April 2014

NVIDIA Android Tegra Thermal Management

Alexander Sebastian Karp
Worcester Polytechnic Institute

Christina Marie Guertin
Worcester Polytechnic Institute

Wesley Nitinthorn
Worcester Polytechnic Institute

NVIDIA Android Tegra Thermal Management

Christina Guertin, Alexander Karp, Kexin Shi, Wesley Nitinthorn

March 2014

A Major Qualifying Project Report:
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Christina Guertin

Alexander Karp

Kexin Shi

Wesley Nitinthorn

Date: March 2014

Approved:

Professor David Finkel, Advisor

Contents

List of Figures

Abstract

NVIDIA's SOC_THERM hardware takes care of the thermal management on their Tegra chips. Originally, the hardware was managed on the software side by a library. We were tasked with documenting this library and eventually converting it into a platform driver. At the conclusion of our project, we were able to convert the SOC_THERM library so that it conformed with the upstream Linux kernel standard for platform drivers.

Acknowledgments

The team would like to thank the following people for their invaluable help with the project by providing information and guidance:

Professor David Finkel
Matthew Longnecker
Paul Walmsley
Diwakar Tundlam

1. Background

1.1 NVIDIA

Founded in January of 1993, NVIDIA is a semiconductor company that makes graphics processors, wireless communications processors, and digital media player software. In 1999, NVIDIA revolutionized computer graphics by inventing the graphics processing unit (GPU), propelling graphics from a feature into an ever-expanding industry. NVIDIA holds over 6,400 patents worldwide and is a very well-known company within the technology sector [11].

1.1.1 Products

NVIDIA's product families include:

- **GeForce**: A series of gaming GPUs [7, 11]. NVIDIA's first GPU series and its most well-known.
- **Quadro**: A GPU series for professionals, offering high-end performance for digital content creation and computer-aided design [11, 12].
- **Tegra**: A system-on-a-chip (SoC) that is built for mobile devices [10, 11].
- **Tesla**: The first dedicated, general purpose GPU. It is designed for high-end parallelized general processing used in supercomputing [8, 11].

1.1.2 Company History

- **1993**: NVIDIA is founded by Jen-Hsun Huang, Chris Malachowsky and Curtis Priem.
- **1995**: NVIDIA launches NV1, its first product.
- **1999**: NVIDIA invents the GPU.
- **2001**: NVIDIA launches nForce, entering into the integrated graphics market.

- **2004**: The Scalable Link Interface (SLI) is launched.
- **2006**: CUDA architecture is unveiled.
- **2008**: NVIDIA launches the Tegra mobile processor.
- **2010**: NVIDIA powers world's fastest supercomputer [9].
- **2012**: NVIDIA launches Kepler architecture-based GPUs [6].

1.2 Tegra

Tegra is a system-on-a-chip (SoC) series developed for mobile devices. The series emphasizes low power consumption and high performance for playing audio and video. It provides camera capabilities, web browsing, and LTE networking through an optional chipset.

1.2.1 Tegra K1

Announced in January of 2014, Tegra K1 is NVIDIA's latest iteration of its Tegra SoC series. It uses a "4-Plus-1" CPU design, pairing a quad-core ARM Cortex A15 with a fifth, low power core, allowing them to improve both performance and battery life. In addition to its upgraded processor, the Tegra K1 is the first of NVIDIA's Tegra SoCs to include one of their Kepler GPUs, a major step forward in graphics performance and computing on mobile devices. The inclusion of Kepler GPUs in the Tegra K1 will allow mobile applications to take advantage of the GPU's 192 cores for compute-intensive tasks [10].

1.2.2 SOC_THERM

SOC_THERM is a proprietary thermal management subsystem developed by NVIDIA, which is responsible for monitoring and managing the temperature of its Tegra chips. In addition to managing the chip's temperature, SOC_THERM also handles over-current interrupts, which are triggered when the chip is drawing too much current, causing the battery to supply insufficient voltage.

In order to prevent the chip from reaching excessive temperatures, SOC_THERM has multiple sensors monitoring different areas of the chip. There are four areas that SOC_THERM monitors: CPU, GPU, memory, and overall temperature (see Figure 1.1). When the temperature in one of these areas exceeds a certain threshold, SOC_THERM throttles the frequency of the CPU and/or GPU, slowing down the system. The magnitude of the throttling is determined by the chip's temperature. Slowing down the clock frequency reduces the chip's

heat output, which allows heat to dissipate faster. Once the chip cools down, it is gradually returned to its normal performance level.

