清 华 大 学

综 合 论 文 训 练

题目：非易失处理器的仿真平台设计

系　　别：电子工程系

专　　业：电子信息科学与技术

姓　　名：张乐凡

指导教师：孙忆南　工程师

2017 年 6 月 16 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：张乐凡　导师签名：孙忆南　日　期：06/12/2017

中文摘要

物联网发展使得处理器节点被放置在更复杂的环境中，这些节点缺少维护人员与电源供给，因此需要从环境中收集能量，并面临频繁的断电与重启，因此，能够在断电时保存处理器状态的非易失处理器成为了研究的热点。然而非易失系统缺少一个准确易用的软件仿真平台，研究者往往需要使用电路级仿真工具，这极大提高了研究非易失处理器的门槛。本文基于 Gem5 体系架构仿真平台提出了一个简单易用的非易失系统软件仿真平台 Gem5-NVP，此仿真平台对硬件的能量行为进行了深入的建模，为每一个模块扩展了能量行为，且在模块间搭建了能量通信协议与接口，这使得此仿真平台有能力对任意模块的不同行为进行仿真，这些模块包含 CPU、内存、甚至外设。较前人工作而言，这个平台拥有较好的用户扩展性，并且是世界上首个能够在行为级对非易失系统中的外设模块进行建模的软件仿真平台。此外，为了验证此软件仿真平台的准确性和易用性，本文给出了此仿真平台的一些实验结果。

关键词： 非易失处理器；仿真器；物联网；Gem5

ABSTRACT

More processors are placed in complicated environments as Internet of Things (IoT) develops. Those processors have neither maintenance nor power supply, and therefore need to harvest energy from the environment, which will cause frequent power failures and recover processes. Thus, non-volatile processors (NVPs) which are able to preserve run-time status at power failures become a hot spot of present research. However, researchers do not have a software simulating platform and rely on circuit-level simulating tools in most times, which significantly increases the difficulty of developing NVPs. In this paper, Gem5-NVP, an easy-to-use software simulating platform of non-volatile systems based on Gem5 simulator, is introduced. Gem5-NVP explores energy behaviors of real-world modules and develops a software energy model for Gem5's objects which enables objects to have their own energy behavior and provides an energy-communicating protocol for them to transmit energy messages to each other. Based on the characteristics mentioned above, the simulator is able to simulate modules of a large variety, including CPU, memory and even IO device. It has an extendibility for users comparing to previous work, and becomes the first software simulator with action-level modeling of IO devices. Moreover, to verify the capability of Gem5-NVP to explore the design of non-volatile systems, some experimental results are provided in this paper.

Keywords: NVP (Non-Volatile Processor); Simulator; IoT (Internet of Things); Gem5

目录

第 1 章 引言 1

1.1 物联网（IoT）发展情况 1

1.2 非易失处理器 2

1.3 非易失处理器验证方式 5

1.4 当前验证方式的不足 6

1.5 毕业设计内容 10

第 2 章 软件仿真器架构 12

2.1 软件采取的 NVP 整体架构 12

2.2 Gem5 类继承关系 13

2.3 为 Gem5 的仿真模块引入能量相关功能 14

2.4 Python 控制端 15

第 3 章 模块间能量信息交互 16

3.1 “能量接口” 16

3.2 能量接口与模块（SimObject）的关系 18

3.3 Python 端配置 20

第 4 章 能量管理模块 22

4.1 能量管理模块简介 22

4.2 能量收集功能 24

4.3 系统状态机模块 25

4.4 能量管理模块类成员介绍 26

4.5 Python 端配置 27

第 5 章 外设行为建模 28

5.1 虚拟外设简介 28

5.2 虚拟外设工作流程 29

5.3 外设地址解析 31

5.4 Python 端配置 31

5.5 程序端配置 32

第 6 章 测试与仿真 33

6.1 DFS 系统仿真 33
6.2 非易失外设仿真 35
第 7 章 结论 38
插图索引 39
表格索引 40
参考文献 41
致谢 43
声明 44
附录 A 外文资料的调研书面翻译 45

第 1 章 引言

1.1 物联网（IoT）发展情况

物联网的是近几天提出的一个新兴概念，其定义如名称所说，是将世界上各种各样的物品通过一定方式接入网络。这些物品可以包含我们日常个人生活中涉及的物品，比如说起搏器、运动鞋、体温计等等，也可以包含社会生活中涉及到的物品，比如说自动贩卖机、售货机、物流车辆等等，物联网甚至可以将一些自然界的事物通过某种方式接入网络，比如说江河中的水流。物联网可以使这些物品分享自身数据，互相协助，以便更好完成某些目标[1]。

