دانشکده‌ی مهندسی کامپیوتر

فرزام کوهی رونقی - ۴۰۱۱۰۶۴۰۳ | ثنا بابایان ونستان - ۴۰۱۱۰۵۶۸۹

مستند پروژه سیستم‌های نهفته

۱ مقدمه

تکنیک DVFS (Dynamic Voltage and Frequency Scaling)، یکی از تکنیک‌هایی است که برای مدیریت توان مصرفی یک سامانه به صورت پویا استفاده می‌شود. در این تکنیک با مقداردهی پویای فرکانس کاری پردازنده‌ها با توجه به موعد زمانی وظیفه‌ها، می‌توانیم توان مصرفی سیستم را مدیریت و بهینه کنیم. از این تکنیک می‌توان برای مدیریت توان مصرفی در سیستم‌های چند هسته‌ای استفاده کرد.

۲ تعریف مسئله

در این پروژه قصد داریم که توان مصرفی کلی در یک سیستم چند هسته، مانند Raspberry Pi را با استفاده از تکنیک های مدیریت توان مصرفی مانند ،DVFS بهینه کنیم. بدین منظور با بررسی میزان توان مصرفی برنامه‌های بنچمارک MiBench یا Parsec و پروفایل توان مصرفی هر یک از برنامه‌ها، می‌خواهیم به هر یک از تسک‌ها یک موعد زمانی اختصاص دهیم تا بتوانیم یک سامانه بی‌درنگ را شبیه‌سازی کنیم. همچنین با تخصیص برنامه‌ها با میزان توان مصرفی بالاتر به هسته‌ها با فرکانس کاری بالاتر، و استفاده از تکنیک DVFS (Dynamic Voltage and Frequency، بتوانیم مصرف انرژی سامانه را بهینه کنیم.

۳ اهمیت موضوع

یکی از نیازمندی‌های بسیاری از سامانه‌های نهفته، توان مصرفی پایین است. بنابراین می‌توان گفت در سامانه‌های نهفته، که بخشی از آن‌ها به صورت سامانه‌های چند هسته‌ای هستند، مدیریت توان مصرفی از اهمیت بالایی برخوردار است.

۴ گام‌های پروژه

در ادامه مراحلی را که در این پروژه قصد داریم طی کنیم، نام می‌بریم.

۱. راه اندازی Raspberry Pi

۲. اجرای وظایف یکی از دو بنچمارک MiBench و Parsec

۳. بررسی محدوده فرکانس مجاز هر هسته از Raspberry Pi و تعیین ۶ سطح ولتاژی با توزیع نرمال

۴. اجرای وظایف بنچمارک‌ها بر روی هسته‌های سیستم و پروفایل زمان اجرا و توان مصرفی آن‌ها

۵. نوشتن برنامه‌ای که با گرفتن وظایف و موعد زمانی آنها، با استفاده از تکنیک DVFS، آن‌ها را روی هسته‌های سیستم برنامه ریزی کند. این برنامه باید از EDF برای برنامه ریزی تسک‌ها استفاده کند.

۶. بررسی و پروفایل توان مصرفی سیستم بعد از استفاده از برنامه فوق

۷. مقایسه و تحلیل نتایج بدست آمده از گام قبل و نتایج بدون استفاده از DVFS

۵ گزارش پیاده سازی

۱.۵ اجرای برخی وظایف بنچمارک MiBench

حال گام‌های ذکر شده را به ترتیب طی می‌کنیم. راه اندازی Raspberry Pi که چالش خاصی ندارد و روی آن، از سیستم عامل Raspberry Pi OS استفاده می‌کنیم.

از میان دو بنچمارک گفته شده، ما از بنچمارک MiBench استفاده خواهیم کرد. بنابراین در ابتدا تعدادی از تسک‌های آن را روی رزبری پای اجرا می‌کنیم.

سورس کد بنچمارک MiBench در این لینک موجود است. برنامه‌های آن از چند دسته automotive, consumer, network, office, security, telecomm تشکیل شده است که جدول زیر نمایی کلی از آنها را نشان می‌دهد.

Auto./Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	GSM dec.
	typeset				

شکل ۱: Mibench

برای مثال، تعدادی از تسک‌ها را با ورودی نمونه خودش اجرا می‌کنیم. تصاویز زیر مربوط به qsort از دسته automotive و نسخه large آن است:

```
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $ ls
COMPILE          input_small.dat  Makefile          output_small.txt  qsort_large.c  qsort_small.c   runme_small.sh
input_large.dat  LICENSE          output_large.txt  qsort_large       qsort_small    runme_large.sh
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $ cat qsort_large.c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define UNLIMIT
#define MAXARRAY 60000 /* this number, if too large, will cause a seg. fault!! */

struct my3DVertexStruct {
  int x, y, z;
  double distance;
};

int compare(const void *elem1, const void *elem2)
{
  /* D = [(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2]^(1/2) */
  /* sort based on distances from the origin... */

  double distance1, distance2;

  distance1 = (*((struct my3DVertexStruct *)elem1)).distance;
  distance2 = (*((struct my3DVertexStruct *)elem2)).distance;

  return (distance1 > distance2) ? 1 : ((distance1 == distance2) ? 0 : -1);
}


int
main(int argc, char *argv[]) {
  struct my3DVertexStruct array[MAXARRAY];
  FILE *fp;
  int i,count=0;
  int x, y, z;

  if (argc<2) {
    fprintf(stderr,"Usage: qsort_large <file>\n");
```