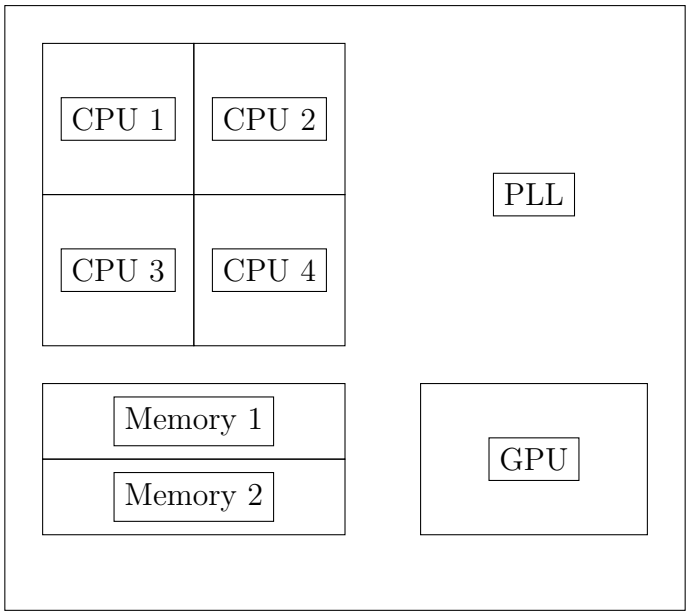

Figure 1.1: An illustration of the different thermal zones on a device. PLL represents the "overall temperature".

Over-current interrupts are handled the same way as overheating events. The only difference is that while the Tegra processor keeps an eye on temperature itself, over-current events are monitored externally by the Power Management Integrated Circuit or PMIC. When the PMIC notices that the chip is drawing too much current, it raises an interrupt to SOC_THERM , which SOC_THERM then handles. SOC_THERM is integrated into Android through the Linux thermal framework (see Section 1.5). This integration allows SOC_THERM to present the information that it has gathered to software in the user-space.

1.3 Static Analysis

Static program analysis is the assessment of software without actually running the program. Basic static analysis software reads through the program's code, or sometimes its object files, and tests it against a predefined set of rules. This is useful for ensuring syntactic and stylistic consistency within a body of code. Rules of inference are also used within static analysis to infer data types in an effort to catch type mismatching.

Advanced static analysis software uses rigorous, formal mathematical methods to prove axioms about a given program in order to prove its behavior.

Static analysis is highly useful within the code development process as it is a fast and efficient way of finding code inconsistencies and defects.

1.4 Device Tree

One of our tasks for this project was to convert hardware data hardcoded in the kernel into Device Tree format. Device Tree is a data structure used to organize and describe hardware. The structure is passed to the kernel during boot time instead of being hard-coded into the kernel. It is used by Open Firmware, an IEEE standard for boot firmware. A Device Tree source file (.dts) is tokenized into a stand-alone Flattened Device Tree (FDT). Converting the hard-coded data into Device Tree data is necessary because the board hardware on which SOC_THERM operates may vary. This variation must be accounted for by SOC_THERM, and may be expressed by using Device Tree format.

In order for this format to be useful, the operating system must be able to parse Device Tree files. "Bindings" are a form of documentation that describes how a hardware device's characteristics are structured in the Device Tree file. There are many predefined and documented bindings that can be found in the ePAPR and IEEE 1275 documentation [5]. SOC_THERM uses predefined thermal framework Device Tree bindings, therefore these are the ones we focused on[1].

The format of Device Tree is that of a tree with nodes and properties. Nodes may contain child nodes as well as properties. Properties are represented as key-value pairs; the value may be omitted. The following is an example of the structure used by the thermal framework:

Snippet 1.1: Device Tree example

```
1  / {
2          thermal-zones {
3                  tsense_cpu0: tsense_cpu0 {
4                          polling-delay = <0>;
5                          trips {
6                                  cpu0_crit: cpu0_crit {
7                                          temperature = <101000>;
8                                          type = "critical";
9                                  };
10                         };
11                         cooling-maps {
12                                 map_crit {
13                                         trip = <&cpu0_crit>;
```

[1]The file located in the Linux kernel source tree: Documentation/devicetree/bindings/thermal/thermal.txt

```
14                                          cooling-device = <&d_cdev THERMAL_NO_LIMIT
                                                THERMAL_NO_LIMIT>;
15                                  };
16                          };
17                  };
18          };
19  }
```