物联网的诞生为人们提供了新的可能性。比如说在医疗健康领域，各式各样的医疗传感器使得人们能够在自己的家中获得完整准确的身体情况信息，网络上的有关医疗健康的服务商可以根据这些身体情况对可能的问题做出预警，对于病人来说，他们能够提早发现自身的病情并尽早就医，对于健康的人，他们能够根据自身的预警信息对疾病做出预防，维持在健康状态。在物流领域，物联网使得追踪任意一辆货运车辆、任意一个包裹成为可能，物流公司能够根据这些信息获知每一个物流节点是否顺畅、哪些方向的请求更加密集等信息，这使得物流公司能够进行合理的统筹规划，提高运送效率。在自然中设置的节点也有很大的意义，比如说，山体的内部压力数据可以通过传感器联网，异常的数据可能意味着即将出现山体滑坡等灾害，通过物联网的手段，灾害预防组织能够更早获得这些信息，进行预防并减少灾害。

物联网还给人们提供新的挑战，这些挑战包含网络地址不足、安全和隐私保护、标准化问题等[1]，但其中最为重要的是能量采集问题，这也是本文最为关注的问题。物联网中的节点有体积小、数量多、分布广泛的特点，然而常规电池并不能支持这些设备长时间工作，如果使用电池为这些设备供电，定期维护的成本将会变得十分巨大，因此很多物联网中的节点摒弃了传统电池，采用了在环境中收集所需能量的方式。这些节点收集能量的方式多种多样，有的收集太阳能，有的能够收集动能[2]，更多的则是收集环境中的电磁波的能量[3]。

由于环境能量稳定度不高，随时间变化较大，这些收集环境能量的设备在工作中将频繁面临掉电，因此这些节点中应该使用有别于基于传统供能系统的能量处理技术。

1.2 非易失处理器

本部分将进行非易失处理器（Non-Volatile Processor）的简单介绍，并给出当今非易失处理器结构设计的研究现状。

1.2.1 非易失处理器简介

如上文所述，收集环境中能量的系统将频繁面临断电，传统处理器在频繁开关机时是无法推进任务的，这是因为在频繁掉电时，处理器寄存器与内存中储存的数据将会完全丢失，导致处理器丢失当前状态，这种传统的处理器是不适合工作在物联网中收集环境能量的节点的。为了能够在频繁断电时还能够进行连续计算，一种新的处理器结构，非易失处理器（Non-Volatile Processors, NVPs）被提出[4]，这些处理器与传统处理器相比有如下区别：

1. 这些处理器有能量采集系统

首先，NVP 是从环境中采集能量的，因此 NVP 中存在能量采集器，这些能量采集器是随着采集能量类型变化的，一般来说，采集装置的最外侧是将外界能量转换成能够使用的电能的器件，接下来一般接入一些整流、负载转换使得电流、电压参数能够被处理器所使用。此外，为了保证足够的时间来使得在断电时系统能够保存当前状态，NVP 中往往有能量收集装置（一般是电容）来收集多余的采集到的能量，当系统面临断电时，处理器将会使用这些临时储存的能量来进行备份操作。

2. 这些处理器有状态机来控制系统状态

和传统处理器不同，NVP 需要自行自动控制当前处理器的状态。一个最简单的例子是，当电量充足的时候，NVP 处于工作状态，当电量不足时，NVP 先进入备份状态，再进入休眠状态，当电量足够开机时，NVP 进入恢复状态，接下来进入正常工作状态。当然，这个例子是理想化的，最简单的状态机模型，实际的系统中可能有更加复杂的状态转换模型。

