Dynamic Frequency System(DFS): a practical nonvolatile processor

Abstract

In this project, we implemented a simulation model, a simple nonvolatile processor with two-threshold energy management unit, based on the current gem5-NVP toolkit. To achieve this goal, we proposed a well-designed system architecture and simple message communication pipes among atomic simple CPU, energy management and CPU clock domain. Detailed and massive simulation experiments based on this implementation has demonstrated the correctness of our designed system. Our study upon simulation has revealed that the DFS design is able to utilize its privileges when energy harvested by the processor is nearly enough to sustain its normal running on low frequency while other design running on high frequency will fail by close down the machine.

1 Introduction

With the rapid development of science and technology, the living conditions of people becomes better and more comfortable. The internet of things is one of the most heated area for improving the quality of human beings' lives. Small and accurate sensors are applied everywhere to capture more delicate first-hand data for making better decision when providing services, such as temperate sensors and oxygen sensors. However, most of these sensors are powered by local batteries and connected to the cloud services. The quality and life of batteries exert an enormous influence on the performance of these lovely small sensors. For example, these sensors may suffer from frequent power cutoff and energy surge. Therefore, it is necessary to design a proper processor architecture to deal with these problems. The idea of nonvolatile processors is put forward.

The essential of nonvolatile processor [Xie et al., 2015] is about backup and restore while power is cutting off. We need to back up the architecture state, micro-architecture state, and performance enhancers. All the state elements should either be designed as nonvolatile memory or backed up to nonvolatile memory. The architecture state is the minimum state required to be backed up for an NVP(nonvolatile processor) design.

In this project we use a modified gem5 [Binkert et al., 2011] branch `https://github.com/zlfben/gem5` to conduct simulations on energy harvest and consumption. This branch has equipped the original version of gem5 with very basic energy management module, which offers chances to produce a more profound view and simulation of NVP.

Here we summarize our **contribution** in this project

- We've explored the Event, Event Queue Management and Message Broadcast mechanism in gem5.
- We carefully analyzed the pros and cons of preexisted systems like AtomicSimpleCPU and two-threshold state machine with detailed experiments.

- We developed a DFS system which has been proved to work well on short session power-off through low frequency running which could help to enhance the performance of processor.

Our **innovation** parts can be listed as follows

- Our implementation is non-trivial and thus is extremely portable. We fully utilizes the message broadcast mechanism and clock domain class so other implementations will not be influenced. Our implementation can be modified easily to accommodate similar settings, e.g., introducing new states like transition state into simulation.
- We've created auto-script whose testing numbers is up to 150 to fully analyze the performance of simulating designs. We provide script to automatically give statistical information based on our automatically generated energy profile settings.

2 System Design and Implementation

Below is the overview of our designed system for two-threshold/frequency NVP. Basically we maintain 3 classes here, ***Energy Management***, ***ClockDomain*** and ***AtomicSimpleCPU***. During the communication among these units, ports of *Energy Management* serves as the master, whereas ports of *AtomicSimpleCPU* and *ClockDomain* serve as slave.

Energy Management module is able to broadcast message to slave ports, and slave ports will handle messages sent and take responding actions. Each tick cycle, Energy Management module will first compute the net value of energy harvested and consumed. After that, it will examine its energy threshold, that is to say, if the current energy is above the higher threshold, message standing for high frequency will be broadcast, and if the current energy is below the lower threshold, message for powering off will be broadcast.

AtomicSimpleCPU will reflect on the message by schedule or deschedule into eventqueue. A power-off message means that this processor needs to be shut down, so *deschedule()* will becalled.

ClockDomain will adjust its CPU clock frequency, i.e. the performance level of CPU, based on message it received.

