Systune: A System Resource Provisioning Framework

Table of Contents

Introduction

Gaining control over system resources such as the CPU, caches, and GPU is a powerful capability in any developer’s toolkit. By fine-tuning these components, developers can optimize the system’s operating point to make more efficient use of hardware resources and significantly enhance the user experience.

For example, increasing the CPU's Dynamic Clock and Voltage Scaling (DCVS) minimum frequency to 1 GHz can boost performance during demanding tasks. Conversely, capping the maximum frequency at 1.5 GHz can help conserve power during less intensive operations.

The Systune framework supports `Signals` which is dynamic provisioning of system resources in response to specific signals—such as app launches or installations—based on configurations defined in JSON. It allows business units (BUs) to register extensions and add custom functionality tailored to their specific needs.

Syslocks APIs

This API suite allows you to manage system resource provisioning through tuning requests. You can issue, modify, or withdraw resource tuning requests with specified durations and priorities.

`tunelock`

Description: Issues a resource provisioning (tuning) request for a finite or infinite duration.

Function Signature:

```
int64_t tunelock(int64_t duration,
                 int32_t prio,
                 int32_t numRes,
                 int8_t backgroundProcessing,
                 std::vector<Resource*>* res);
```

Parameters:

- `duration` (`int64_t`): Duration in milliseconds for which the Resource(s) lock should be held. Use `-1` for an infinite duration.
- `prio` (`int32_t`): Priority level of the request.
- `numRes` (`int32_t`): Number of resources to be tuned as part of the Request.
- `backgroundProcessing` (`int8_t`): Boolean Flag to indicate if the Request is to be processed in the background.
- `res` (`std::vector<Resource*>*`): Pointer to a list of resources to be provisioned. Details about the resource format are provided below (Refer section "Resource Format").

Returns: `int64_t`

- **A positive unique handle** identifying the issued request (used for future `retune` or `untune` operations)
- `-1` otherwise.

`retunelock`

Description: Modifies the duration of an existing Tune request.

Function Signature:

```
int8_t retunelock(int64_t duration,
                  int64_t handle);
```

Parameters:

- `duration` (`int64_t`): New duration in milliseconds. Use `-1` for an infinite duration.
- `handle` (`int64_t`): Handle of the original request, returned by the call to `tunelock`.

Returns: `int8_t`

- `0` if the request was successfully submitted.
- `-1` otherwise.

`untunelock`

Description: Withdraws a previously issued resource provisioning (or Tune) request.

Function Signature:

```
int8_t untunelock(int64_t handle);
```

Parameters:

- `handle` (`int64_t`): Handle of the original request, returned by the call to `tunelock`.

Returns: `int8_t`

- `0` if the request was successfully submitted.
- `-1` otherwise.

---

getprop

Description: Gets a property from the Config Store

Function Signature:

```
int8_t getprop(const char* prop,
               char* buffer,
               size_t buffer_size,
               const char* def_value);
```

Parameters:

- `prop` (`const char*`): Name of the Property to be fetched.
- `buffer` (`char*`): Pointer to a buffer to hold the result, i.e. the property value corresponding to the specified name.
- `buffer_size` (`size_t`): Size of the buffer.
- `def_value` (`const char*`): Value to be written to the buffer in case a property with the specified Name is not found in the Config Store

Returns: `int8_t`

- `0` If the Property was found in the store, and successfully fetched
- `-1` otherwise.

setprop

Description: Modifies an already existing property in the Config Store.

Function Signature:

```
int8_t setprop(const char* prop,
               const char* value);
```

Parameters:

- `prop` (`const char*`): Name of the Property to be fetched.
- `value` (`const char*`): A buffer holding the new the property value.

Returns: `int8_t`

- `0` If the Property with the specified name was found in the store, and was updated successfully.
- `-1` otherwise.

Resource Format

As part of the tunelock APIs, the resources (which need to be provisioned) are specified by using a List of `Resource` structures. The format of the `Resource` structure is as follows:

```
typedef struct Resource {
    uint32_t OpId;
    uint32_t OpInfo;
    uint32_t OptionalInfo;
    uint16_t NumValues;
    union {
        int32_t Value;
        int32_t *Values;
    };
} Resource;
```

OpId: An unsigned 32-bit unique identifier for the resource. It encodes essential information that is useful in abstracting away the system specific details.