3. 这些处理器有备份-恢复装置与策略

当发生断电时，NVP 需要将当前状态备份到非易失储存器中。Flash 和磁碟由于速度慢、功耗大等原因不适合作为这种低功耗、快速断电的系统的非易失存储装置。近年来有一些新兴器件，如 RRAM[5]，nvFF[6]，能够以接近 SRAM 的延时存放数据，且数据不会随断电而丢失。NVP 中往往有这样的非易失储存阵列，当发生断电时，系统将易失的寄存器和内存中的全部或部分数据储存在这些非易失储存器中，保证系统状态不丢失。

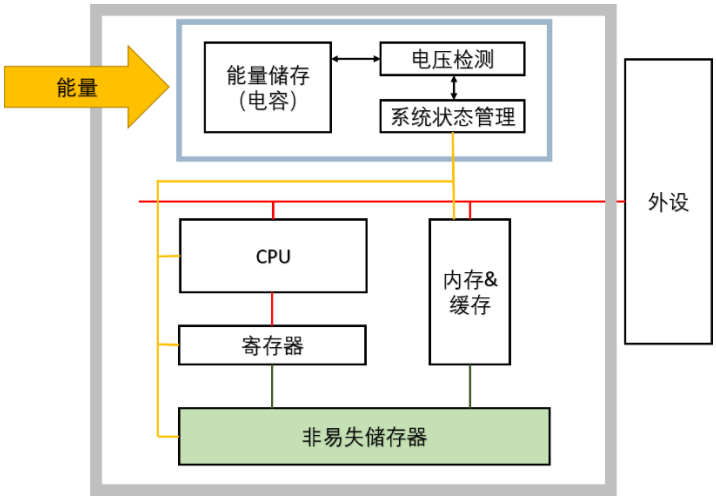


图 1.1 非易失处理器结构

图 1.1 中介绍了一个典型的非易失处理器的结构。非易失处理器有着广泛的研究空间，无论是备份方式、系统状态机设计还是能量管理模块都有很多细节值得研究，下文将会分别介绍非易失处理器的这些研究方向。

1.2.2 非易失处理器备份策略研究

尽管非易失处理器是最近几年提出的概念，但可中断计算一直是一个被研究的话题。在没有非易失处理器的时代，人们需要在传统 CPU 的基础上进行一些软件-硬件上的开发，以便系统能够在断电时不丢失全部工作状态，一个典型的手段是设定记录点（checkpoint）。在记录点中，CPU 将会把需要备份的数据从内存和寄存器搬运到硬盘中。由于传统的处理器并不能够像非易失处理器一样能够保证在断电时有足够的时间和能量备份全部需要备份的数据，设定记录点的方式是值得研究的。论文[7]提出了一种设定记录点的方式，这种方式是纯软件的，文章[7]修改了 LLVM 编译器，使得普通程序在被编译时会被插入一些函数来决定是否进

行备份，这些函数在运行时会根据现有的能量情况估算短时间内是否会发生断电，如果估计的结果为会发生断电，这些函数会触发记录点备份，这样一来，断电重启时系统将会从记录点而不是程序的最开始被启动。

文章[7]中提出的算法的不足是，运行这种算法需要特殊的编译器，需要在软件端编译时修改程序，这给软件开发带来了复杂度，而且，使用事先插入的记录点触发位置并不能保证程序的运行效率，如果触发点位置不佳可能会导致系统在重启后有较大回退，增加程序运行时间。基于这一点，文章[8]提出了另外一种备份策略，这种备份策略是软件-硬件协同的，这种备份策略为系统引入了能量不足中断，当系统能量低于一定阈值时，会触发中断，这时中断处理函数将会进行备份，将系统状态保存到硬盘中，可以看出这种方式已经极为接近当今非易失处理器的备份方式了。