The / indicates a single root node. Line 2 of the example above is a node with the name `thermal-zones`. In accordance with the bindings, this node must be named `thermal-zones` so that it may be parsed properly. This node has one child node, `tsense_cpu0`, as seen on line 3. This node may be given an arbitrary name because the parser already knows that it is a thermal zone. It should be noted that any node may have any number of child nodes. The thermal zone `tsense_cpu0` has a number of properties and child nodes. The first property is seen in line 4. `polling-delay` is a cell-property with a value of 0. A cell-property may have any number of 32 bit unsigned integers inside angle brackets as long as they are separated by a space. For example, a cell property may look like: `polling-delay = <1 2 3 4>`. Line 5 is the beginning of a child node of `tsense_cpu0`. There may be multiple trip points, but in the example, there is only one: `cpu0_crit`. `cpu0_crit` has two properties, `temperature` and `type`. `type` is a string data type. Line 11 is another child node of `tsense_cpu0` that defines the mapping from `cooling-device` to `trips`. Line 13 defines the `trip` that the node is mapping from and line 14 defines the `cooling-device` that is being mapped to. In line 13, `&cpu0_crit` is called a phandle. This is a reference to the above `cpu0_crit` defined in the `trips` node starting at line 5. Line 14 uses a phandle as the first cell in the `cooling-device` property, and then the macro `THERMAL_NO_LIMIT` in the last two cells. There are many more property types that are not shown in the example above. This Device Tree format allows SOC_THERM to properly handle differences in hardware and not rely on hard-coded data.

1.5 Thermal Framework

Linux manages the system's temperature through its thermal framework. The framework includes thermal zones, thermal sensors, cooling devices, governors, trip points and thermal instances. The thermal framework also exposes information to user-space applications through *sysfs*, a virtual filesystem for device and driver information provided by Linux.

A thermal zone represents a physical region on a device. Generally, a thermal zone is associated with one or more of the following components: thermal sensor units, and cooling devices. A thermal sensor unit measures the temperature at its specific location and reports the temperature as the temperature of the zone to which it is bounded. A cooling device can

be classified as either hardware or software. Hardware cooling devices include fans and other physical devices that are designed to reduce the system's temperature. On the other hand, software cooling devices attempt to control system's temperature by manipulating the CPU and GPU's clock speed. In our paper, we will solely focus on software cooling devices. A thermal instance is created when a thermal zone is bounded to a cooling device at a specific trip point. A trip point specifies an action to be performed if a temperature threshold has been crossed. Trip points are located in within thermal instances.

1.6 of-thermal.c

The SOC_THERM driver does not have direct access to data manipulated by the thermal framework such as thermal zone structures. Instead, an API called of-thermal.c is responsible for this interaction as well as parsing the Device Tree file containing thermal zone data[2]. It is also used to register thermal zones outside of the sensor driver.

During initialization, the processes shown in 1.6 occur. First, machine initialization code calls a function in of-thermal.c (in 1.6 this is displayed as the line extending from of-thermal.c to Thermal Zone Device Tree) to parse the Device Tree file containing thermal zone data. of-thermal.c then creates and populates its own `__thermal_zone` structures which are internal representations of thermal zones. This can be seen as the dashed lined of 1.6 extending from of-thermal.c to __thermal_zone. These structures contain trip data, polling data, binding parameters, sensor ids, and function callbacks. The function callbacks are left blank; they are populated later on. Data parsed from the Device Tree file is then passed to the thermal framework (Thermal Core in 1.6) where thermal zone structures are registered (created and populated). In the below figure, this can be seen as the line extending from of-thermal.c to Thermal Core. Functions from of-thermal.c are passed as callback functions through this registration process. SOC_THERM calls an of-thermal.c registration function for each thermal sensor which adds callback functions to the `__thermal_zone` structs previously created. This step is shown in 1.6 as the line extending from Sensor Driver (SOC_THERM) to of-thermal.c. These functions are for getting sensor temperatures and trends.

1.7 Device Drivers and Platform Drivers

A driver is a piece of software that acts as a bridge between the operating system and the underlying device hardware. Drivers abstract hardware operations into a set of APIs

[2]The file located in the Linux kernel source tree: drivers/thermal/of-thermal.c

for applications or operating systems to utilize [4]. This allows programs to be written independently of the underlying hardware. Device drivers are discoverable by the kernel. A platform driver is a special type of driver for devices that cannot announce their presence to the kernel. Platform drivers provide a way for the kernel to probe for the existence of these devices [3].