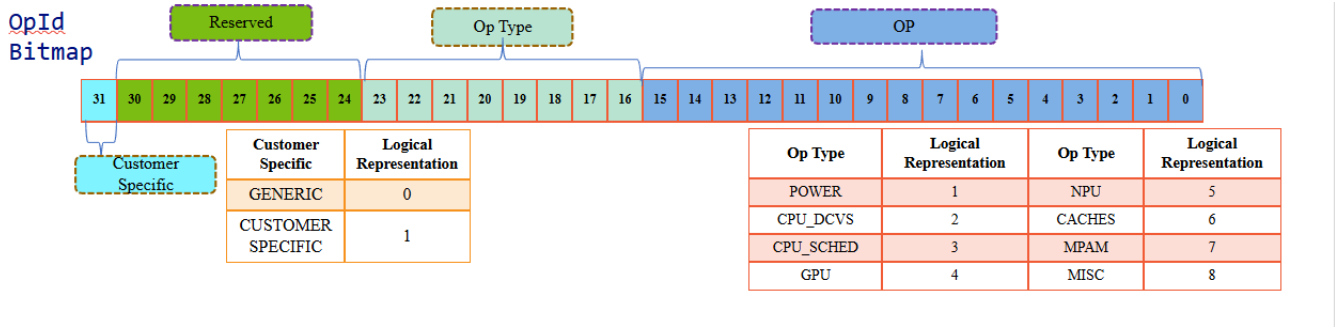
Systune implements a System Independent Layer (SIL) which Provides a Transparent and Consistent way for Indexing Resources. This makes it easy for the Clients to Identify the Resource they want to provision, without needing to worry about Compatability Issues across Targets and the Order in which the Resources are described in the JSON files. Essentially, the Resource ID (unsigned 32 bit) is broken up into two fiels:

- Resource Opcode (16 bits)
- Resource Optype (8 bits)
- [Additionally if BU is providing their own Custom Resource Config Files, then the MSB must be set to "1", Indicating this is a Custom Resource else it shall be treated as a Default Resource].

These fields can uniquely identify a Resource across targets, hence making the code operating on these Resources interchangable. In Essence, we ensure that the Resource with ID "x", refers to the same Tunable Resource across different Targets.

Examples:

- The Resource ID: 65536 [00000000 00000001 00000000 00000000], Refers to the Default Resource with Opcode 0 and Optype 1.
- The Resource ID: 2147549185 [10000000 00000001 00000000 00000001], Refers to the Custom Resource with Opcode 1 and Optype 1.



Customer Specific	Logical Representation
GENERIC	0
CUSTOMER SPECIFIC	1

Op Type	Logical Representation	Op Type	Logical Representation
POWER	1	NPU	5
CPU_DCVS	2	CACHES	6
CPU_SCHED	3	MPAM	7
GPU	4	MISC	8

OpInfo: Encodes operation-specific information such as the Logical cluster and core IDs, and MPAM part ID.

OptionalInfo: Additional optional metadata, useful for custom or extended resource configurations.

NumValues: Number of values associated with the resource. If multiple values are needed, this must be set accordingly.

Value / Values: It is a single value when the resource requires a single value or a pointer to an array of values for multi-value configurations.

Example Usage of Syslock APIs

tunelock

The below example demonstrates the use of the tunelock API for Resource Provisioning.

```
void sendRequest() {
    // Define resources
    Resource cpuFreq;
    cpuFreq.OpId = 0x00010001;          // Example OpId for CPU frequency
    cpuFreq.OpInfo = 0x00000001;        // Example: target cluster/core
    cpuFreq.OptionalInfo = 0;           // No optional info
    cpuFreq.NumValues = 1;
    cpuFreq.Value = 1500000;            // 1.5 GHz

    Resource gpuFreq;
    gpuFreq.OpId = 0x00020001;          // Example OpId for GPU frequency
    gpuFreq.OpInfo = 0x00000001;        // Example: target GPU unit
    gpuFreq.OptionalInfo = 0;
    gpuFreq.NumValues = 1;
    gpuFreq.Value = 600000;             // 600 MHz

    std::vector<Resource*> resources = { &cpuFreq, &gpuFreq };

    // Issue a tuning request for 10 seconds with priority 1
    int64_t handle = tunelock(10000, 1, resources.size(), &resources);
    if (handle < 0) {
        std::cerr << "Failed to issue tuning request." << std::endl;
        return -1;
    }
    std::cout << "Tuning request issued. Handle: " << handle << std::endl;
}
```