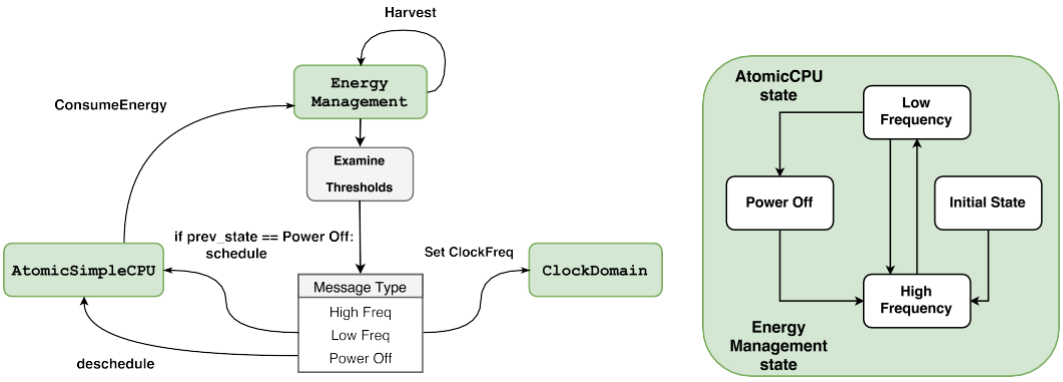

Figure 1: System Design

To implement our system, we modified several classes in gem5-NVP. Detailed information can be found through our source codes.

3 Experiment

This section will provide respondents to the list of items required by project assignments.

3.1 Event Mechanism of GEM5

TickEvent will be triggered during the running of AtomicSimpleCPU. It is pushed into the Event Queue through function *schedule()*. When the *TickEvent* is in the front of **EventQueue**, *tickEvent:process()*, i.e. *AtomicSimpleCPU::tick()* will be called during routinely calling of *EventQueue::serviceOne()*. At the end of *tickEvent:process()*, another *schedule()* function will be called so that this mechanism could run in circle, just acting like a real running CPU. *activateContext()* and *suspendContext()* will control the start and end of scheduling process of each thread. It is worth noticing that **EventQueue** is not necessarily FIFO, this could be modified by *deschedule()* and *reschedule()* function provided by *EventManager*.

3.2 Memory Access Time Simulation

Time consumption varies between instructions which access memory and instructions that don't access memory. To be specific, the requirements of accessing memory will usually results in stalls for corresponding instructions, so time consumption of these instructions should be larger. In *AtomicSimpleCPU*, this feature is achieved inside logic design in *AtomicSimpleCPU::tick()*. Gem5 will calculate latency using preset parameters. Latency will affect the schedule-deschedule pipeline of *AtomicSimpleCPU*, thus results in the difference between two types of instructions. Codes related to the calculation of latency can be viewed as follows.

```
Tick stall_ticks = 0;
if (simulate_inst_stalls && icache_access)
        stall_ticks += icache_latency;

if (simulate_data_stalls && dcache_access)
        stall_ticks += dcache_latency;

if (vdev_set) {
        vdev_set = 0;
        stall_ticks += vdev_set_latency;
}

if (stall_ticks) {
        // the atomic cpu does its accounting in ticks, so
        // keep counting in ticks but round to the clock
        // period
        latency += divCeil(stall_ticks, clockPeriod()) * clockPeriod();
}
```

Note that we also have to take care of the calculation of cache latency here, for ichache latency, piece of codes goes as follows.

```
Tick icache_latency = 0;

...

if (fastmem && system->isMemAddr(ifetch_pkt.getAddr()))
        system->getPhysMem().access(&ifetch_pkt);
else
        icache_latency = icachePort.sendAtomic(&ifetch_pkt);
```

There's also a piece of code related to dcache, which is

```
Tick dcache_latency = 0;

...

if (req->isMmappedIpr())
        dcache_latency += TheISA::handleIprRead(thread->getTC(), &pkt);
        else {
        if (fastmem && system->isMemAddr(pkt.getAddr()))
        system->getPhysMem().access(&pkt);
        else
        dcache_latency += dcachePort.sendAtomic(&pkt);
}
```