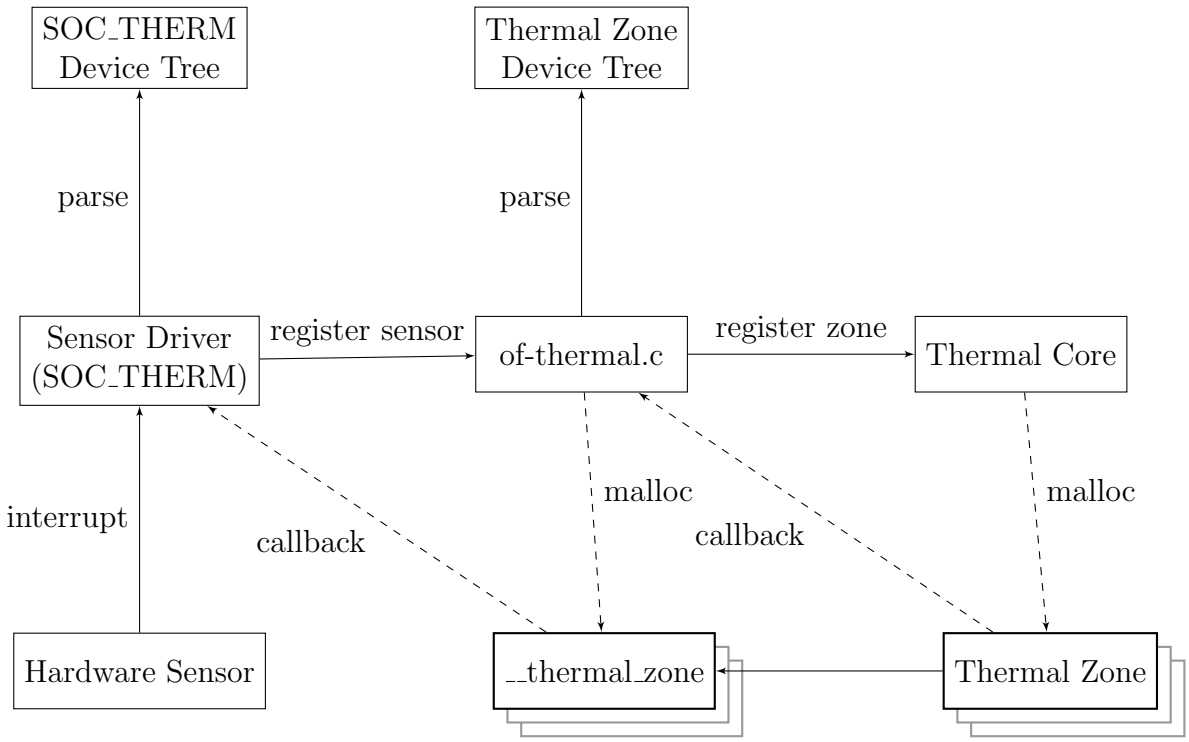

Figure 1.2: Diagram describing how SOC_THERM and the thermal framework interact via of-thermal.

2. Project Phases

Our mentors divided our project into multiple phases. The different phases were to acclimate us with NVIDIA code base, workflows, and kernel hacking in general.

2.1 Driver Documentation

Code documentation is an important process within software development. Documentation provides developers with an accurate understanding of the code in a succinct manner. Without proper documentation, valuable time is lost in trying to understand the code. However, in an environment in which there are very few developers trying to fulfill the demand of adding features and fixing bugs, developers often neglect to document their code sufficiently, if at all.

The first step of our project was to document NVIDIA's existing SOC_THERM library. This step accomplished two things. First, the SOC_THERM library gained a comprehensive set of documentation. Second, we would get a chance to understand the code we will be working with. The second component is very important for us because it created a solid foundation for us before we begin making functional changes to the driver code. As a side benefit, we also familiarized ourselves with version control and NVIDIA's rigorous code review process.

The standard that our group followed for documenting the SOC_THERM driver is called kernel-doc [2]. This format is an easy to maintain style of documentation for functions and data structures in the Linux kernel. Following this convention allows for consistency and readability throughout the kernel. This documentation is located within source files. The documentation blocks themselves are located adjacent to the function or data structure they describe. Documentation in the kernel-doc format is structured as follows: the first line of the comment must be `/**`. Comment blocks that are not in kernel-doc format should not start with `/**`. The kernel-doc formatted comment blocks generally ends with `*/`, but may end with `**/`. An example of a full comment block detailing a function named `cputemp_get` is below in Snippet 2.1. This function gets the temperature of the CPU from the physical

thermal sensor corresponding to the CPU and sets a pointer to this temperature. It takes in two parameters, data, and val.

