name block. The name block is constructed to guarantee that each string appears only once in the name block regardless of the number of times it is referenced by different elements. Since a desired string (e.g. “cpu”) can appear at most once in the name block, the index to that string in the name block becomes as unique as the string itself.

With this knowledge a more trivial method of searching the MD, is to first find the strings of interest in the name block - thus identifying the unique index for each string name. Then the MD itself can be seached by trivially matching the first 64 bytes of each element.

For example, suppose we wish to count the number of cpus represented in the MD. We first identify the string “cpu” in the name block; for our example it might appear at index 0x123. Thus any element uniquely identify the start of a cpu node will have the tag value 'N', name length of 4 (3 plus the nul string terminator) and name offset of 0x123. So then in the binary image of our example MD the first 64bits of any “cpu” node element will have the unique value of 0x4e0300000123.

A trivial linear search of the MD for this pattern enables nodes of type “cpu” to be counted;

Similarly, sought elements within a node can be matched using the same method of testing the first 64bits of the element structure.

Elements describing the start of a node have the specific property that the value field (elem_ptr->d.val) holds the index of the element for the next node in the machine description. So when searching specifically for node elements, other elements in the MD are trivially skipped thus speeding the search;

It is recommended that guests using the MD initially search and cache the indices of desired strings from the MD name block to avoid even the cost of finding the matching string index for each new MD search.

It should noted however, that the name block is unique to a particular MD. If the guest requests a new copy of a MD from the hypervisor, there is no guarantee that strings will have the same indices in the name block of the new MD as they have in the name block of the old MD.

3 Directed Acyclic Graph

3.1 Graph contents

The intrinsic Machine Description (MD) is a directed acyclic graph (DAG) of nodes describing resources or information available within a machine. This information is provided upon request to a guest operating system via the machine description request API.

3.1.1 Graph nodes

The DAG nodes are defined by the “NODE” element within the element list, and contain all the properties and arcs described until the subsequent NODE_END element. DAG node names form a well defined name space such that a particular name describes the type of a well defined entity. A different type of entity must be described by a node of a different name. For example, a CPU may be described by of type “cpu”, while a cache is described by a node of type “cache”.

Each node is a specific instance of the entity it describes. Properties or named values held within that node provide relevant details of the corresponding entity. For example, a cache node will hold a list of properties describing attributes of that cache.

As a node is defined by a specific “NODE” element within the element list, then for a specific MD, we can uniquely refer to that node by the index of its starting node element withing the element list. Thus if a “cpu” node starts at list element number 27, then a unique reference to that “cpu” node is the index value 27.

Using these index values for node start list elements, we can now provide pointers or “arcs” to point to other nodes. In the construction of the MD element list, we define the 64bit data payload of a “NODE” element to contain the index to the next “NODE” element in the element list. Thus a simple linear list of nodes is formed within the MD element list that enables searching for nodes of specific types without having to scan every list element looking for “NODE” and “NODE_END” tags.

Simlarly, using the PROP_ARC, type we can build a link or arc from one node to another. The value field of a PROP_ARC element is the 64bit element index of the “NODE” element pointed to. It is illegal for a PROP_ARC element to point to anything other than a NODE element, or a NOOP element (outside a node).

3.1.2 DAG construction

A DAG is constructed as described above by arcs that link the nodes together. The interconnection of these arcs explicitly defines the relationship between the nodes. For example, if node A has an arc to node C and node B has an arc to node C then the relationship exposed is that within the graph both nodes A and B share node C ***and*** any nodes that C arcs to. In the example illustration shown in figure 2 below we can see an instruction cache that is shared by two cpu nodes. The sharing is indicated by the existence of arcs from each cpu node to the same cache node.

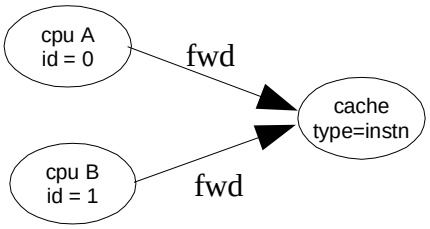

Figure 2Sharing relationship between nodes

The default DAG described within the MD is defined by arcs (element type PROP_ARC) with a name of “fwd”. For convenience in walking this DAG. Arcs named “back” are also provided that define the inverse DAG. Thus for every node A that has a “fwd” arc pointing to another node B, there is a corresponding “back” arc for node B pointing back to A.