شکل ۲: بخش ابتدایی کد qsort_large.c

```
  double distance1, distance2;

  distance1 = (*((struct my3DVertexStruct *)elem1)).distance;
  distance2 = (*((struct my3DVertexStruct *)elem2)).distance;

  return (distance1 > distance2) ? 1 : ((distance1 == distance2) ? 0 : -1);
}


int
main(int argc, char *argv[]) {
  struct my3DVertexStruct array[MAXARRAY];
  FILE *fp;
  int i,count=0;
  int x, y, z;

  if (argc<2) {
    fprintf(stderr,"Usage: qsort_large <file>\n");
    exit(-1);
  }
  else {
    fp = fopen(argv[1],"r");

    while((fscanf(fp, "%d", &x) == 1) && (fscanf(fp, "%d", &y) == 1) && (fscanf(fp, "%d", &z) == 1) &&  (count < MAXARRA
        array[count].x = x;
        array[count].y = y;
        array[count].z = z;
        array[count].distance = sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2));
        count++;
    }
  }
  printf("\nSorting %d vectors based on distance from the origin.\n\n",count);
  qsort(array,count,sizeof(struct my3DVertexStruct),compare);

  for(i=0;i<count;i++)
    printf("%d %d %d\n", array[i].x, array[i].y, array[i].z);
  return 0;
}
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $
```

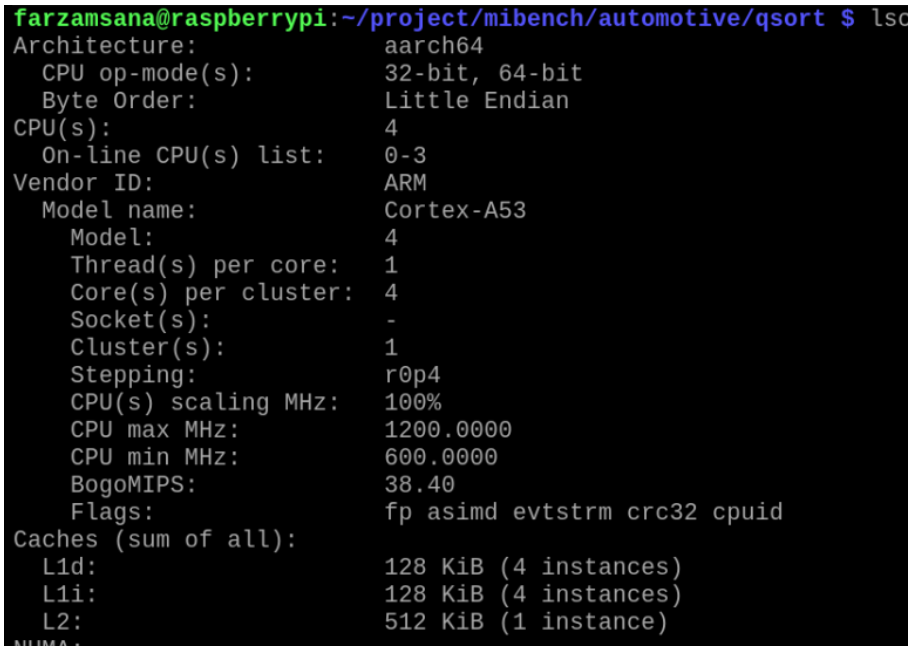
شکل ۳: بخش دوم کد

```
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $ ./runme_large.sh
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $ ls
COMPILE          input_small.dat  Makefile          output_small.txt  qsort_large.c  qsort_small.c   runme_small.sh
input_large.dat  LICENSE          output_large.txt  qsort_large       qsort_small    runme_large.sh
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $ head output_large.txt

Sorting 50000 vectors based on distance from the origin.

25138398 28611231 9838998
3188060 13891849 39584992
7746427 100619473 40496197
98214313 68449046 35269744
38686879 83001081 100587830
3230455 134784516 23406264
93924301 82680824 57298071
farzamsana@raspberrypi:~/project/mibench/automotive/qsort $
```

شکل ۴: نتیجه خروجی runme_large.sh

۲.۵ بررسی محدوده فرکانس هسته‌های رزبری پای

با استفاده از دستور lscpu، متوجه شدیم که رزبری پای چهار هسته پردازشی دارد که محدوده فرکانس آنها بین 600MHz تا 1200MHz می‌باشد.



