The HotSniper User Manual

Martin Rapp and Anuj Pathania

July 1, 2021

This guide builds on top of the Sniper multi-core user manual available here.

1 Write a Custom Resource Management Policy

This guide provides step-by-step instructions to implement a new policy. We take the Linux ondemand governor as an example for a DVFS policy.

1. Create the files `dvfsOndemand.cc` and `dvfsOndemand.h` in the folder `common/scheduler/policies`.

Our policy requires access to the performance counters, as well as several configurable parameters. We provide them to the policy in the constructor. We implemet the `DVFSPolicy` interface that requires a function `getFrequencies` as indicated below. Finally, we define some class member variables to hold some internal state.

2. Change the code of `dvfsOndemand.h` as listed in Appendix A.1.

Next, we provide the actual implementation of our policy. The policy is queried periodically (method `getFrequencies`). This function returns a new vector of per-core frequencies to be used.

3. Change the code of `dvfsOndemand.cc` as listed in Appendix A.2.

We defined several parameters for our policy that we would like to expose in the configuration files. In the next step, we add them there and provide values. Using namespaces (e.g.: `[scheduler/open/dvfs/ondemand]` is optional but helps structuring the code better.

4. Add the following code to `config/base.cfg`:

```
[scheduler/open/dvfs/ondemand]
up_threshold = 0.7
down_threshold = 0.3
dtm_cricital_temperature = 80
dtm_recovered_temperature = 78
```

We are now ready to instantiate our policy.

5. Include our new header file in `common/scheduler/scheduler_open.cc`:

```
#include "policies/dvfsOndemand.h"
```

6. Extend the method `initDVFSPolicy` in `common/scheduler/scheduler_open.cc`:

```
} else if (policyName == "ondemand") {
        float upThreshold = Sim()->getCfg()->getFloat("scheduler/open/dvfs/
            ondemand/up_threshold");
        float downThreshold = Sim()->getCfg()->getFloat("scheduler/open/dvfs/
            ondemand/down_threshold");
        float dtmCriticalTemperature = Sim()->getCfg()->getFloat("scheduler/open/
            dvfs/ondemand/dtm_cricital_temperature");
        float dtmRecoveredTemperature = Sim()->getCfg()->getFloat("scheduler/open
            /dvfs/ondemand/dtm_recovered_temperature");
        dvfsPolicy = new DVFSOndemand(
                performanceCounters,
                coreRows,
                coreColumns,
                minFrequency,
                maxFrequency,
                frequencyStepSize,
                upThreshold,
                downThreshold,
                dtmCriticalTemperature,
                dtmRecoveredTemperature
        );
} //else if (policyName ="XYZ") {... } //Place to instantiate a new DVFS logic.
    Implementation is put in "policies" package.
```

This code checks whether the policy with the name `ondemand` shall be used. If this is the case, it first reads the four configuration options and then instantiates our policy.

Finally, we need to select our new policy in the config.

7. Perform the following change in `config/base.cfg`:

```
[scheduler/open/dvfs]
logic = ondemand
```

Finally, rebuild HotSniper.

8. Execute `make` in the base folder.

1.1 Notes

- Policies for task mapping can be created similarly. The interface to be implemented is `MappingPolicy`. The policy must be instantiated in `initMappingPolicy`. The new policy is selected in the config in `scheduler/open/logic` for mapping.
- It is perfectly fine to implement several interfaces in one policy. For example, a resource management could do joint mapping and frequency selection by implementing both the respective interfaces in one class.

A Code

A.1 `dvfsOndemand.h`

```
/**
 * This header implements the ondemand governor with DTM.
 * The ondemand governor implementation is based on
 * Pallipadi, Venkatesh, and Alexey Starikovskiy.
 * "The ondemand governor."
 * Proceedings of the Linux Symposium. Vol. 2. No. 00216. 2006.
 */

#ifndef __DVFS_ONDEMAND_H
#define __DVFS_ONDEMAND_H

#include <vector>
#include "dvfspolicy.h"
#include "performance_counters.h"

class DVFSOndemand : public DVFSPolicy {
public:
    DVFSOndemand(
        const PerformanceCounters *performanceCounters,
        int coreRows,
        int coreColumns,
        int minFrequency,
        int maxFrequency,
        int frequencyStepSize,
        float upThreshold,
        float downThreshold,
        float dtmCriticalTemperature,
        float dtmRecoveredTemperature);
    virtual std::vector<int> getFrequencies(const std::vector<int> &
        oldFrequencies, const std::vector<bool> &activeCores);

private:
    const PerformanceCounters *performanceCounters;

    unsigned int coreRows;
    unsigned int coreColumns;
    int minFrequency;
    int maxFrequency;
    int frequencyStepSize;
    float upThreshold;
    float downThreshold;
    float dtmCriticalTemperature;
    float dtmRecoveredTemperature;

    bool in_throttle_mode = false;
    bool throttle();
};

#endif
```