The reason that memory accessing time is calculated but not shown in the simulating results is because python script is not properly set. As can be viewed from above, *stall_ticks* will be added up only when *simulate_data_stalls* and simulate_inst_stalls is set *True*, and the default setting in python script is *False*. We can modify the piece of codes as follows and the access time is reflected in outputs since then.

```
parser.add_option("--icache-stall", action="store_true", default=False,
                  help="imulate.icache.stall.cycles.in.Atomic.CPU")
parser.add_option("--dcache-stall", action="store_true", default=False,
                  help="imulate.dcache.stall.cycles.in.Atomic.CPU")
# ......
for i in xrange ( np ):
    if options . cpu_type == "atomic":
            if options . icache_stall :
                system . cpu [ i ]. simulate_inst_stalls = True
            if options . dcache_stall :
                system.cpu[i].simulate_data_stalls = True
```

Here we give sample of outputs on our experiments conducted on 8 queens algorithm compiled on ARM. The second one is the modified version. Some tick lapses beteen two instructions vary from originally exactly 500 ticks to more than that. We can see from the results that the later one has considered time costs on data cache, which proves the correctness of our modification.

```
161558000: system.cpu: Tick
161558500: system.cpu: Tick
161559000: system.cpu: Tick
161559000: system.cpu.dcache_port: received snoop pkt for addr:0x17798 ReadReq 1
61559500: system.cpu: Tick
161559500: system.cpu.dcache_port: received snoop pkt for addr:0x1779c ReadReq 1
61560000: system.cpu: Tick
161560000: system.cpu.dcache_port: received snoop pkt for addr:0x177a0 ReadReq 1
61560500: system.cpu: Tick
161560500: system.cpu.dcache_port: received snoop pkt for addr:0x177a4 ReadReq 1
61561000: system.cpu: Tick
161561000: system.cpu.dcache_port: received snoop pkt for addr:0x177b8 ReadReq 1
61561500: system.cpu: Tick
161561500: system.cpu.dcache_port: received snoop pkt for addr:0x177bc ReadReq 1
61562000: system.cpu: Tick
161562000: system.cpu.dcache_port: received snoop pkt for addr:0x177c0 ReadReq 1
61562000: system.cpu: SuspendContext 0
Exiting @ tick 161562000 because target called exit ()
```

```
16177968500: system.cpu: Tick
16177969000: system.cpu: Tick
16177969500: system.cpu: Tick
16177969500: system.cpu.dcache_port: received snoop pkt for addr:0x17798 ReadReq 1
6178011000: system.cpu: Tick
16178011000: system.cpu.dcache_port: received snoop pkt for addr:0x1779c ReadReq 1
6178093500: system.cpu: Tick
16178093500: system.cpu.dcache_port: received snoop pkt for addr:0x177a0 ReadReq 1
6178176000: system.cpu: Tick
16178176000: system.cpu.dcache_port: received snoop pkt for addr:0x177a4 ReadReq 1
6178217500: system.cpu: Tick
16178217500: system.cpu.dcache_port: received snoop pkt for addr:0x177b8 ReadReq 1
6178259000: system.cpu: Tick
16178259000: system.cpu.dcache_port: received snoop pkt for addr:0x177bc ReadReq 1
6178300500: system.cpu: Tick
16178300500: system.cpu.dcache_port: received snoop pkt for addr:0x177c0 ReadReq 1
6178300500: system.cpu: SuspendContext 0
Exiting @ tick 16178300500 because target called exit ()
```

3.3 Simple Energy Management

Any state machine of energy management will inherit from *BaseEnergySM* which inherited from *EnergyMgmt*. A new message ready to send is settled by calling *BaseEnergySM::broadcastMsg(const EnergyMsg &msg)*, this function will further call *EnergyMgmt::broadcastMsgAsEvent(const EnergyMsg &msg)* to schedule to *msg* to an *EventWrapper* instance *event_msg*. If *msg_togo* is empty, then the *msg* will be scheduled directly, however, if *msg_togo* is not empty, this actually means that there are some messages scheduling currently at *EnergyMgmt::BroadcastMsg()*, so we can just simply wait for others to handle the message first. Every time we update the status of state machine, *msg* will be broadcast, this observation could be drawn from the piece of codes of function *SimpleEnergySM::update(double _energy)* following

```
if (state == STATE_INIT) {
        state = STATE_POWERON;
} else if ( state == STATE_POWERON && _energy < 0)
        { state = STATE_POWEROFF;
        msg.type = MsgType ::POWEROFF;
        broadcastMsg(msg);
} else if ( state == STATE_POWEROFF && _energy > 0)
        { state = STATE_POWERON;
        msg.type = MsgType::POWERON;
        broadcastMsg(msg);
}
```