Snippet 2.1: Kernel-doc for cputemp_get

```
1  /**
2   * cputemp_get() - gets the CPU temperature.
3   * @data:      not used
4   * @val:       a pointer in which the temperature will be placed
5   *
6   * Reads the temperature of the thermal sensor associated with the CPU.
7   *
8   * Return: 0 if successful
9   */
10  static int cputemp_get(void *data, u64 *val)
```

Line 1 opens the kernel-doc comment block. Line 2 gives the name of the function followed by an opened and closed parenthesis. This is followed by a short description of the function which may be multiple lines long. Lines 3 and 4 describe the functions parameters. Each argument must be preceded by an `@` symbol. In the example above, the argument `@data` is not used in the function. Lines 5 and 7 are blank lines, but must have `*` symbols regardless. Line 6 is a longer description of the function; it describes the function in more detail. This longer description can be multiple lines long. Line 8 describes what the function returns. Line 9 ends the kernel-doc comment block.

The documentation format of data structures is similar to that of functions, but omits a `Return` line. An example can be seen below in Snippet 2.2.

Snippet 2.2: Kernel-doc for a data structure

```
1  /**
2   * struct soctherm_platform_data - board-specific SOC_THERM info.
3   * @num_oc_irqs:      Number of over-current IRQs
4   * @tsensor_clk_rate: Clock rate for the thermal sensors.
5   */
6   struct soctherm_platform_data {
7         int num_oc_irqs;
8         unsigned long tsensor_clk_rate;
9   }
```

Line 2 gives the name of the structure followed by a short description of it. Lines 3 and 4 describe each of the structure's members. These descriptions may be multiple lines long. Line 5 closes the kernel-doc comment block. Data structures may have longer descriptions, just as functions. This longer description would be located after the structure's

member descriptions [2]. By following these conventions for functions and data structures, kernel-doc format provides consistency in documentation and increases the readability of the SOC_THERM driver code.

In order to provide meaningful documentation, we had to first truly understand what the code was doing. Unfortunately, reading through the code only showed us what each function did, but not what they meant within SOC_THERM as a whole. To bridge that gap, our mentors spent several hours teaching us not only SOC_THERM concepts, but kernel concepts as well. In addition to our mentors' help, we used stack tracing to figure out when each function is called while a device is running.

As a result of our documentation, we came away with a much clearer understanding of not only the SOC_THERM driver, but also a more thorough understanding of the Linux kernel as a whole. This new understanding became very useful throughout our subsequent projects. For NVIDIA, the benefit was two-fold. First, having a documented driver will significantly reduce time spent by other employees in trying to understand the code. Second, adding kernel-doc documentation brings the driver in-line with the upstream Linux kernel.

2.2 Driver Cleanup

We used multiple static analysis tools (see Section 1.3) to clean up the SOC_THERM driver. These tools included checkpatch, sparse, smatch, and cppcheck. This exercise accomplished two things. First, it aligned NVIDIA's driver code with the syntactic and stylistic guidelines of the upstream kernel, since upstream development has a strict set of guidelines in terms of documentation and code formatting. Second, the code cleanup accustomed us to the proper way to write code within the Linux kernel environment.

We ran the driver's source code through each tool, which provided us with warnings and errors. Following that feedback, we modified the code accordingly. Most of the warnings were simple to fix because they dealt with stylistic changes. Those that were not stylistic required careful inspection of the error message. We uploaded the changes for our mentors to review that our fixes were legitimate and not sweeping the issues under the rug. In the end, the code was clean and free from any obvious errors.

2.3 Platform Driver Conversion

When we began, the SOC_THERM driver was a library that did not follow the Linux kernel's standard for a platform driver. Similar to the code documentation, NVIDIA was incurring a lot of technical debt. Generally in software development, technical debt refers to shortcuts

taken during the development process that decrease the maintainability of a piece of software. NVIDIA took on this debt because other tasks were prioritized first. The engineers who worked on the driver do not only have to fix bugs, but are also responsible for adding supports for new chips. We helped NVIDIA to repay the debt by converting the SOC_THERM driver library into a proper platform driver.