虽然当前非易失处理器的备份方式都是类似的，但仍然有一些细节值得被研究。在非易失处理器中，备份过程的能量消耗和时间消耗仍然是系统性能的瓶颈，如果备份过程的能量消耗过大，则系统需要有更高的掉电能量阈值，这会导致系统更经常发生开关机，降低在特定能量环境下的运行效率。非易失储存器，无论是 nvSRAM，RRAM，还是 nvFF 备份都需要一定能量和一定延时，一种简单直观的想法就是在发生掉电时不备份全部内存，而是只备份最近被使用过的内存块（Lest Recently Used, LRU），降低备份所需能量和时间，然而，测试表明这种方式与全备份相比带来的性能提升不大[9]。在一些研究工作中描述了一种新的备份时内存块选择策略，一种死亡块预测算法（SBDP）被提出，测试表明这种算法能够有效提高备份的能量需求和时间需求[10]。

1.2.3 非易失处理器与外设的交互

尽管非易失处理器能够保证在掉电发生时保存当前状态，但一旦涉及到非易失处理器与外设协同工作，情况将变得复杂，这是因为绝大部分外设是易失的，也就是说一旦在外设工作时发生断电，则需要命令外设重复工作。这种复杂性使得非易失处理器与外设的交互过程需要被研究，这方面的研究存在两个方向，第一个就是研究外设硬件的设计，使得外设硬件成为非易失的，当发生断电、掉电时不需要处理器的干涉就能自行恢复并且继续完成任务，另一个方向是研究处理器调用外设的调度方式，使得处理器能够在特定的外界能量情况下尽可能不让外设任务被断电所打断。

论文[11]是一个典型的处理 NVP 本身与外设协同的工作，在文章中作者将上

电/掉电带来的额外时间计算在内，提出了一种调度外设工作的策略，当同一时间有很多外设任务需要完成时，合适的调度外设的策略使得系统能够在某一特定的外界能量环境下用最短的时间完成所有外设的工作。

论文[12]同样针对外设的调度问题基于一个太阳能收集节点进行了研究，然而在此文章中调度的优化目标并不是在最短时间内完成所有外设工作，而是使得更大比例的外设工作能够在断电之前完成工作，也就是优化长期的 DMR（deadline miss rate）。

1.3 非易失处理器验证方式

以上所有有关非易失处理器的研究方向均有很大的研究空间，当有关非易失处理器的新技术被提出时，应该有准确、合理的手段对这些新技术的性能进行验证。比方说，在涉及非易失处理器备份策略的研究时，研究者可能会关心如下问题：当系统的备份方式从内存、寄存器全备份变为部分备份内存、寄存器时，备份所需能量会发生什么变化，备份时间占用程序运行整体时间发生怎样的变化，部分备份造成的数据缺失会产生程序运行多大的时间增加？事实上部分备份减少了备份所需要的时间，但是能量恢复后的数据缺失可能会要求处理器耗费更多时间来恢复这些数据。这是一个极为复杂的问题，涉及处理器运行的很多方面，外界能量收集情况、处理器运行程序种类可能均会对最终的结果产生影响，这些影响很难通过一个特定的公式来定量描述。因此，仿真是验证新技术性能的主要手段。

由于非易失处理器诞生较晚，非易失处理器的仿真是一个崭新的研究方向，非易失处理器的仿真往往比较原始，即设计好电路再在电路的基础上进行验证。此外，近年来诞生了少许非易失处理器的仿真软件，比如说基于 Gem5 实现的非易失处理器仿真软件 NVPsim[9]。下文会分别介绍电路级仿真以及 NVPsim 仿真器。

1.3.1 电路级仿真

顾名思义，电路级仿真就是用一些硬件描述语言设计好电路，并使用一些特定的数字电路仿真软件对设计出的数字电路进行仿真。这是一种通用的办法，适用于一切数字电路，非易失处理器也不例外。目前多数有关于非易失处理器的研

究基于这种仿真方法，研究者已有一个非易失处理器原型（多数为实验室开发的示例平台），实现其提出功能的硬件电路设计，集成到已有的实验平台上，并使用硬件仿真工具（如 Modelsim[13]）进行仿真。