EventWrapper instance *event_msg* will be responsible for calling *EnergyMgmt::broadcastMsg()*, and this function will further call *_meport* to broadcast the message so that CPU could receive it.

When message is received by *AtomicSimpleCPU*, *AtomicSimpleCPU::handleMsg(const EnergyMsg &msg)* will be called to handle the message, say, broadcasted by EnergyMgmt. This could be verified by example output like follows

```
0: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 6.57149 1000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 5.57149 2000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 4.57149 3000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 3.57149 4000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 2.57149 5000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 1.57149 6000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 0.571488 7000:
system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: -0.428512 7000:
system.cpu: AtomicSimpleCPU handleMsg called at 7000, msg.type=1
1000000: system.energy_mgmt: Energy 7.57149 is harvested. Energy remained: 7.14298
1000000: system.cpu: AtomicSimpleCPU handleMsg called at 1000000, msg.type=2      1
000000: system.energy_mgmt: Energy -0 is harvested. Energy remained: 7.14298
```

3.4 Two Threshold Energy Management

Below are two piece of outputs that show state changing during the running of NVP. We see that when energy remained has exceeded the upper threshold, state machine will change from off to on. On the other hand, when Energy remained has been consumed to be less than 10000(lower threshold), state machine will then turn off.

```
2760000000: system.energy_mgmt: Energy 9.53194 is harvested. Energy remained: 19984.9
2761000000: system.energy_mgmt: Energy 9.53194 is harvested. Energy remained: 19994.4
2762000000: system.energy_mgmt: Energy 9.53194 is harvested. Energy remained: 20004  2
762000000: system.energy_mgmt.state_machine: State change: off->on
            state =1 , _ energy =20004 , th res_ h ig h = 20000
2762000000: system.cpu: AtomicSimpleCPU handleMsg called at 2762000000, msg.type=2 2
762000000: system.energy_mgmt: Energy -0 is harvested. Energy remained: 20004     27
62000000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 20003 276
2001000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 20002
```

```
26607095000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 10002  2
6607096000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 10001  26
607097000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 9999.96 266
07097000: system.energy_mgmt.state_machine: State change: on->off
            state =2 , _ energy = 9999.96 , th res_ lo w = 10000
26607097000: system.cpu: AtomicSimpleCPU handleMsg called at 26607097000, msg.type=1  2
6608000000: system.energy_mgmt: Energy 9.35796 is harvested. Energy remained: 10009.3
```

The above outputs could help to demonstrate the correctness of our implementation.

```
system.energy_mgmt = EnergyMgmt(path_energy_profile = options.energy_profile,
                                energy_time_unit = options.energy_time_unit,
                                state_ m ach in e = T w o T h resS M ( th res_ h ig h = options . thres_high ,
                                                                      th res_ lo w = options . th res_ lo w ))
```

3.5 Two-Threshold/Frequency Energy Management (DFS system)

Here we also give several trace examples showing the transfer in status, note that this time we have 3 internal states, namely high, low and off. The transfer among those states are controlled by two thresholds, upper threshold and lower threshold. The most noticeable difference between this state machine and the above two-thres machine is that, when the level of energy remained run down through the upper threshold, this, machine will change from high frequency to low frequency, avoiding energy being consumed too quickly to stop the machine. By having more time to run at low freq state, this machine maintains a higher probability to keep its running status, which conforms to the aim of NVP.