retunelock

The below example demonstrates the use of the retunelock API for modifying a Request's duration.

```
void sendRequest() {
    // Modify the duration to 20 seconds
    if (retunelock(20000, handle) == 0) {
        std::cout << "Tuning request updated to 20 seconds." << std::endl;
    } else {
        std::cerr << "Failed to retune request." << std::endl;
    }
}
```

untunelock

The below example demonstrates the use of the untunelock API for untuning a previously issued Request.

```
void sendRequest() {
    // Withdraw the tuning request
    if (untunelock(handle) == 0) {
        std::cout << "Tuning request successfully withdrawn." << std::endl;
    } else {
        std::cerr << "Failed to untune request." << std::endl;
    }
}
```

Config Files Format

Systune utilises JSON files for configuration. This includes the Resources, Signals and Profiles Config Files. The BUs can provide their own Config Files, which are specific to their use-case through the Extension Interface

1. Resource Configs

These files store resource-specific information. They are stored as a JSON-array.Their detailed format and description is shown below:

Field Descriptions

Field	Type	Description
`Opcode`	`string`	Operation ID associated with the lock.
`Name`	`string`	Path to the system sysfs node.
`Supported`	`boolean`	Indicates if the lock is supported on the target.
`HighThreshold`	`number`	Upper threshold value for the resource.
`LowThreshold`	`number`	Lower threshold value for the resource.
`Permissions`	`string`	Type of client allowed (`system` or `regular`).
`Modes`	`array`	Display modes applicable (`"display_on"`, `"display_off"`, `"doze"`).
`Policy`	`string`	Concurrency policy (`"higher_is_better"`, `"lower_is_better"`, `"instant_apply"`, `"lazy_apply"`).

Example

```
"LocksConfigs": [
  {
    "Opcode": "0x0",
    "Name": "/proc/sys/kernel/sched_util_clamp_min",
    "Supported": true,
    "HighThreshold": 1024,
    "LowThreshold": 0,
    "Permissions": "system",
    "Modes": ["display_on", "doze"],
    "Policy": "higher_is_better"
  },
  {
    "Opcode": "0x1",
    "Name": "/proc/sys/kernel/sched_util_clamp_max",
    "Supported": true,
    "HighThreshold": 1024,
```

```
      "LowThreshold": 0,
      "Permissions": "system",
      "Modes": ["display_on", "doze"],
      "Policy": "higher_is_better"
    }
  ]
```

---

2. Properties Configs

This targetPropertiesConfigs.json file stores various properties which are used by the Systune Modules internally (for example, to allocate sufficient amount of Memory, or to determine the Pulse Monitor Duration) as well as by the End Client.

Field Descriptions

Field	Type	Description
`Name`	`string`	Unique name of the parameter
`Value`	`integer`	The value for the parameter.

Example

```
{
  "PropertyConfigs": [
    {"Name": "syslocks.maximum.concurrent.requests", "Value" : "130"},
    {"Name": "syslocks.maximum.resources.per.request", "Value" :"5"},
  ]
}
```

3. SysSignals

This file stores all tunable parameters that the end-user can tune according to their usecase. This enables our service to perform optimally for different clients. For example: The amount of memory pre-allocated differs a lot if the service is being used in a large server or a small modem. This user-aware optimizations can be made through these configs.

Field Descriptions

Field	Type	Description
`Name`	`string`	Unique name of the parameter
`Value`	`integer`	The value for the parameter.

Example

```
{
  "PropertyConfigs": [
    {"Name": "syslocks.maximum.concurrent.requests", "Value" : "130"},
    {"Name": "syslocks.maximum.resources.per.request", "Value" :"5"},
  ]
}
```

Systune Features

1. Permissions

Certain resources can be tuned only by system clients and some which have no such restrictions and can be tuned even by third party clients. The Client permissions are dynamically determined, the first time it makes a Request. If a client with Third Party Permissions tries to tune a Resource, which allows only clients with System Permissions to tune it, then the Request shall be dropped.

2. Policies

To ensure efficient and predictable handling of concurrent requests, each system resource is governed by one of four predefined policies. Selecting the appropriate policy helps maintain system stability, optimize performance, and align resource behavior with application requirements.