شکل ۵: lscpu

طبق خواسته پروژه، باید نتایج توان مصرفی را برای ۶ سطح فرکانس (ولتاژ) انجام دهیم. سطوح انتخابی ما با توجه به بازه $[600 - 1200]MHz$، مقادیر ۶۰۰، ۷۲۰، ۸۴۰، ۹۶۰، ۱۰۸۰ و ۱۲۰۰ قرار بود باشد، اما نکته‌ای که هست این است که امکان جابجایی فرکانسی با مقدار ۱۰۰ مگاهرتز وجود دارد و یعنی فقط به مقادیر 600, 700, 800, 900, 1000, 1100, 1200 دسترسی خواهیم داشت. بنابراین ما کمی فرضیات پروژه را تغییر داده و با این ۷ سطح کار می‌کنیم.

۳.۵ یافتن ابزاری برای پروفایل توان مصرفی سیستم

برای محاسبه پروفایل توان مصرفی سیستم، به دنبال ابزارهای بسیاری گشتیم. متاسفانه با جستجو در نت متوجه شدیم که راه نرم افزاری مناسبی برای این کار وجود ندارد و ناچار باید از قطعات سخت‌افزاری استفاده کنیم. از جمله ابزارهایی که به آن اشاره می‌شود، که در نهایت ما این کار را با استفاده از وسیله‌ای به نام USB Power Meter می باشد.

شکل ۶: نمایی از USB Power Meter

در ادامه یک سری از روش‌هایی را که امتحان کردیم ولی نتیجه مناسبی به دست نیاوردیم را می آوریم:

- استفاده از دستور :powerstats این دستور که در مستند توضیحات پروژه نیز قرار گرفته بود، برای اندازه گیری توان مصرفی سیستم استفاده می‌شود، اما مشکل این است که برای Raspberry Pi کار نمی‌کند.

```
farzamsana@raspberrypi:~ $ powerstat
Device is not discharging, cannot measure power usage.
Perhaps re-run with -z (ignore zero power)
farzamsana@raspberrypi:~ $ powerstat -z
Running for 480.0 seconds (48 samples at 10.0 second intervals).
Power measurements will start in 0 seconds time.

  Time    User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts
07:58:19   7.5   0.0   1.1  91.4   0.1    2    559   8876   0.00E
```

شکل ۷: اجرای powerstat روی رزبری پای، همانطور که مشاهده می‌شود، مدام zero power می‌دهد.

- یک ابزار نرم افزاری اندازه‌گیری توان PowerAPI می باشد که فقط برای هسته‌های ARM پاسخگو نبود.

> Please notice that you need a **Linux distribution** in order to use the HWPC Sensor installed by the script as well as a **comptible Intel** (Sandy Bridge and newer) or **AMD Processor** (Zen). You also need docker. **Power/ARM/RISCV are not supported** architectures. HWPC Sensor will **not work on a Virtual Machine**. However, you can install the Formula by hand in a Virtual Machine if need it.

شکل ۸: کارنکردن PowerAPI روی هسته‌های ARM

- رجوع به پوشه sys/class/power_supply، این پوشه دارای فایل‌هایی برای نمایش جریان و ولتاژ کاری سیستم است، اما این پوشه در رزبری پای خالی است.
- با استفاده از دستور vcgencmd صرفا توانستیم ولتاژ کاری سیستم را به دست آوریم نه توان سیستم را.

شکل ۹: اندازه گیری با استفاده از USB Power Meter

۴.۵ پروفایل توان مصرفی وظایف بنچمارک

حال، تعدادی از وظایف موجود در بنچمارک را به طور مستقل روی رزبری پای به ازای ۷ سطح فرکانسی گفته شده اجرا کرده و زمان اجرا و توان مصرفی آن را اندازه گیری می‌کنیم.

بدین منظور برای پروفایل کردن توان مصرفی، بش اسکریپت زیر را نوشتیم.

```
#!/bin/bash

if [ $# -ne 2 ]; then
    echo "Usage: $0 <script_name> <number_of_runs>"
    exit 1
fi

SCRIPT_NAME=$1
NUM_RUNS=$2

FREQ_LEVELS=(600 720 840 960 1080 1200)

for FREQ in "${FREQ_LEVELS[@]}"; do
    echo "Setting CPU frequency to $FREQ MHz"
    sudo cpufreq-set -c 0 -f ${FREQ}Mhz
    sudo cpufreq-set -c 1 -f ${FREQ}Mhz
    sudo cpufreq-set -c 2 -f ${FREQ}Mhz
    sudo cpufreq-set -c 3 -f ${FREQ}Mhz


    start_time=$(date +%s.%N)
```

```
22
23      for ((i=1; i<=NUM_RUNS; i++)); do
24          ./"$SCRIPT_NAME"
25      done
26
27      end_time=$(date +%s.%N)
28
29      runtime=$(echo "$end_time - $start_time" | bc -l | xargs printf "%.3f")
30
31      echo "Frequency: $FREQ MHz, Total Time: $runtime seconds"
32      echo "-----------------------------------"
33  done
```

برای استفاده از این بش اسکریپت کافی است به این گونه آن را اجرا کنیم:

./profiling.sh [script_name] [number_of_runs]

که در آن script_names نام فایل بش موجود در وظایف بنچمارک MiBench است و number_of_runs نیز تعداد دفعات اجرا به ازای هر سطح فرکانسی است.

