The HotSniper User Manual

June 29, 2023

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

1 Write a Custom DVFS Policy

This guide provides step-by-step instructions to implement a new policy. We take the Linux ondemand governor as an example of a DVFS policy. To showcase the use of integrated thermal feedback, we implement a reactive Dynamic Thermal Management (DTM) that responds to peak core temperatures.

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 implement 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 structure 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.

2 Write a Custom Task Mapping and Migration Policy

This section describes how to create a thermal-aware task mapping and migration policy. When a new application arrives, we want to map it to the coldest cores. When cores get too hot, we want to migrate the threads away from them to the coldest cores. This policy also serves as an example of how to build a policy that uses several knobs (mapping and migration) simultaneously.

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

As this policy should perform both task mapping and migration, we implement both interfaces `MappingPolicy` and `MigrationPolicy`. If a policy should also do DVFS, additionally implement the `DVFSPolicy` as in Section 1. The open scheduler calls the `map` function whenever a new application needs to be mapped, and periodically calls the `migrate` function.

2. Change the code of `coldestCore.h` as listed in Appendix A.3.

We now implement the logic of these functions. `map` needs to return the list of cores for this application (in the one-thread-per-core model). `migration` can return an arbitrarily long list of thread migrations to be performed. Each migration needs to specify the source and target core of the migration. The `swap` attribute is used to indicate that both source and target core are used, and the corresponding threads should be swapped. While this information would also be present in the scheduler, the requirement to explicitly mark swap operations serves to detect bugs in the migration policy early.

3. Change the code of `coldestCore.cc` as listed in Appendix A.4.

Our policy migrates threads away from hot cores when the temperature has exceeded a certain threshold. This threshold is passed to the constructor. We want to have this value easily configurable. First, we add the value to a new config section for our policy. Notice that adding a new section is optional, but maintains the readability of the config.

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

```
[scheduler/open/migration/coldestCore]
criticalTemperature = 80
```

We are now ready to instantiate our policy.

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

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

As we perform both mapping and migration, we need to add our policy to two places.

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

```
} else if (policyName == "coldestCore") {
       float criticalTemperature = Sim()->getCfg()->getFloat(
              "scheduler/open/migration/coldestCore/criticalTemperature");
       mappingPolicy = new ColdestCore(performanceCounters, coreRows,
           coreColumns, criticalTemperature);
} //else if (policyName ="XYZ") {... } //Place to instantiate a new mapping
    logic. Implementation is put in "policies" package.
```

7. Extend the method `initMigrationPolicy` in `common/scheduler/scheduler_open.cc`:

```
} else if (policyName == "coldestCore") {
       float criticalTemperature = Sim()->getCfg()->getFloat(
              "scheduler/open/migration/coldestCore/criticalTemperature");
       migrationPolicy = new ColdestCore(performanceCounters, coreRows,
           coreColumns, criticalTemperature);
} //else if (policyName ="XYZ") {... } //Place to instantiate a new migration
    logic. Implementation is put in "policies" package.
```

Now, we are ready to select our new policy for both mapping and migration.

8. Perform the following changes in `config/base.cfg`:

```
[scheduler/open]
logic = coldestCore

[scheduler/open/migration]
logic = coldestCore
```

Finally, rebuild HotSniper.

9. Execute `make` in the base folder.

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;
}
```

A.3 coldestCore.h

```
/**
 * This header implements a policy that maps new applications to the coldest
     core
 * and migrates threads from hot cores to the coldest cores.
 */

#ifndef __COLDESTCORE_H
#define __COLDESTCORE_H

#include <vector>
#include "mappingpolicy.h"
#include "migrationpolicy.h"
#include "performance_counters.h"

class ColdestCore : public MappingPolicy, public MigrationPolicy {
public:
    ColdestCore(
        const PerformanceCounters *performanceCounters,
        int coreRows,
        int coreColumns,
        float criticalTemperature);
    virtual std::vector<int> map(
        String taskName,
        int taskCoreRequirement,
        const std::vector<bool> &availableCores,
        const std::vector<bool> &activeCores);
    virtual std::vector<migration> migrate(
        SubsecondTime time,
        const std::vector<int> &taskIds,
        const std::vector<bool> &activeCores);

private:
    const PerformanceCounters *performanceCounters;

    unsigned int coreRows;
    unsigned int coreColumns;
    float criticalTemperature;

    int getColdestCore(const std::vector<bool> &availableCores);
    void logTemperatures(const std::vector<bool> &availableCores);
};