1.3.2 NVPsim

Gem5 提供了一个高自由度、配置方便的处理器体系架构仿真平台[14]，在此基础上，Gu et. al.拓展了 Gem5 的功能，为 Gem5 提供了仿真处理器、内存在外界能量变化时行为的功能[9]。NVPsim 的作者使用 NVPsim 对使用不同种类非易失储存器、不同备份策略的非易失处理器进行了仿真，并得出了非易失储存器和备份策略对处理器效率影响的一些结论。

NVPsim 基于 Gem5 的 TimingSimpleCPU 模型，加入了电压检测模块、系统状态机、备份/恢复模块，并对 Gem5 事件队列的管理模块进行了一些修改。这些对 Gem5 的增补使得 NVPsim 能够仿真非易失处理器在特定外界能量条件下的性能。

NVPsim 在系统运行过程中，通过 Power Trace 文件获取外部能量收集情况，并随时追踪系统的各个模块耗电情况，当能量不足时，系统状态机告知事件队列暂停当前的工作，并通过备份/恢复模块对需要备份的数据进行备份。当关机系统收集够充足能量时，系统状态机告知事件队列继续运行，并通过备份/恢复模块对内存、寄存器中得到备份的数据进行恢复。NVPsim 在仿真过程中主要的关注点在于备份、恢复所需的能量和时间，在备份、恢复模块对能量和所需时间进行集中计算，而系统在上电、掉电时的行为主要是由事件队列管理模块完成的。NVPsim 相比电路级仿真能够大大降低仿真的难度。

1.4 当前验证方式的不足

无论是电路级仿真还是使用 NVPsim 仿真软件进行仿真均存在不足之处，本部分会分别分析二者的不足。

1.4.1 电路级仿真的缺陷

虽然电路级仿真有着仿真结果准确、与实际系统误差几乎可以忽略不计等优点，但是这种仿真方式过于通用，并没有对非易失处理器进行专门的设计，因此，这种仿真方式给使用者造成了巨大的不便，极大增加了仿真的工作量。以下为电

路级仿真的缺陷：

高门槛

首先，电路级仿真需要有完整的非易失处理器电路设计才能够完成。研究者为了研究非易失处理器中某一些特定部分的行为，往往需要拥有整个非易失处理器的电路设计。当前非易失处理器问题的研究者通常需要研究小组前期对电路设计有所准备。为了探索 NVP 的特定部分，研究过程为：

非易失处理器原始平台设计（prototype）-原始平台仿真、流片-修改原始平台需要研究的部分-测试、仿真

这使得世界上只有少数拥有 NVP 原始平台（prototype）的研究小组有能力对非易失处理器技术进行研究，大大增大了非易失处理器研究的入门门槛。

我们可以将其与传统处理器体系架构的研究进行对比，传统处理器有一些通用、公开的设计，并有着大量方便的仿真工具，在对体系架构进行研究时，研究者实际上并不需要拥有完整的处理器电路设计。

配置不方便

在使用电路级仿真时，任何微小的修改往往都会对电路设计产生巨大的影响，研究者为了探索新技术时往往需要深入硬件描述代码并进行大量修改。仍以掉电时内存备份策略研究为例，为了将系统在掉电时的备份方式从全备份改为部分备份，通常来说硬件描述语言中整个备份模块都需要被重写，备份方式的改变还会对电路的其他部分造成影响，比如说备份/恢复需要的线路宽度可能会发生变化。

在探索非易失系统中的新技术时，研究者并不希望微小的修改牵涉出复杂、混乱的系统变化，而是希望仅仅修改一些参数就能够对新技术的性能进行仿真，显然电路级仿真并不能做到这一点。

仿真时间过长

在使用电路级仿真时，仿真软件需要模拟出系统的每一个引脚在每一时刻的信号值，这会使仿真的时间开销变得巨大。而非易失处理器是非常复杂的电路，有着大规模集成芯片的等级，此外，为了能够获得准确的仿真结果，仿真使用的测试程序（benchmark）所需周期数往往不低。

非易失处理器仿真的这些特点会进一步增加仿真所需的时间，使得每一次电路级仿真的时间变得难以忍受。而电路级仿真的优点——对电路细节描述的全面与准确往往并不是被关注的对象，研究者关注的可能仅仅是运行测试程序（benchmark）所需要的时间和能量效率。