Example of state machine that moves from off state to high freq state:

```
5183000000: system.energy_mgmt: Energy 11.5393 is harvested. Energy remained: 19997.95
184000000: system.energy_mgmt: Energy 11.5659 is harvested. Energy remained: 20009.451
84000000: system.energy_mgmt.state_machine: State change: off->high
            state=1, _energy=20009.4, thres_high=20000
5184000000: system.cpu_clk_domain: SrcClockDomain handleMsg called at 5184000000, msg.type=2
5184000000: system.cpu: AtomicSimpleCPU handleMsg called at 5184000000, msg.type=2          5
184000000: system.energy_mgmt: Energy -0 is harvested. Energy remained: 20009.4
```

```
5184000000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 20008.4
5184001000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 20007.4
```

Example of state machine that moves from high freq state to low freq state:

```
5184007000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 20001.45
184008000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 20000.451
84009000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19999.4518
4009000: system.energy_mgmt.state_machine: State change: high->low
            state=2, _energy=19999.4, thres_high=20000
5184009000: system.cpu_clk_domain: SrcClockDomain handleMsg called at 5184009000, msg.type=3
5184009000: system.cpu: AtomicSimpleCPU handleMsg called at 5184009000, msg.type=3          5
184010000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19998.4       51
84012000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19997.4       518
4014000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19996.4       5184
016000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19995.4       51840
18000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19994.4       518402
0000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19993.4       5184022
000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 19992.4
```

Example of state machine that moves from low freq state to off state:

```
13456305000: system.energy_mgmt: Energy 1 is consumed by xxx. Energy remained: 9999.981
3456305000: system.energy_mgmt.state_machine: State change: low->off
             state =3 , _energy = 9999.98 , th res_lo w = 10000
13456305000: system.cpu_clk_domain: SrcClockDomain handleMsg called at 13456305000, msg.type=1
13456305000: system.cpu: AtomicSimpleCPU handleMsg called at 13456305000, msg.type=1          1
3457000000: system.energy_mgmt: Energy 7.57149 is harvested. Energy remained: 10007.5        13
458000000: system.energy_mgmt: Energy 7.57149 is harvested. Energy remained: 10015.1        134
59000000: system.energy_mgmt: Energy 7.57149 is harvested. Energy remained: 10022.7
```

The above outputs could help to demonstrate the correctness of our implementation.

3.6 How DFS system performs under different energy circumstances

Here we've prepared **150** different energy profile settings and selected several typical results which could in all help to demonstrate the overall performance of DFS system compared with other settings, e.g., two-threshold state machine.

Below (Figure 2) is the figure to analyze our implementation of DFS system. The envirnoment energy profile are a series of square wave forms of different period and duty ratio. We could see that under any period settings, three curves maintains a similar performance.

More specifically, we argue that the red curve refers to the condition that energy supply is sufficient in systems, while the blue curve is set to denote the kind of condition that energy supply is scanty. We can see that when energy supply is sufficient, there's no much difference between the two systems, since both processors are able to run at high frequency mode for most of the time. And as for the case that energy supply is always scanty, it is obvious that two-thres state machine energy management always performs better than DFS system. This is due to the reason that after both systems have to wait for a long time for energy to collect to beyond the same power on threshold, DFS system could only maintain high frequency status for a short session and then the energy level will decrease to be lower than upper threshold and thus CPU starts to run in low frequency mode, while two-thres system will still run in high frequency mode. That is to say, when lacking of energy, DFS system tends to run in low frequency mode but two-thres runs in high frequency.

Besides, the green curve is the most interesting curve here, since it demonstrates the situation that energy supply is just around the threshold of being enough. We find that DFS system has privileges in two conditions. Roughly, DFS performs better when environment square wave energy feeding is short-period (period of 10 or 20) and its duty ratio is really small. It also shows slightly advantages when environment square wave energy feeding is long-period (period of 50 or 20) and its duty ratio is almost 1. These two conditions corresponds to pulse energy feeding and pulse power cutoff, which will be discussed in detail later.