همچنین کد دیگری نیز زده‌ایم که بتوانیم فرکانس را به عنوان ورودی بدهیم و روی تمامی فرکانس‌ها حلقه نزنیم.

```
1   #!/bin/bash
2
3   if [ $# -ne 3 ]; then
4       echo "Usage: $0 <script_name> <number_of_runs> <frequency>"
5       exit 1
6   fi
7
8   SCRIPT_NAME=$1
9   NUM_RUNS=$2
10  FREQ=$3
11
12  echo "Setting CPU frequency to $FREQ MHz"
13  sudo cpufreq-set -c 0 -f ${FREQ}Mhz
14  sudo cpufreq-set -c 1 -f ${FREQ}Mhz
15  sudo cpufreq-set -c 2 -f ${FREQ}Mhz
16  sudo cpufreq-set -c 3 -f ${FREQ}Mhz
17
18  start_time=$(date +%s.%N)
19
20  for ((i=1; i<=NUM_RUNS; i++)); do
21      ./"$SCRIPT_NAME"
22  done
23
24  end_time=$(date +%s.%N)
25  runtime=$(echo "$end_time - $start_time" | bc -l |
26  xargs printf "%.3f")
```

```
27
28 echo "Frequency: $FREQ MHz, Total Time: $runtime seconds"
29 echo "-----------------------------------"
```

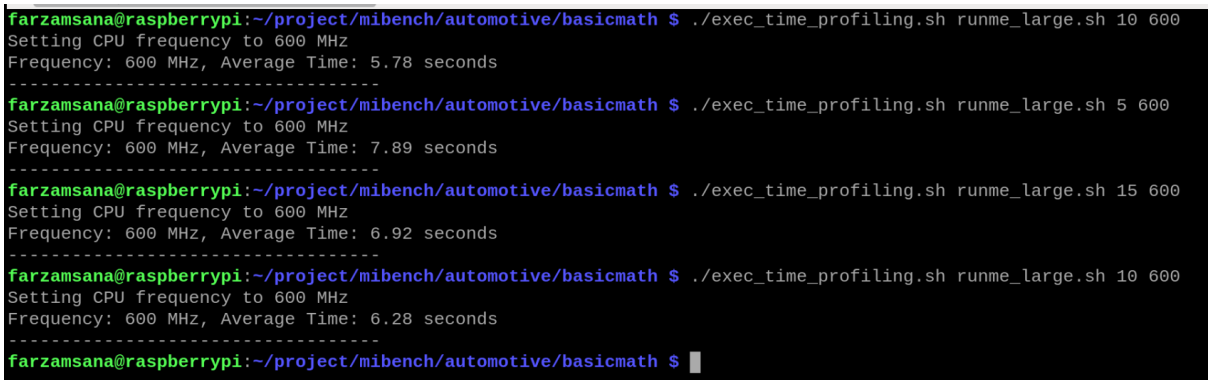
استفاده از این کد نیز به صورت زیر خواهد بود:

./profiling.sh [script_name] [number_of_runs] [frequency]

همچنین یک برنامه دیگر نیز نوشتیم که با ورودی گرفتن تعداد دفعات اجرا، مدت زمان کل را محاسبه کرده و با تقسیم بر تعداد اجرا، میانگین زمان اجرای هر برنامه را بدست می‌آورد.

```
1  #!/bin/bash
2
3  if [ $# -ne 3 ]; then
4      echo "Usage: $0 <script_name> <number_of_runs> <frequency>"
5      exit 1
6  fi
7
8  SCRIPT_NAME=$1
9  NUM_RUNS=$2
10 FREQ=$3
11
12 echo "Setting CPU frequency to $FREQ MHz"
13 sudo cpufreq-set -c 0 -f ${FREQ}Mhz
14 sudo cpufreq-set -c 1 -f ${FREQ}Mhz
15 sudo cpufreq-set -c 2 -f ${FREQ}Mhz
16 sudo cpufreq-set -c 3 -f ${FREQ}Mhz
17
18
19 start_time=$(date +%s.%N)
20
21 for ((i=1; i<=NUM_RUNS; i++)); do
22     ./"$SCRIPT_NAME"
23 done
24
25 end_time=$(date +%s.%N)
26
27 runtime=$(echo "$end_time - $start_time" | bc -l | xargs printf "%.3f")
28
29 echo "Frequency: $FREQ MHz, Average Time: $(echo "scale=2; $runtime /
     $NUM_RUNS" | bc) seconds"
30 echo "-----------------------------------"
```

برای مثال، برای basicmath در automotive اجرا کردیم و نتایج مطابق زیر بود:



شکل ۱۰: اجرای بش اسکریپت روی basicmath

دقت شود که ما در تمام برنامه‌های بنچمارک، ورژن large را اجرا کردیم.

از آنجا که مدت زمان اجرای هر وظیفه برای یک مرتبه مقداری بسیار کم است (در حدود ۱ ثانیه)، بررسی توان مصرفی در آن مدت زمان کوتاه با چشم بسیار سخت است، بنابراین تعداد دفعات اجرا را عددی مناسب قرار می‌دهیم که بررسی میزان انرژی و توان مصرفی با چشم ساده‌تر شود.