#endif
```

A.4 coldestCore.cc

```
#include "coldestCore.h"

#include <iomanip>

using namespace std;

ColdestCore::ColdestCore(
        const PerformanceCounters *performanceCounters,
        int coreRows,
        int coreColumns,
        float criticalTemperature)
    : performanceCounters(performanceCounters),
      coreRows(coreRows),
      coreColumns(coreColumns),
      criticalTemperature(criticalTemperature) {

}

std::vector<int> ColdestCore::map(
        String taskName,
        int taskCoreRequirement,
        const std::vector<bool> &availableCoresRO,
        const std::vector<bool> &activeCores) {

    std::vector<bool> availableCores(availableCoresRO);

    std::vector<int> cores;

    logTemperatures(availableCores);

    for (; taskCoreRequirement > 0; taskCoreRequirement--) {
        int coldestCore = getColdestCore(availableCores);

        if (coldestCore == -1) {
            // not enough free cores
            std::vector<int> empty;
            return empty;
        } else {
            cores.push_back(coldestCore);
            availableCores.at(coldestCore) = false;
        }
    }

    return cores;
}

std::vector<migration> ColdestCore::migrate(
        SubsecondTime time,
        const std::vector<int> &taskIds,
        const std::vector<bool> &activeCores) {

    std::vector<migration> migrations;
```

```
    std::vector<bool> availableCores(coreRows * coreColumns);
    for (int c = 0; c < coreRows * coreColumns; c++) {
        availableCores.at(c) = taskIds.at(c) == -1;
    }

    for (int c = 0; c < coreRows * coreColumns; c++) {
        if (activeCores.at(c)) {
            float temperature = performanceCounters->getTemperatureOfCore(c);
            if (temperature > criticalTemperature) {
                cout << "[Scheduler][coldestCore-migrate]: core" << c << " too hot
                     (";
                cout << fixed << setprecision(1) << temperature << ") -> migrate";
                logTemperatures(availableCores);

                int targetCore = getColdestCore(availableCores);

                if (targetCore == -1) {
                    cout << "[Scheduler][coldestCore-migrate]: no target core
                        found, cannot migrate" << endl;
                } else {
                    migration m;
                    m.fromCore = c;
                    m.toCore = targetCore;
                    m.swap = false;
                    migrations.push_back(m);
                    availableCores.at(targetCore) = false;
                }
            }
        }
    }

    return migrations;
}

int ColdestCore::getColdestCore(const std::vector<bool> &availableCores) {
    int coldestCore = -1;
    float coldestTemperature = 0;
    // iterate all cores to find coldest
    for (int c = 0; c < coreRows * coreColumns; c++) {
        if (availableCores.at(c)) {
            float temperature = performanceCounters->getTemperatureOfCore(c);
            if ((coldestCore == -1) || (temperature < coldestTemperature)) {
                coldestCore = c;
                coldestTemperature = temperature;
            }
        }
    }
    return coldestCore;
}

void ColdestCore::logTemperatures(const std::vector<bool> &availableCores) {
    cout << "[Scheduler][coldestCore-map]: temperatures of available cores:" <<
        endl;
```

```
    for (int y = 0; y < coreRows; y++) {
        for (int x = 0; x < coreColumns; x++) {
            if (x > 0) {
                cout << " ";
            }
            int coreId = y * coreColumns + x;

            if (!availableCores.at(coreId)) {
                cout << " - ";
            } else {
                float temperature = performanceCounters->getTemperatureOfCore(
                    coreId);
                cout << fixed << setprecision(1) << temperature;
            }
        }
        cout << endl;
    }
}
```