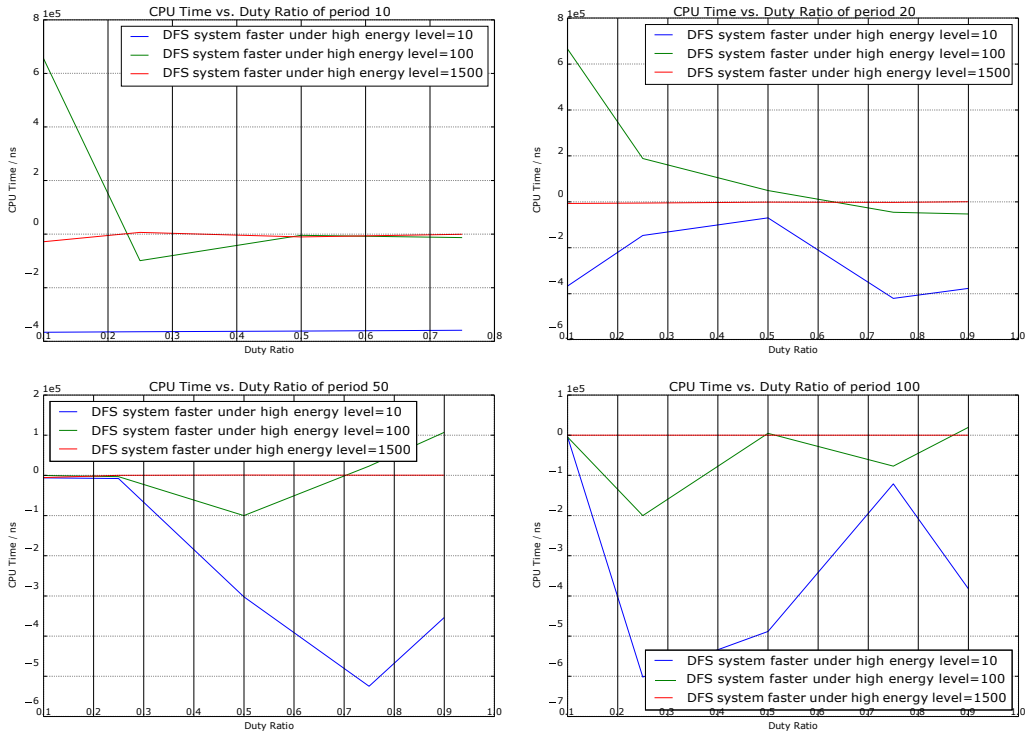

Figure 2: CPU time vs Duty ratios with different time period

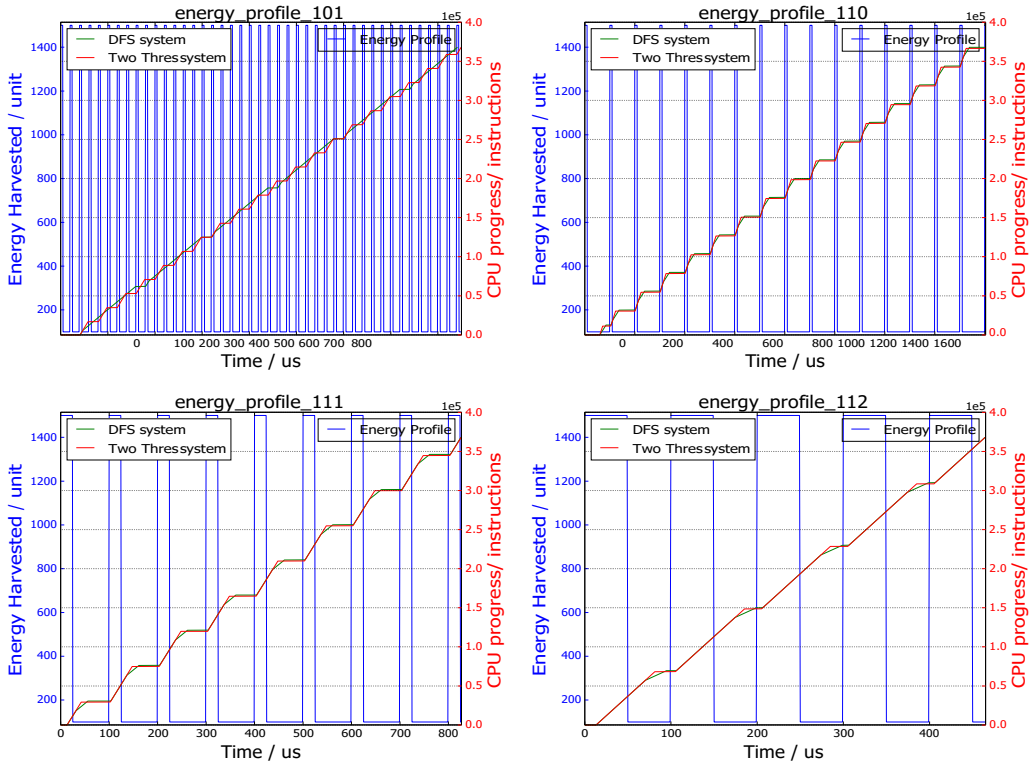

Figure 3: CPU performance under pulse energy feeding. The blue curve is energy profile and the red/green curve is the progress of DFS/two-thres CPU when runing the queens program.

The graph (Figure 3 tests on setting of pulse energy feeding, in which the processor will receive an energy impulse after a relatively long interval periodically. From the graph we could observe that, if the amount of energy given at the pulse is big enough to sustain the processor running in low frequency during the long interval while it is not the case for high frequency, DFS system will show its privilege over two-thres system.

From the four settings above, we could see that the green line is able to run for a longer time than the red line each time after the loss of energy supply. From energy profile 101 we could clearly inspect that the green line is able to stop earlier than the red line, which clearly indicates that DFS system has benefited in running this algorithm.

Besides, we figured out that a more profound insight of this problem is that the actually aspect that influences the performance of DFS system is the combination, or the proportion of impact made by duty ratio, harvested energy per sec together. We could inspect from the forth figure that the DFS system could still outperform two-thres SM, as long as the energy harvested per cycle is near the proper amount that could sustain the DFS system to run a longer time during low frequency period, which is particular noticeable in the forth figure.

The figure 4 tests on setting of pulse power cutoff, in which the processor will lose sustained energy supply suddenly for a very short time after a long interval periodically. In this case, if the energy supply is not very sufficient, it is only able to back up for a short time running in low frequency for the processor but not for high frequency. In this light, DFS system also precedes two-thres system.

Other features of this system when running in pulse power cutoff energy profile is similar to the above discussion.