نکته بسیار مهم

پس از بررسی‌ها و تلاش‌های فراوان، دیدیم که نمونه برداری از توان مصرفی از طریق USB Power Meter بسیار دشوار است، بنابراین تصمیم گرفتیم که انرژی مصرفی را trace کنیم، که با واحد mWh مانیتور می‌شود، و با تقسیم آن بر زمان اجرای وظیفه، می توانیم توان مصرفی آن وظیفه را بدست آوریم. ما پروفایل توان مصرفی و میانگین زمان اجرا را به ازای تعدادی از وظایف موجود در بنچمارک MiBench بدست آوردیم و در این صفحه گسترده موجود است.

۵.۵ پیاده سازی الگوریتم EDF و DVFS

برای پیاده سازی این دو، ما دو کد زدیم، کد اول به نام offline_scheduler.cpp ورودی‌ای به صورت فایل به نام test_case.in ورودی می‌گرفت که در آن به ازای هر هسته، گفته می‌شود که کدام وظایف بنچمارک با چه ددلاینی قرار است اجرا شوند. برای مثال یک نمونه از فایل test_case.in می‌تواند به شکل زیر باشد.

```
0 basicmath 5
1 basicmath 10
2 jpeg 23
2 basicmath 10
3 jpeg 17
1 jpeg 12
```

در هر خط به ترتیب شماره هسته،نام وظیفه و ددلاین آن وظیفه را ورودی می دهیم.

این کد، قصد دارد ورودی را به شکل گفته شده دریافت کرده و سپس خروجی را به صورت EDF و تعیین فرکانس به ازای هر کدام از وظایف نوشته و در فایل خروجی output.txt قرار دهد. بدین منظور، از پروفایلی که قبلا تهیه کردیم به شکل فایل csv استفاده کرده و با تعیین فرکانس‌های مناسب، **کمینه توان مصرفی ممکن را به گونه‌ای که**

هیچ موعد زمانی‌ای miss نشود. برای مثال فایل خروجی متناسب با ورودی گفته شده می‌تواند به شکل زیر باشد:

```
0 basicmath 900 5
1 basicmath 600 10
1 jpeg 600 12
2 basicmath 600 10
2 jpeg 600 23
3 jpeg 600 17
```

همانطور که مشاهده می‌شود، خروجی به ازای هر هسته، وظایف را به صورت EDF مرتب کرده و مقدار فرکانس کاری مناسب برای هر وظیفه نیز مشخص شده است.

در ادامه کد نوشته شده را توضیح می‌دهیم:

۱.۵.۵ توضیح کد offline_scheduler.cpp

در ابتدا موجودیت‌های زیر را تعریف کردیم:

```cpp
struct Task_Type {
    string task_name;
    map <int, pair<float, float>> frequency_time_power; // key is frequency , first is time, second is power
};

struct Task {
    int core;
    struct Task_Type* task_type;
    float deadline;
    float end_time;
};

struct Core {
    int id;
    vector <struct Task*> tasks;
    vector<int> DVFS_frequencies;
    float total_power_consumption;
};
```

استراکت Core در واقع نمایانگر هسته‌های رزبری‌پای است که دارای id یا شماره هسته، مجموعه تسک‌های اختصاص داده شده به هسته، فرکانس‌های تعیین شده بعد از تکنیک DVFS به ازای هر وظیفه تخصیص داده شده به هسته و همچنین مجموع توان مصرفی هسته با استفاده از پروفایل و فرکانس‌های تخصیص داده شده می‌باشد.

استراکت Task_Type در اصل نمایانگر هر یک از وظایف موجود در بنچمارک است، یعنی به ازای هر وظیف بنچمارک، یک Task_Type داریم که یک نام مخصوص خود دارد و همچنین یک map دارد که کلید آن، فرکانس کاری هسته (مانند ۶۰۰ و ۷۰۰ و ۸۰۰و ۹۰۰ و ۱۰۰۰ و ۱۱۰۰ و ۱۲۰۰) است و مقدار آن زوج مرتب‌های زمان اجرا و توان مصرفی وظیفه است.

استراکت Task نیز شامل شماره هسته مربوط به آن نمونه وظیفه، نوع تسک

حال از توضیحات بیشتر جزئیات پرهیز می‌کنیم و صرفا وDVFS EDF را که روی هر هسته اجرا می‌شوند، توضیح می‌دهیم:

```
bool compareTasks(struct Task* a, struct Task* b) { // EDF
    return a->deadline < b->deadline;
}

void EDF_scheduling(struct Core** cores) {
    for (int i = 0; i != 4; i++) {
        struct Core* core = cores[i];
        sort(core->tasks.begin(), core->tasks.end(), compareTasks);
    }
}
```