A.2 dvfsOndemand.cc

```
#include "dvfsOndemand.h"
#include <iomanip>
#include <iostream>

using namespace std;
```

```
DVFSOndemand::DVFSOndemand(
        const PerformanceCounters *performanceCounters,
        int coreRows,
        int coreColumns,
        int minFrequency,
        int maxFrequency,
        int frequencyStepSize,
        float upThreshold,
        float downThreshold,
        float dtmCriticalTemperature,
        float dtmRecoveredTemperature)
        : performanceCounters(performanceCounters),
      coreRows(coreRows),
      coreColumns(coreColumns),
      minFrequency(minFrequency),
      maxFrequency(maxFrequency),
      frequencyStepSize(frequencyStepSize),
      upThreshold(upThreshold),
      downThreshold(downThreshold),
      dtmCriticalTemperature(dtmCriticalTemperature),
      dtmRecoveredTemperature(dtmRecoveredTemperature) {

}

std::vector<int> DVFSOndemand::getFrequencies(const std::vector<int> &
    oldFrequencies, const std::vector<bool> &activeCores) {
    if (throttle()) {
        std::vector<int> minFrequencies(coreRows * coreColumns, minFrequency);
        cout << "[Scheduler][ondemand-DTM]: in throttle mode -> return min.
            frequencies" << endl;
        return minFrequencies;
    } else {
        std::vector<int> frequencies(coreRows * coreColumns);

        for (unsigned int coreCounter = 0; coreCounter < coreRows * coreColumns;
            coreCounter++) {
            if (activeCores.at(coreCounter)) {
                float power = performanceCounters->getPowerOfCore(coreCounter);
                float temperature = performanceCounters->getTemperatureOfCore(
                    coreCounter);
                int frequency = oldFrequencies.at(coreCounter);
                float utilization = performanceCounters->getUtilizationOfCore(
                    coreCounter);

                cout << "[Scheduler][ondemand]: Core " << setw(2) << coreCounter
                    << ":";
                cout << " P=" << fixed << setprecision(3) << power << " W";
                cout << " f=" << frequency << " MHz";
                cout << " T=" << fixed << setprecision(1) << temperature << " C";
                    // avoid the little circle symbol, it is not ASCII
                cout << " utilization=" << fixed << setprecision(3) << utilization
                     << endl;
                // use same period for upscaling and downscaling as described in "
                    The ondemand governor."
                if (utilization > upThreshold) {
                    cout << "[Scheduler][ondemand]: utilization > upThreshold";
                    if (frequency == maxFrequency) {
                        cout << " but already at max frequency" << endl;
                    } else {
                        cout << " -> go to max frequency" << endl;
                        frequency = maxFrequency;
                    }
                } else if (utilization < downThreshold) {
                    cout << "[Scheduler][ondemand]: utilization < downThreshold";
                    if (frequency == minFrequency) {
                        cout << " but already at min frequency" << endl;
                    } else {
                        cout << " -> lower frequency" << endl;
                        frequency = frequency * 80 / 100;
                        frequency = (frequency / frequencyStepSize) *
                            frequencyStepSize; // round
                        if (frequency < minFrequency) {
                            frequency = minFrequency;
                        }
                    }
                }

                frequencies.at(coreCounter) = frequency;
            } else {
                frequencies.at(coreCounter) = minFrequency;
            }
        }

            return frequencies;
    }
}

bool DVFSOndemand::throttle() {
    if (performanceCounters->getPeakTemperature() > dtmCriticalTemperature) {
        if (!in_throttle_mode) {
            cout << "[Scheduler][ondemand-DTM]: detected thermal violation" <<
                endl;
        }
        in_throttle_mode = true;
    } else if (performanceCounters->getPeakTemperature() <
        dtmRecoveredTemperature) {
        if (in_throttle_mode) {
            cout << "[Scheduler][ondemand-DTM]: thermal violation ended" << endl;
        }
        in_throttle_mode = false;
    }
    return in_throttle_mode;
}
```