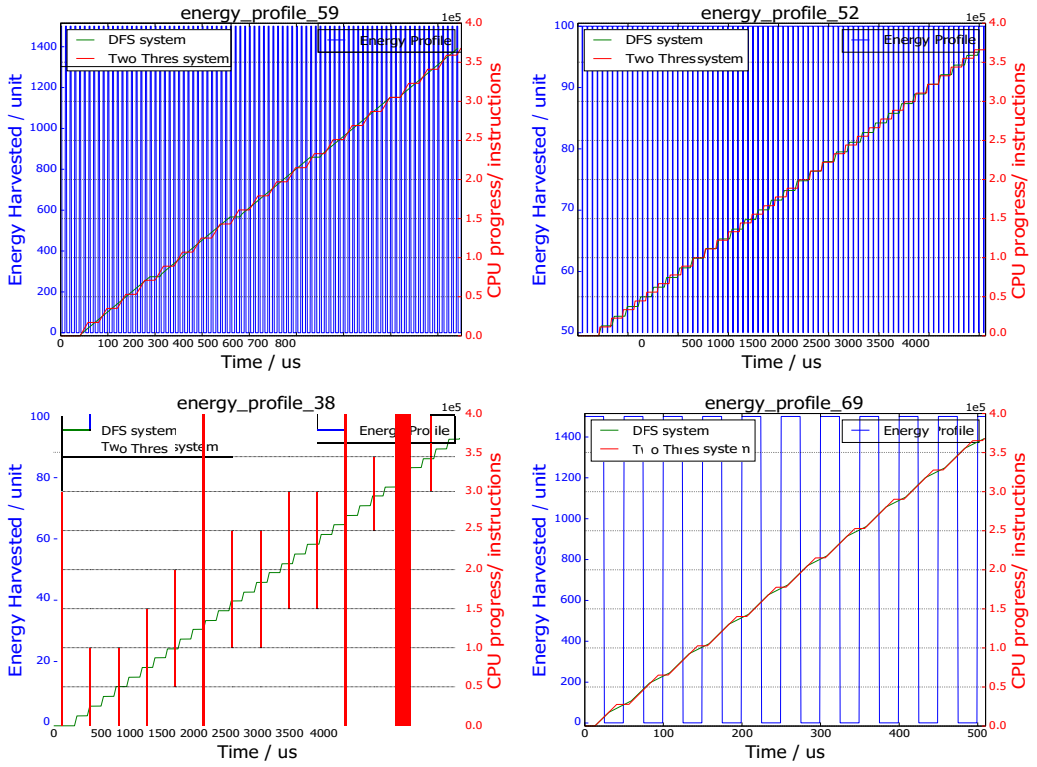

Figure 4: CPU performance under pulse power cutoff. The blue curve is energy profile and the red/green curve is the progress of DFS/two-thres CPU when running the queens program.

To verify our suspicion that the ultimate reason and condition that enable DFS processor system to perform better than two thres system is the kind of case that energy harvested by the processor is nearly enough to sustain its normal running on low frequency while a two-thres machine will fail by close down the machine, we did some other tests on sine curve energy profiles rather than square

wave form. We could inspect from the figure 5 that previous suspicion also works well in this case. That is to say, the green line in the figure keeps increasing with a relatively low slope (matching the low frequency state) while the red line will become flat once in a while for lack of energy to run.

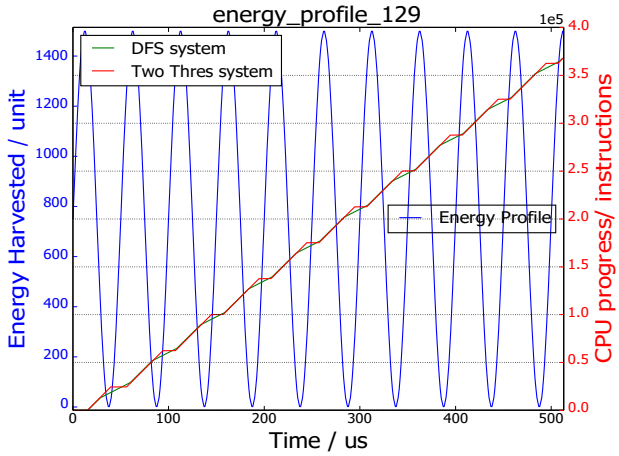

Figure 5: CPU performance under sine function form energy profile. The blue curve is energy profile and the red/green curve is the progress of DFS/two-thres CPU when running the queens program.

4 Conclusion

Building on the recent great progress of implementation of gem5-NVP, we have developed a DFS processor system and show that it is possible to replace traditional two-thres energy management machine with DFS system with three inner states to boost up the performance of system at some specific energy usage conditions. From our discussion above and detailed experiment analysis, we have shown that this DFS system is able to utilize its privileges and outperform two-thres system performance when energy harvested by the processor is nearly enough to sustain its normal running on low frequency while other design running on high frequency will fail by close down the machine. It provides a strong theoretical and experimental support for business development of nonvolatile processors in field of internet of things and others.

5 Acknowledgement

It's been a great journey exploring system and processor simulation techniques and deploying it into a state-of-the-art research domain, Non-Volatile Processor. We've devoted plenty of time into learning related topics as well as debugging codes and environment setting.

Here we give special appreciation to teachers and TA of this course. We couldn't reach such level without their attention and help.

References

Mimi Xie, Mengying Zhao, Chen Pan, Jingtong Hu, Yongpan Liu, and Chun Jason Xue. Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor. In *Proceedings of the 52nd Annual Design Automation Conference*, page 184. ACM, 2015.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.