الگوریتم EDF به شکل بالا است، که صرفا تسک‌های هر هسته را به ترتیب موعد زمانی آنها و به شکل صعودی مرتب می‌کند.

```
bool is_deadline_missed(struct Core* core, vector<int> tasks_frequencies) {
    if (core->tasks.size() == 0)
        return false;
    struct Task* first_task = core->tasks[0];
    first_task->end_time = first_task->task_type->frequency_time_power[
       tasks_frequencies[0]].first;

    if (first_task->end_time > first_task->deadline) {
        return true;
    }

    for (int i = 1; i != tasks_frequencies.size(); i++) {
        struct Task* task = core->tasks[i];
        struct Task* last_task = core->tasks[i-1];
        task->end_time = last_task->end_time + task->task_type->
           frequency_time_power[tasks_frequencies[i]].first;
        if (task->end_time + 0.5 > task->deadline) {
            return true;
        }
    }
    return false;
}

void greedy_frequency_determiner(Core* core, vector<int> tasks_frequencies,
    float tasks_power_consumption) {
    if (tasks_frequencies.size() == core->tasks.size()) {
        if (is_deadline_missed(core, tasks_frequencies))
            return;
        if (tasks_power_consumption < core->total_power_consumption) {
```

```
27              core->total_power_consumption = tasks_power_consumption;
28              core->DVFS_frequencies = tasks_frequencies;
29          }
30          return;
31      }
32      int task_number = tasks_frequencies.size();
33      for (int i = 0; i != 7; i++) {
34          float original_power = tasks_power_consumption;
35
36          tasks_frequencies.push_back(FREQUENCIES[i]);
37
38          tasks_power_consumption += core->tasks[task_number]->task_type->
                frequency_time_power[FREQUENCIES[i]].second;
39
40
41          if (!is_deadline_missed(core, tasks_frequencies) && core->
                total_power_consumption >= tasks_power_consumption) {
42              greedy_frequency_determiner(core, tasks_frequencies,
                    tasks_power_consumption);
43          }
44          tasks_frequencies.pop_back();
45
46          tasks_power_consumption = original_power;
47      }
48  }
49
50  bool DVFS(struct Core** cores) {
51      for (int i = 0; i != 4; i++) {
52          struct Core* core = cores[i];
53          core->total_power_consumption = INT64_MAX;
54          vector<int> tasks_frequencies;
55          greedy_frequency_determiner(core, tasks_frequencies, 0);
56          if (core->DVFS_frequencies.size() != core->tasks.size())
57              return false;
58      }
59      return true;
60  }
```

کد بالا نیز، پیاده سازی DVFS است، به این شکل که یک تابع بازگشتی زدیم که به ازای همه‌ی فرکانس‌های ممکن (۷ سطح فرکانسی) برای هر وظیفه تخصیص داده شده به هسته‌ها، میزان توان مصرفی کل را محاسبه کرده و در صورتی که هیچ ددلاینی میس نشده بود (برای اطمینان، تعریف میس نشدن ددلاین را به اینگونه گفتیم که فاصله زمان پایان وظیفه، با موعد زمانی آن، از یک threshold با مقدار ۵.۰ بیشتر باشد)، با کمینه توان مصرفی‌ای که تا اکنون بدست آورده مقایسه می‌کند، در صورتی که این میزان کمتر بود، آن را جایگزین می‌کند و در غیر اینصورت به سراغ مورد بعدی می‌رود. دقت شود که رویکرد ما برای پیدا کردن فرکانس‌های بهینه وظایف، یک رویکرد greedy است و ما تمام حالات ممکن را بررسی می‌کنیم.

پس از اینکه خروجی این برنامه آماده شد، که شامل ترتیب مناسب وظایف روی هر هسته و فرکانس کاری مناسب هر وظیفه است، آن را به کد دیگری به نام simulator.cpp می‌دهیم که در ادامه آن را توضیح خواهیم داد.

۲.۵.۵ توضیح کد simulator.cpp

این کد قرار است با گرفتن ترتیب مناسب وظایف هر هسته و فرکانس کاری مناسب هر وظیفه، ۴ کد بش اسکریپت به ازای هر هسته بنویسد که به صورت core_i.sh نام گذاری شده‌اند که i شماره هسته را نشان می‌دهد. همچنین یک کد بش اسکریت دیگر به نام simuator.sh نیز تولید می‌کند که قصد دارد چهار بش اکسریپت مرتبط با هسته‌ها را به طور همزمان صدا بزند تا تمامی هسته‌ها کارشان را با هم شروع کنند.