- Instant Apply (or Always Apply): This policy is for resources where the latest request needs to be honored. This is kept as the default policy.
- Higher is better: This policy honors the request writing the highest value to the node. One of the cases where this makes sense is for resources that describe the upper bound value. By applying the higher-valued request, the lower-valued request is implicitly honored.
- Lower is better: Self-explanatory. Works exactly opposite of the higher is better policy.
- Lazy Apply: Sometimes, you want the resources to apply requests in a first-in-first-out manner.

3. Priorities

As part of the tunelock API call, the Client is allowed to specify a desired Priority Level for the Request. Systune supports 4 priority levels:

- System High [SH]
- System Low [SL]
- Third Party High (or Regular High) [TPH]
- Third Party Low (or Regular Low) [TPL]

Requests with a higher Priority will always be prioritized, over another Request with a lower priority. Note, the Request Priorities are related to the Client Permissions. A client with System Permission is allowed to acquire any priority Level it wants, however a Client with Third Party Permissions can only acquire either Third Party High (TPH) or Third Party Low (TPL) level of Priorities. If a Client with Third Party Permissions tries to acquire a System High or System Low level of Priority, then the Request will not be honoured.

4. Detection of Dead Clients and Subsequent Cleanup

To improve efficiency and conserve Memory, it is essential to Regularly Check for Dead Clients and Free up any System Resources associated to them. This includes, Untuning all (if any) Ongoing Tune Request issued by this Client and Freeing up the Memory used to store Client Specific Data (Example: Client's List of Requests (Handles), Health, Permissions, Threads Associated with the Client etc). Systune Ensures that such clients are detected and Cleaned Up within 90 seconds of the Client Terminating.

Systune performs these actions by making use of two components:

- Pulse Monitor: Pulse Monitor scans the list of the Active Clients, and checks if any of the Client (PID) is dead (It does by checking if an entry for that PID exisits in /proc/pid/). If it finds a Dead Client, it schedules the Client for Cleanup by adding this PID to a Queue (called the GC Queue).
- Client Garbage Collector: When the Garbage Collector runs it iterates over the GC Queue and Performs the Cleanup.

Both Pulse Monitor and Client Garbage Collector run as Daemon Threads.

5. Preventing System Abuse

Systune has a built in RateLimiter component that prevents abuse of the system by limiting the number of requests a client can make within a given time frame. This helps to prevent clients from overwhelming the system with requests and ensures that the system remains responsive and efficient. RateLimiter works on a Reward / Punishment methodology. Whenever a Client enters the System for the first time, it is assigned a "Health" of 100. A Punishment is incurred if a Client makes subsequent Requests in a very short Time Interval (called Delta, say 5 ms). A Reward results in increasing the health of a Client (not above 100), while a Punishment involves decreasing the health of the Client. If at any point this value of Health reaches Zero then any further Requests from this Client wil be dropped. Note the Exact value of Delta, Punishment and Rewards are BU-configurable.

6. Duplicate Checking

Systune's RequestManager component is Responsible for detecting any duplicate Requests issued by a Client, and dropping them. This is done by maintaining a List of all the Requests issued by a Client. Whenever a new Request is received, it is checked against this List to see if it is a duplicate. If it is, then the Request is dropped. If it is not, then the Request is added to this List and processed. Duplicate Checking helps to improve System Efficiency, by saving wasteful CPU time on processing Duplicates.

7. Logical to Physical Mapping

Logical to Physical Core / Cluster Mapping helps us to achieve achieve decoupling on the Client side, as the Client does not need to be aware of the Physical Topology of the Target to issue Resource Tuning Requests. Instead the Client can specify Logical values for Core and Cluster. Systune will translate these values to their physical counterparts and apply the Request accordingly. Logical to Physical mapping in essence like System Independent Layer makes the same client code interchangable across different Targets, and Systune will take care of the mapping.

8. Display-Aware Operational Modes

The system's operational modes are influenced by the state of the device's display. To conserve power, certain system resources are optimized only when the display is active. However, for critical components that require consistent performance—such as during background processing or time-sensitive tasks, resource tuning can still be applied even when the display is off, including during low-power states like Doze mode. This ensures that essential operations maintain responsiveness without compromising overall energy efficiency.