2.3.1 Converting to SOC_THERM Driver

Since we documented the code, we were familiar with the SOC_THERM code. There are three major files that we need to modify: board files, board-power files, and the SOC_THERM library files. Each board file contains initialization functions for a particular NVIDIA board configuration. Similar to the board files, board-power files also contain power systems related initialization functions for a particular board. Lastly, the SOC_THERM driver files which contains all of SOC_THERM's functionalities.

First, we disabled the blocks of code that were originally responsible for initializing SOC_THERM. This allowed us to be sure that our code was initializing SOC_THERM. Once SOC_THERM was disabled, we gathered information about different components of the platform driver. Our major sources of information were from the Linux documentation for platform drivers[1] and the source code for Samsung's Exynos SoC's thermal driver[2].

For our driver to work, we needed to create at least two callback functions: probe and remove. The kernel uses probe as part of the driver binding process [3]. "Driver binding is the process of associating a device with a device driver that can control it" [1]. Each driver maintains a structure with a list of 'compatible' strings. Compatible strings are essentially names of devices that are compatible with a particular driver. Each device has a name, if this name matches a compatible string in a driver, the linking process begins. The binding process figures out which driver to bind which device to by pairing compatible strings to the device's name.

As its name suggests, the remove function is called when the kernel wants to remove the device from the system or unbind. The responsibility of the remove function is to clean up and release any memory allocated during the runtime of the device.

After we had some idea of what we needed to do, we identified which part(s) of the code needed to be changed to support the platform driver. We added the probe function and other relevant components into the library file and rearranged some of the existing initialization code to work with the Linux driver model. Then we moved onto the remove function. We

[1]The file located in the Linux kernel source tree: Documentation/driver-model/platform.txt
[2]The file located in the Linux kernel source tree: drivers/thermal/exynos_thermal.c

walked through the initialization process line-by-line to identify if we needed to undo any steps taken by the initialization process before the device could be unbound.

We were working on this phase in parallel with implementing of-thermal SOC_THERM (see Section 2.6). This meant that we had to hack together some code to test the functionality of the platform driver we created. The hack involved moving board-specific configuration information into SOC_THERM driver and its associated initialization functions. The hack has been flagged for future removal once the separation of SOC_THERM is completed.

We initially limited the scope of the conversion to one board since we only have one board to test on. After we finished the conversion, we submitted the code for review. Then we began adding support for all the boards that SOC_THERM supports. Since we did not physically have the boards, we had to test the code using NVIDIA's automated test system, which compiled our code and ran it on every board.

The last thing that we had to do to complete our library-to-driver conversion was to remove all instances of static variables. The use of static variables within the library meant that there could only ever be one instance of SOC_THERM running at a time. While it is unlikely that there will ever need to be more than one concurrent instance of SOC_THERM, allowing for that functionality helped to bring the driver into line with the upstream Linux kernel.

When the driver is first probed and initialized, it receives a pointer to a `platform_device` struct, from which it is able to gather all of its initialization data. Originally, the library took that data and stored it in a static variable to use throughout its lifetime. However, that meant that there could only ever be one concurrent instance of that variable. In order to fix this, we first identified the functions that were using this `plat_data` variable. Next, we had to modify each of those functions so that they took in an additional parameter, a `platform_device` struct. For functions that were solely called from within SOC_THERM, this was not much of a challenge. However, some of the functions that needed to be changed were used as callbacks from other parts of the kernel. As such, we couldn't change the prototypes for those functions. In order to get around that, we found ways to include our data into the parameters that were already being passed in. For example, one of the callback functions took in a `thermal_zone_device`, which has a field called `devdata`. Upon creation of our thermal zone devices, we were able to pass in our `platform_device` struct as devdata, which allowed us to access it in those callback functions. After modifying all of those functions so that they had a copy of the `platform_device` struct, we made sure that our probe function took the device-specific data that would be needed and stored it within the `platform_device`'s own `devdata` field. Finally, we replaced all instances of `plat_data` with the device-specific data that we stored within `platform_device`.

2.4 Device Tree Conversion of Hardware Data

Before we began our project work, hardware data was hard coded into C structures and interpreted by SOC_THERM , where it was then used to register thermal zones. This was a bad practice because it did not account for variability in hardware. Instead, we converted the hardware data for eight thermal sensors (four CPU, two MEM sensors, one GPU, and one PLLX sensor) into Device Tree format. We only converted these eight thermal sensors because they were not previously in use; converting the four sensor groups (which are in use) would take much more time and was not within the scope of our project.

