dPF API

Version 0.1, February 2025

# 1 Overview

The DPF API provides a simple yet robust mechanism for communication between user-space applications and the kernel module. It is designed to manage system parameters such as CPU core allocation, DDR bandwidth settings, and monitoring intervals. All communication occurs through a dedicated procfs interface: /proc/dpf.

The kernel module creates an entry with the name dpf under /proc/. This entry should allow simultaneous read/write access with permissions 0666. The module reads messages from /proc/dpf, decodes them according to the defined message structures, processes the request, and writes a response. The module supports asynchronous read/write operations to ensure proper communication with user-space. Furthermore the module supports concurrent request handling to support multiple user-space applications at the same time.

User-space application is required to: 1) Ensure message buffers are properly allocated before writing to /proc/dpf. 2) Handle partial reads/writes appropriately, considering large messages.

## 1.1 Message structure

The API is based on user-space request messages to the kernel and then kernel responses back to the users-space application. Note that a series of responses can be given, such as subscription to PMU data updates.

All message structs from user-space are started with the name dpf\_req\_ and all responses from the kernel starts with dpf\_resp\_. For example, the initialization sequence is made through the struct dpf\_req\_init and the kernel response is given by struct dpf\_resp\_init.

All messages starts with a common header to identify the type of message:

struct dpf\_msg\_header {

\_\_u32 type;

\_\_u32 payload\_size;

};

**type** is the message type identifier, see appendix A for the list of message types to numbers.  
**payload\_size** is the number of bytes for the whole message structure including the header.

## 1.2 Initialization sequence

The user-space application should verify that the /proc/dpf interface is available. If this is not the case then the kernel module is not loaded and the appropriate error message should be given.

The first message the send to the kernel module is the DPF\_MSG\_INIT. This initializes the communication and the current API version support is returned. Make sure to only continue if the version is compatible with the user-space application.

# 2 API Messages

## **2.1 DPF\_MSG\_INIT - API Version Negotiation**

Used by user-space to request the current API version from the kernel module.

### User-space request

struct dpf\_req\_init {

struct dpf\_msg\_header header;

};

No user-space argument is used for the init request.

### Kernel response

struct dpf\_resp\_init {

struct dpf\_msg\_header header;

\_\_u32 version;

};

**version** specifies the API version supported by the kernel.

### Example

Request: {DPF\_MSG\_INIT, 8}

Where DPF\_MSG\_INIT is the message id. 8 is the size of the message.

Response:{DPF\_MSG\_INIT, 12, 1}

Where DPF\_MSG\_INIT is the message id. 12 is the size of the message. 1 is the API version.

## 2.2 DPF\_MSG\_CORE\_RANGE - Core Range Configuration

This defines the CPU core range allocated for processing, allowing users to optimize workload distribution. Thread count indicates the actual number of threads available, accounting for factors like unavailable cores and system limits. Simply subtracting the start core from the end core may lead to inaccurate results.

### User-space Request

struct dpf\_core\_range {

struct dpf\_msg\_header header;

\_\_u32 core\_start;

\_\_u32 core\_end;

};

**core\_start** specifies starting CPU core.  
**core\_end** specifies the last CPU core in the range.

### Kernel Response

struct dpf\_resp\_core\_range {

struct dpf\_msg\_header header;

\_\_u32 core\_start;

\_\_u32 core\_end;

\_\_u32 thread\_count;

};

**core\_start** specifies starting core confirmed by the kernel.  
**core\_end** specifies ending core confirmed by the kernel.  
**thread\_count** specifies actual number of available threads within the range.

### Example:

Request: {DPF\_MSG\_CORE\_RANGE, 16, 8, 12}

Here, DPF\_MSG\_CORE\_RANGE is the message ID, 16 is the message size, 8 is the starting core, and 12 is the ending core, indicating a request to allocate cores from 8 to 12 (inclusive).

Response: {DPF\_MSG\_CORE\_RANGE, 20, 8, 12, 5}

Here, DPF\_MSG\_CORE\_RANGE is the message ID, 20 is the message size, 8 is the start core, 12 is the end core, and 5 is the thread count.

## 2.3 DPF\_MSG\_DDRBW\_SET - DDR Bandwidth Setting

Allows users to manually set the DDR bandwidth, which can be useful for performance tuning or limiting memory usage.

### User-space Request

struct dpf\_ddrbw\_set {

struct dpf\_msg\_header header;

\_\_u32 set\_value;

};

**set\_value** specifies the DDR bandwidth value to be set by the user.

### Kernel Response

struct dpf\_resp\_ddrbw\_set {

struct dpf\_msg\_header header;

\_\_u32 confirmed\_value;

};

**confirmed\_value** provides the DDR bandwidth value that the kernel has confirmed as set.

### Example:

Request: {DPF\_MSG\_DDRBW\_SET, 12, 4000}

Where DPF\_MSG\_DDRBW\_SET is the message ID, 12 is the size of the message, and 4000 is the desired DDR bandwidth in MB/s.