خود این کد توضیح خیلی خاصی ندارد و صرفا یک نمونه از خروجی آن را که مرتبط با ورودی‌های قبلی است نشان می‌دهیم. کد زیر، core_1.sh است که درواقع وظایف مربوط به هسته ۱ را به همراه تعیین فرکانس کاری مرتبط به وظایف را انجام می‌دهد و بعد از اجرا، در صورتی که ددلاین میس شده بود، آن را اعلام می‌کند.

```
#!/bin/bash

LOG_FILE=core_1_log.txt
echo 'Core 1 Execution Log' > $LOG_FILE
FIRST_TIME=$(date +%s.%N)
echo 'Setting Core 1 to 600 MHz'
sudo cpufreq-set -c 1 -f 600Mhz
echo 'Running basicmath on Core 1'
taskset -c 1 ./benchmarks/basicmath/runme_large.sh
END_TIME=$(date +%s.%N)
TURN_TIME=$(echo "$END_TIME - $FIRST_TIME" | bc)
if (( $(echo "$TURN_TIME > 10" | bc -l) )); then
    echo "Task basicmath on Core 1 MISSED DEADLINE! Took $TURN_TIME sec,
        deadline was 10 sec" >> $LOG_FILE
else
    echo "Task basicmath on Core 1 met deadline. Took $TURN_TIME sec,
        deadline was 10 sec" >> $LOG_FILE
fi
echo 'Setting Core 1 to 600 MHz'
sudo cpufreq-set -c 1 -f 600Mhz
echo 'Running jpeg on Core 1'
taskset -c 1 ./benchmarks/jpeg/runme_large.sh
END_TIME=$(date +%s.%N)
TURN_TIME=$(echo "$END_TIME - $FIRST_TIME" | bc)
if (( $(echo "$TURN_TIME > 12" | bc -l) )); then
    echo "Task jpeg on Core 1 MISSED DEADLINE! Took $TURN_TIME sec,
        deadline was 12 sec" >> $LOG_FILE
else
    echo "Task jpeg on Core 1 met deadline. Took $TURN_TIME sec, deadline
        was 12 sec" >> $LOG_FILE
fi
```

همچنین کد زیر، در اصل کد master یا همان simulator.sh است که تمامی بش‌های مربوط به هسته‌ها را به طور همزمان صدا می‌زند و wait می‌کند تا اجرای آنها تمام شود.

```
#!/bin/bash

echo 'Starting all cores in parallel...'
./core_0.sh &
./core_1.sh &
./core_2.sh &
./core_3.sh &
wait
echo 'All core scripts have started their tasks.'
```

حال برنامه‌های نوشته شده را به ازای تعدادی ورودی متفاوت، هم با DVFS و هم بدون آن بررسی کرده و نتایج را مقایسه می‌کنیم.

۶.۵ محدودیت‌ها و ایده‌های بعدی

با توجه به حافظه محدود ۸ گیگا بایتی بوردمان، تصمیم بر این گرفتیم که بخش offline را در سیستم local خودمان انجام دهیم و همچنین قسمت simulation را و سپس ۵ فایل بش خروجی حاصل را به سیستم رزبری‌پای منتقل کنیم.

یک سری ایده‌هایی مطرح شد که می توان برای بهبود پروژه آن را به کار برد ولی به دلیل مدت زمان محدود در زیر بیانشان می کنیم:

- می توان برای اینکه اطمینان از میت شدن موعدهای زمانی تسک‌ها و همچنین کاهش اردر محاسباتی می توانیم در اجرای DVFS بدین صورت عمل کنیم که برای هر تسک زمان باقی مانده را به نسبت Slack Timeهای توزیع کنیم و اگر تسکی ممکن بود که ددلاینش میس شود، مینیمم ددلاین آن تسک و زمان حاصل از نسبت Slack Time ها را به آن اختصاص می دهیم.
- برای کاهش اردر محاسباتی در قسمت offline می توانیم از thread استفاده کنیم.

۶ آزمایش و تحلیل نتایج

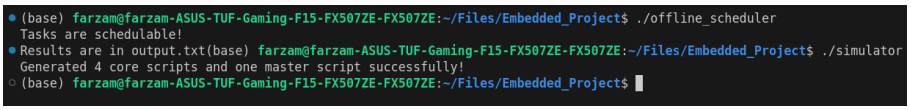
برای انجام آزمایشات، در ابتدا نیاز است که سناریوهایی تعریف کرده و آنها را بهمراه فایل csv پروفایل توان مصرفی، در کنار offline_scheduler.cpp قرار دهیم. سپس با اجرا کردن offline_scheduler.cpp و simulator.cpp، فایل‌های بش مربوط به هر هسته و فایل بش شبیه سازی را بدست آورده و روی Raspberry Pi در کنار بنچمارک‌ها اجرا می‌کنیم.

برای مثال سناریوی زیر را در نظر بگیرید:

هسته	نام کار	ددلاین
0	basicmath	7
0	dijkstra	8
0	basicmath	15
1	qsort	0.8
1	qsort	1.5
1	dijkstra	2
1	basicmath	9
2	dijkstra	0.5
2	qsort	1.5
2	basicmath	8.5
2	qsort	10.5
3	bitcount	0.6
3	bitcount	1.5
3	qsort	2.5
3	dijkstra	3
3	qsort	3.5

جدول ۱: زمان‌بندی وظایف روی هسته‌های مختلف

این سناریو را به offline_scheduler داده و سپس simulator.cpp را اجرا می‌کنیم تا فایل‌های بش را برای ما ایجاد کند.