In order to do this conversion, we needed to view the bindings documentation[3]. This file explained how the Device Tree file should be formatted for thermal zones. In accordance with the documentation, a node named `thermal-zones` must be created, with eight child-nodes. Each child-node represents a thermal zone. These child-nodes then have three properties and two child-nodes. The properties are as follows: `polling-delay`, `polling-delay-passive`, and `thermal-sensors`. `polling-delay` and `polling-delay-passive` are cell-properties (see Section 1.4). These two properties were given the value 0. `thermal-sensors` is a phandle (see Section 1.4) that is set to `&soctherm`, in reference to a node named `soctherm`, followed by a number indicating the sensors id. This id value is unique to each sensor within the `thermal-zones` node. The child-nodes that must be described are `trips` and `cooling-maps`. These child-nodes have properties of their own, but for the eight thermal sensors these nodes are left blank because the sensors do not have trip points or cooling devices associated with them. Once the conversion was completed, the thermal framework was able to parse and register these sensor zones. We confirmed that the new eight thermal zones showed up in sysfs.

2.5 User-space App Implementation

`regs_show()` is a function inside the SOC_THERM driver. It gathers various register data and system status. With this function, tegra11_soctherm.c exposes a debugfs file called "regs", allowing userspace to see a lot about the state of SOC_THERM hardware.

In computing, a hex dump is a hexadecimal view of the computer data. It is commonly used for debugging. Our task was to reduce the amount of code inside the SOC_THERM driver by moving the register parsing from the kernel space to userspace. In order to achieve this goal, we first changed the regs output to a hex dump. After changing regs into a hex dump, we modified a minimal but safe user-space app called soctherm_spy.c to parse the new

[3]File located in the Linux kernel source tree at: Documentation/devicetree/bindings/thermal/thermal.txt

format. It reads the hex dump generated by regs, stores the hexadecimal-formatted register data and system status into an array buffer, matches each element with the corresponding state of the SOC_THERM hardware, and prints them out.

2.6 of-thermal Implementation

The SOC_THERM driver was previously responsible for directly calling a registration function of thermal zones from the thermal framework. We were tasked with moving this registration to of-thermal (see Section 1.6 for details on of-thermal). In order to implement this change, first the registration had to be removed from SOC_THERM. This was done by removing the code which called the thermal framework registration function for each thermal sensor.

The next step was to create an expose function which calls the sensor registration function from of-thermal and passes in a number of parameters to it. These parameters include two callback functions and the SOC_THERM device which contains Device Tree nodes. The of-thermal registration function takes previously created `__thermal_zone` structs and populates their callback function fields. The rest of the fields have already been created during previous initializations. These callback functions were to get the sensor temperatures and to get thermal trends. They were written in a similar manner to pre-existing SOC_THERM functions, but were tailored to work specifically with of-thermal. The function which obtained the current thermal trend could not be written because the available data and architecture did not allow for a required field to be passed in.

In addition to having an expose function, a remove function needed to be written. This function would be called to undo the registration, such as when the SOC_THERM driver is unbound. The responsibility of this function is to unregister a given zone via a function in of-thermal. It is also responsible for freeing memory allocated in the expose function. In order to unregister thermal zones, the of-thermal unregister function takes a thermal zone struct. To make these structs accessible in SOC_THERM , we needed to store the registered thermal zone structs in an array where the index of the sensor corresponds to the index in the array.

Lastly, we were tasked with making trip temperatures accessible by the SOC_THERM sensor driver. Previously, SOC_THERM had access to this data inside the SOC_THERM code because it was stored in hard-coded C structures (see Section 2.4). With the conversion from hard coded hardware data to Device Tree format, this was no longer the case. These trip temperatures are needed in order to program hardware shutdown cases; that is, when a sensor reaches a critical temperature, the device should power off. This should be

programmed during initialization, therefore inside of the registration function in of-thermal, we passed in another callback function called `program_trip()`. This callback function takes two parameters: a pointer to a temperature and a thermal trip type (active, passive, hot, or critical). These parameters are obtained via of-thermal `of_thermal_get_trip_temp()` and `of_thermal_get_trip_type()` functions, which are then passed in to `program_trip()`. `program_trip()` checks to see if the given trip type is critical; if it is not, the function ends (only at critical trip points does the hardware shut down). It then calls another function which configures the hardware to shutdown the system if a given sensor reaches a given temperature. This configuration function could not be completed by our group because it was not within the scope of our project. Instead, the function prints out useful information about what was passed into it as well as what the next steps to finishing it are.