Response: {DPF\_MSG\_DDRBW\_SET, 12, 4000}

Where DPF\_MSG\_DDRBW\_SET is the message ID, 12 is the size of the message, and 4000 is the confirmed DDR bandwidth in MB/s.

## 2.4 DPF\_MSG\_CORE\_WEIGHT - Core Weight Assignment

Assigns priority weights to CPU cores, enabling fine-grained control over task scheduling and resource allocation. A higher weight value indicates greater priority, influencing how tasks are distributed across the available cores.

### User-space Request

struct dpf\_core\_weight {

struct dpf\_msg\_header header;

\_\_u32 count;

\_\_u32 weights[];

};

**count** specifies the number of weights being set.  
**weights[]** holds the of weight values to be assigned.

### Kernel Response

struct dpf\_resp\_core\_weight {

struct dpf\_msg\_header header;

\_\_u32 confirmed\_weights[];

};

**confirmed\_weights[]** provides the confirmed weight values as acknowledged by the kernel.

### Example:

Request: {DPF\_MSG\_CORE\_WEIGHT, 28, 4, [10, 20, 30, 40]}

Where DPF\_MSG\_CORE\_WEIGHT is the message ID, 28 is the size of the message, 4 is the number of weight values, and [10, 20, 30, 40] are the assigned weights.

Response: {DPF\_MSG\_CORE\_WEIGHT, 24, [10, 20, 30, 40]}

Where DPF\_MSG\_CORE\_WEIGHT is the message ID, 24 is the size of the message, and [10, 20, 30, 40] are the weight values.

## 2.5 DPF\_MSG\_TUNING\_CONTROL - Kernel Tuning Control

Allows users to enable or disable kernel tuning. By default, tuning is off when the kernel module is loaded.

### User-space Request:

struct dpf\_tuning\_control {

struct dpf\_msg\_header header;

\_\_u32 enable;

};

**enable** determines whether kernel tuning is enable or disable. A value of 0 disables kernel tuning, while 1 enables it.

### Kernel Response:

struct dpf\_resp\_tuning\_control {

struct dpf\_msg\_header header;

\_\_u32 active;

};

**active** indicates the current status of kernel tuning— one (1) if active and zero (0) if inactive.

### Example:

Request: {DPF\_MSG\_TUNING\_CONTROL, 12, 1}

Where DPF\_MSG\_TUNING\_CONTROL is the message ID, and 12 is the size of the message.

Response: {DPF\_MSG\_TUNING\_CONTROL, 12, 1}

Where DPF\_MSG\_TUNING\_CONTROL is the message ID, 12 is the size of the message, and 1 confirms that tuning is active.

Request: {DPF\_MSG\_TUNING\_CONTROL, 12, 0}

Where DPF\_MSG\_TUNING\_CONTROL is the message ID, and 12 is the size of the message.

Response: {DPF\_MSG\_TUNING\_CONTROL, 12, 0}

Where DPF\_MSG\_TUNING\_CONTROL is the message ID, 12 is the size of the message, and 0 confirms that tuning is disabled.

## 2.6 DPF\_MSG\_MSR\_READ - MSR Value Retrieval

Allows users to read the latest MSR value for a specific core.

### User-space Request:

struct dpf\_msr\_read {

struct dpf\_msg\_header header;

\_\_u32 core\_id;

};

**core\_id** specifies the core from which the MSR value is to be read.

### Kernel Response:

struct dpf\_resp\_msr\_read {

struct dpf\_msg\_header header;

\_\_u64 msr\_value[];

};

**msr\_value[]** is an array that holds the MSR values retrieved from the core.

### Example:

Request: {DPF\_MSG\_MSR\_READ, 12, 3}

Where DPF\_MSG\_MSR\_READ is the message ID, and 12 is the size of the message.

Response: {DPF\_MSG\_MSR\_READ, 16, [0x1A2B3C4D5E6F, 0x1234567890ABCDEF, 0x9876543210FEDCBA]}

Where DPF\_MSG\_MSR\_READ is the message ID, 32 is the size of the message, and 0x1A2B3C4D5E6F is the retrieved MSR value.

## 2.7 DPF\_MSG\_PMU\_READ - PMU Value Retrieval

Allows users to read the latest PMU value for a specific core.

### User-space Request:

struct dpf\_pmu\_read {

struct dpf\_msg\_header header;

\_\_u32 core\_id;

};

**core\_id** specifies the core from which the PMU (Performance Monitoring Unit) value is to be read.

### Kernel Response:

struct dpf\_resp\_pmu\_read {

struct dpf\_msg\_header header;

\_\_u64 pmu\_value[];

};

**pmu\_value[]** is an array that holds the PMU values retrieved from the core.

### Example:

Request: {DPF\_MSG\_PMU\_READ, 12, 3}

Where DPF\_MSG\_PMU\_READ is the message ID, and 12 is the size of the message.

Response: {DPF\_MSG\_PMU\_READ, 16, 0x9ABCDEF12345}

Where DPF\_MSG\_PMU\_READ is the message ID, 16 is the size of the message, and 0x9ABCDEF12345 is the retrieved PMU value.

# 3 Appendix A

…