9. Crash Recovery

In case of Server Crash, Systune ensures that all the Resource Sysfs Nodes are restored to a Sane State, i.e. they are reset to their Original Values. This is done by maintaining a List of all the Resource Sysfs Nodes and their Original Values, before any modification was made on behalf of the Clients by Systune. In the event of Server crash, this File is read and all Sysfs Nodes are reset to their Original Values.

10. Flexible Packaging

The Users are free to pick and Choose the Systune Modules they want for their use-case and which fit their constraints. The Syslock Module is the core (central) module, however if the Users choose they can add on top of it other Modules: SysSignals, Profiles and Extensions.

11. Pre-Allocate Capacity for efficiency

Systune provides a MemoryPool component, which allows for pre-allocation of memory for certain commonly used type at the time of Server initialization. This is done to improve the efficiency of the system, by reducing the number of memory allocations and deallocations that are required during the processing of Requests. The allocated memory is managed as a series of blocks which can be recycled without any system call overhead. This reduces the overhead of memory allocation and deallocation, and improves the performance of the system.

Further, a ThreadPool component is provided to pre-allocate processing capacity. This is done to improve the efficiency of the system, by reducing the number of thread creation and destruction required during the processing of Requests, further ThreadPool allows for the Threads to be repeatedly reused for processing different tasks.

Extension Interface

The Systune framework allows business units (BUs) to extend its functionality and customize it to their use-case. Extension Interface essentially provides a series of hooks to the BUs to add their own custom behaviour. This is achieved through a lightweight extension interface using macros. This happens in the initialisation phase before the service is ready for requests.

Specifically the Extension Interface provides the following capabilities:

- Registering custom resource handlers
- Registering Custom Configuration Files (This includes Resource Configs, Signal Configs, Profiles and Property Configs). This allows, for example the specification of Custom Resources.

Macros

URM_REGISTER_RESOURCE

Registers a custom resource handler with the system. This allows the framework to invoke a user-defined callback when a specific resource opcode is encountered. A function pointer to the callback is to be registered. Now, instead of the normal resource handler, this callback function will be called when a syslock request for this particular resource opcode arrives.

Usage Example

```
int32_t applyCustomCpuFreqCustom(Resource* res) {
    // Custom logic to apply CPU frequency
    return 0;
}

URM_REGISTER_RESOURCE(0x00010001, applyCustomCpuFreqCustom);
```

URM_REGISTER_CONFIG

Registers a custom configuration JSON file. This enables the BU to provide their own Config Files, i.e. allowing them to provide their Own Custom Resources for Example.

Usage Example

```
URM_REGISTER_CONFIG(RESOURCE_CONFIG,
"/etc/bin/targetResourceConfigCustom.json");
```

The above line of code, will indicate to Systune to Read the Resource Configs from the file "/etc/bin/targetResourceConfigCustom.json" instead of the Default File. Note, the BUs must honour the

structure of the JSON files, for them to be read and registered successfully.

Custom Signal Config File can be specified similarly:

Usage Example

```
URM_REGISTER_CONFIG(SIGNALS_CONFIG,
"/etc/bin/targetSignalConfigCustom.json");
```

Server CLI

The **Systune Server** runs as a background service, initializing configurations and registering extensions to handle incoming requests.

Commands

- `start` Launches the server, loads all configuration files, and prepares for request handling.
- `exit` Gracefully shuts down the server.
- `dump` Displays all currently active requests in the system.

Client CLI

REDO: NEED TO REWRITE TO MATCH CURRENT IMPLEMENTATION

The **Systune Client** sends tuning-related requests to the server via command-line interface.

Usage Examples

1. Send Tune Requests from JSON File

```
./client_ex -j
```

- Reads requests from `SampleRequests.json`. TODO:show example.

2. Send Tune Request via CLI

```
./client_ex -i -v 0:567 -d 5000 -p 1
```

- `-i` : Initiates a tune request
- `-v` : Opcode:Value pairs (comma-separated, no spaces)
- `-d` : Duration in milliseconds
- `-p` : Priority level

3. Send Untune Request

```
./client_ex -u -h 1
```

- `-u` : Untune request

- `-h` : Handle ID

4. Send Retune Request

```
./client_ex -r -h 1 -d 8000
```

- `-r` : Retune request
- `-h` : Handle ID
- `-d` : New duration in milliseconds