3. Results

Overall, our project was successful. We met our project goals and our code was going through NVIDIA's review process when we left the project site.

Driver Documentation

Our first project phase gave NVIDIA comprehensive documentation for their SOC_THERM library. This allows NVIDIAs employees and partners to spend more time on something that matters instead of trying to comprehend SOC_THERM code. Moreover, we walked away with a better understanding of how the monolithic Linux kernel works in conjunction with a device library. In addition to the documentation, SOC_THERM code is now aligned with upstream kernel coding guidelines. We ran multiple static analysis tools to make sure that SOC_THERM is free of any stylistic issues and obvious errors that may have been overlooked.

Platform Driver Conversion

The platform driver conversion project consisted of three main objectives. Each objective was followed by a code review phase, which provided valuable feedback on both stylistic and syntactic changes that needed to be made. Each code review phase iterated until all parties involved were satisfied.

The first main objective of our project was to set up platform driver support for the boards with which we were working. We removed the code that the board files used to start up SOC_THERM and instead initialized it via Device Tree. We ran tests so that we could be assured that the platform driver version of SOC_THERM functioned in the exact same way as the library version.

The second main objective of our project was to add support for all of the other boards with Tegra 4 and K1 chips. Since we removed the code that all of the board files used to start up SOC_THERM , we could also remove the initialization function within SOC_THERM that all of the board files were calling.

The third main objective of our project was to remove static variables from the newly converted driver so that it could handle more than one instance of SOC_THERM hardware at a time. We placed those static variables within the `platform_device` struct and converted all of SOC_THERM's functions so that they also took in a pointer to that struct. This allowed us to pass the information around without storing it into a global variable.

Device Tree Conversion of Hardware Data

The Device Tree conversion of hardware data added flexibility to the thermal management system. Before this implementation, the SOC_THERM driver was directly working with board hardware data within hardcoded C structures. If the hardware that the sensor driver functioned on was changed, the sensor driver would then have to be modified to accommodate this change. With the data stored in Device Tree files, this modification is no longer necessary. During initialization, the Device Tree file containing the appropriate hardware data is parsed and then used to populate the C structures which the thermal framework then manages.

User-space App Implementation

The userspace app implementation consisted of two main objectives. The first main objective was to convert the "regs" output from SOC_THERM debugfs into a hex dump. We modified the code of `regs_show()` inside SOC_THERM driver. The second objective was to modify a userspace app to parse the hex dump from "regs". We implemented a minimal but safe app called soctherm_spy.c. The overall goal of this task was to reduce the amount of code in SOC_THERM kernel space but to keep the same information in the userspace.

of-thermal Implementation

Reworking the SOC_THERM driver moved the responsibility of thermal zone structure creation from SOC_THERM to the thermal framework via of-thermal. Preventing the sensor driver from creating thermal zones was necessary because this is handled during initializations by of-thermal. Instead, the sensor driver adds callback functions to the pre-existing of-thermal thermal zone structures and the thermal framework structures via of-thermal.

References

[1] Driver Binding. https://www.kernel.org/doc/Documentation/driver-model/binding.txt.

[2] kernel-doc nano-howto. https://www.kernel.org/doc/Documentation/kernel-doc-nano-HOWTO.txt.

[3] The Platform Device API. http://lwn.net/Articles/448499/.

[4] What Is A Device Driver. https://www.pc-gesund.de/it-wissen/what-is-a-device-driver.

[5] Power.org™ Standard For Embedded Power Architecture™ Platform Requirements (ePAPR) v1.1. http://www.power.org/documentation/epapr-version-1-1/, April 2011.

[6] NVIDIA. Company history. http://www.nvidia.com/page/corporate_timeline.html.

[7] NVIDIA. GeForce. http://www.geforce.com.

[8] NVIDIA. High Performance Pomputing and Supercomputing. http://www.nvidia.com/object/tesla-supercomputing-solutions.html.

[9] NVIDIA. Tesla GPUs Power World's Fastest Supercomputer. http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&prid=678988&releasejsp=release_157.

[10] NVIDIA. The Tegra K1 Supercomputing Mobile Processor. http://www.nvidia.com/object/tegra-k1.html.

[11] NVIDIA. The Visual Computing Company. http://www.nvidia.com/object/visual-computing.html.

[12] NVIDIA. Workstation Solutions. http://www.nvidia.com/object/workstation-solutions.html.