شکل ۱۱: خروجی موفقیت آمیز scheduling و ساخت فایل‌های بش

حال فایل‌های بش را روی رزبری پای ریخته و بش اصلی، یعنی simulator.sh را اجرا می‌کنیم. سپس فایل‌های log را بررسی کرده تا متوجه شویم چه ددلاین‌هایی میس شده است. از آنجا که ما با AET کار کرده ایم، ممکن است تعدادی از ددلاین‌ها میس شده باشند اما به طول کلی، عملکرد قابل قبول است.

```
farzamsana@raspberrypi:~/project $ ./simulator.sh
Starting all cores in parallel...
Setting Core 1 to 1200 MHz
Setting Core 0 to 700 MHz
Setting Core 2 to 600 MHz
Setting Core 3 to 600 MHz
Running basicmath on Core 0
Running qsort on Core 1
Running dijkstra on Core 2
Running bitcount on Core 3
Setting Core 2 to 900 MHz
Running qsort on Core 2
Setting Core 1 to 1200 MHz
Running qsort on Core 1
Setting Core 3 to 800 MHz
Running bitcount on Core 3
Setting Core 2 to 600 MHz
Setting Core 1 to 600 MHz
Running basicmath on Core 2
Running dijkstra on Core 1
Setting Core 3 to 600 MHz
Running qsort on Core 3
Setting Core 1 to 600 MHz
Running basicmath on Core 1
Setting Core 3 to 600 MHz
Running dijkstra on Core 3
Setting Core 3 to 600 MHz
Running qsort on Core 3
Setting Core 0 to 600 MHz
Running dijkstra on Core 0
Setting Core 0 to 600 MHz
Running basicmath on Core 0
Setting Core 2 to 600 MHz
```

شکل ۱۲: اجرای شبیه‌سازی روی رزبری پای

```
Setting Core 0 to 600 MHz
Running dijkstra on Core 0
Setting Core 0 to 600 MHz
Running basicmath on Core 0
Setting Core 2 to 600 MHz
Running qsort on Core 2
All core scripts have started their tasks.
farzamsana@raspberrypi:~/project $ cat core_0_log.txt
Core 0 Execution Log
Task basicmath on Core 0 met deadline. Took 3.942038887 sec, deadline was 7 sec
Task dijkstra on Core 0 met deadline. Took 4.358835462 sec, deadline was 8 sec
Task basicmath on Core 0 met deadline. Took 10.458749345 sec, deadline was 15 sec
farzamsana@raspberrypi:~/project $ cat core_1_log.txt
Core 1 Execution Log
Task qsort on Core 1 met deadline. Took .729331989 sec, deadline was 0.8 sec
Task qsort on Core 1 MISSED DEADLINE! Took 1.644615625 sec, deadline was 1.5 sec
Task dijkstra on Core 1 MISSED DEADLINE! Took 2.072064571 sec, deadline was 2 sec
Task basicmath on Core 1 met deadline. Took 8.538262387 sec, deadline was 9 sec
farzamsana@raspberrypi:~/project $ cat core_2_log.txt
Core 2 Execution Log
Task dijkstra on Core 2 MISSED DEADLINE! Took .591585385 sec, deadline was 0.5 sec
Task qsort on Core 2 MISSED DEADLINE! Took 1.618719644 sec, deadline was 1.5 sec
Task basicmath on Core 2 met deadline. Took 8.196827093 sec, deadline was 8.5 sec
Task qsort on Core 2 met deadline. Took 8.825385724 sec, deadline was 10.5 sec
farzamsana@raspberrypi:~/project $ cat core_3_log.txt
Core 3 Execution Log
Task bitcount on Core 3 MISSED DEADLINE! Took .744212735 sec, deadline was 0.6 sec
Task bitcount on Core 3 MISSED DEADLINE! Took 1.661633293 sec, deadline was 1.5 sec
Task qsort on Core 3 met deadline. Took 2.435788979 sec, deadline was 2.5 sec
Task dijkstra on Core 3 met deadline. Took 2.980592962 sec, deadline was 3 sec
Task qsort on Core 3 MISSED DEADLINE! Took 3.804957957 sec, deadline was 3.5 sec
farzamsana@raspberrypi:~/project $
```

شکل ۱۳: نتیجه شبیه‌سازی

۷ نتیجه‌گیری

با توجه به اینکه یکی از موارد بسیار مهم در سیستم‌های نهفته، مدیریت توان مصرفی است، استفاده از تکنیک DVFS به همراه پروفایلینگ وظایف اختصاص داده شده به سیستم، می‌تواند عملکرد بسزایی در مدیریت توان مصرفی داشته باشد. ما در این پروژه با پروفایل توان مصرفی تعدادی از وظایف بنچمارک MiBench و همچنین محاسبه میانگین زمان اجرای آنها، توانستیم با روشی حریصانه یک برنامه ریزی بهمراه DVFS از آن ارائه دهیم که در عین حال که توان مصرفی را با تغییر پویای فرکانس کمینه می‌کند، تلاش دارد تا موعد زمانی وظایف را نیز رعایت کند. عملکرد این برنامه ریز نسبتا خوب و قابل قبول است و ددلاین‌ها را تا حد خوبی رعایت می‌کند.