The use of named arcs enables other DAGs to be built and contained within the same MD, however none other than the DAGs defined by the “fwd” and “back” arcs are currently defined.

3.2 Required nodes

The MD DAG will vary according to the resources available within a machine, and certain nodes may be present in a machine on one machine architecture, but not on a different machine architecture.

The MD concept is designed to allow for certain nodes to be “optional”, however, to allow for the MD to be useable at all certain nodes must be defined and present in the description. These are “required” nodes and are guaranteed to be present if the resource they describe is present within the machine.

3.3 The vanilla MD

Normally a MD is a full description of the resources available to specific logical domain. However, it is a requirement for any sun4v guest operating system that it be able to handle any machine description capable of being defined by this document and its subsequent revisions. To this end, a Guest operating system must be able to ignore / skip over nodes whose type and definitions the OS has never seen before, and most importantly that same Guest must follow some default fall-back behavior when information is not available.

To test the requirement for a default fall-back behavior, we define a “vanilla” description that contains only the core and required nodes for a given platform. This guarantees that a Guest OS is given no information about the platform upon which it is running, and to test that it continues to boot and execute - though optimal performance is no longer required.

The nodes in the vanilla MD are therefore required and sufficient to describe a guest environment for a basic sun4v compatible Operating System.

3.4 Formation and meaning of a DAG

As mentioned above a parition description currently contains only one DAG, and this is defined by all arcs with the name “fwd”. As a courtesy, in order to speed certain searches, the MD also contains the inverse of this DAG built using arcs of name “back”. Clearly the “back” DAG could be built by a guest from the “fwd” DAG, however the basic MD contains both to help lower the burden on the Guest.

Future revisions of this spec. may include new nodes, and importantly new DAGs within the same MD. Current software should be designed to ignore arcs with names other than “fwd” and “back” in order to remain future proof. Future MD will be implemented so as not to have conflicts with the vanilla fwd and back DAGs.

To understand how to use the DAGs in a MD consider the DAG built using the “fwd” arcs.

The root of the “fwd” DAG is a node of type “root”. This is by definition the very first node in the MD. It can be found very simply by scanning the MD element list for the first NODE definition (though unfortunately, due to the existence of NOOP elements, this need not be at element index 0).

From the root node, “fwd” arcs lead to nodes describing the various components within the logical domain a guest is using.

The root node in turn contains “fwd” arcs to collective nodes for cpus, memory and various forms of I/O, as well as nodes targeted to specific consumers such as OpenBoot, the sun4v verification suite (sun4v-vs) etc. A simple DAG is shown below by way of illustration.

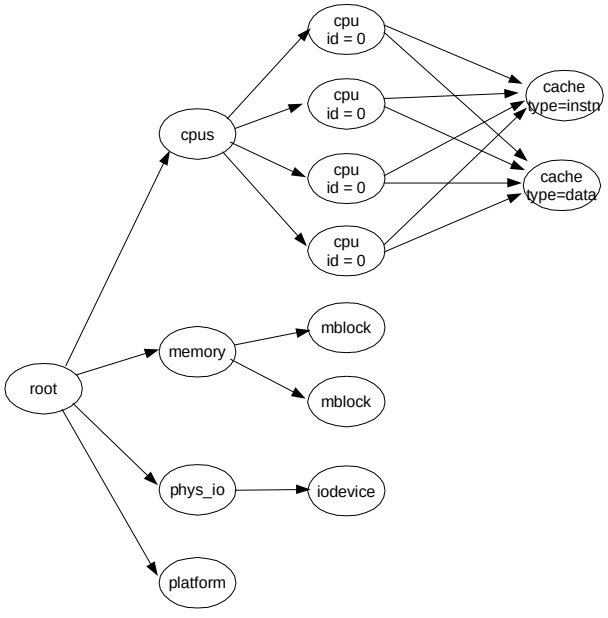

Figure 3 A Simple DAG

For clarity, not all DAG arcs have been shown.