软硬件接口复杂

使用电路级仿真时，被仿真的对象是硬件本身，缺少一个简单、易用的软硬件接口，因此，使电路级仿真运行研究者关心的测试程序（benchmark）往往会消耗额外的时间。当前已有的非易失处理器使用的并不是大规模商业化的架构（如x86、ARM），比如说 THU1010n 非易失处理器芯片使用的是 8051 架构[15]，这些架构可能存在一些编译器，但并没有针对非易失平台特殊编写的编译器，因此研究者在电路仿真中运行特定程序往往需要手动或者用一些工具将程序翻译成机器语言并以硬件的形式写入硬件描述语言中，这显然是十分复杂的。

在理想情况下，使用高级语言（如 C）编写的程序就能够运行在仿真平台上，这会大大降低仿真非易失处理器行为的难度，然而电路级仿真并不能够支持这一点。

1.4.2 NVPsim 的缺陷

NVPsim 继承了 Gem5 仿真平台的一些特点，因此避免了上述电路级仿真带来的不足。比如说 NVPsim 像 Gem5 一样支持运行 ARM、x86 等流行的程序，这使得用户能够使用高级语言编写测试程序（benchmark）并使其运行在仿真平台上。NVPsim 也忽略了电路中过于细节的部分，只描述系统模块的行为，这大大增加了 NVPsim 的仿真速度。NVPsim 还使得用户能够通过修改少量仿真参数直接改变目标非易失处理器的行为，降低了仿真的配置难度。

然而，NVPsim 仍然存在着一些不足，使得其并不能成为一个广泛使用的非易失处理器仿真平台。NVPsim 存在着如下不足：

对硬件的能量行为描述不自由

NVPsim 对系统的行为采用了“集中式”管理办法，并没有对每一个模块分别编写在系统状态发生变化（上电、掉电）时的行为。NVPsim 在系统状态发生改变时改变模块行为方式的唯一渠道是事件队列管理模块，当系统发生掉电时，事件队列管理模块读取 Gem5 的事件队列，将所有事件暂停，这样一来系统就停止了运行，而当系统发生上电时事件队列管理模块再将所有事件重新调度使得系统恢复运行。

这种对掉电、上电的建模方式虽然有效，但是缺少对硬件在系统状态改变时行为描述的自由度。NVPsim 将系统中所有的掉电、上电时的行为描述都集中在了一起，在上/掉电时只对系统整体的事件队列进行操作，而不对每个模块分别进行操作，这样一来所有的模块在系统状态变化时都表现出了相同特性：暂停运行。

事实上，每一个模块在系统状态改变时的行为是不同的，易失的模块应该丢

弃当前内部的所有状态，非易失模块应该保持内部状态不变，而另一些模块比如说电压检测模块、系统状态机模块应该继续工作，不受掉电、上电的影响，显然NVPsim 对这样行为的描述是不足的。

对正确性验证的先天不足

上文描述了 NVPsim 的工作流程和对掉电、上电的建模方式，这种建模方式仅仅暂停了硬件触发的所有事件，实际上默认了所有模块都是“非易失”的，也就是说，掉电时所有模块的内部状态都没有发生改变，仅仅是暂停运行而已。NVPsim 并没有提供对硬件易失性的描述，也没有为编写者提供描述硬件易失性的接口。

实际上非易失系统中还是存在某些易失模块的，比如说 CPU 内的寄存器或者传统的内存模块再掉电时应该丢失储存的全部数据。这一点不足使得 NVPsim 只能描述非易失系统的运行时间、消耗能量等信息，而并不能描述非易失系统在掉电-上电过程中运行结果是否正确。

对外设缺少描述

一般来说，非易失系统都是从环境中获取电能，会经常面临电力不足，而且获取的电能十分有限，因此非易失系统一般并不适合做大量计算（只涉及处理器、内存），相反，非易失系统的工作任务常常是作为终端节点收集环境中的数据并发送给服务端（这种工作任务设计处理器，内存， 以及传感器与网卡等外设）。因此，对于非易失处理器和外设的交互行为的仿真往往是仿真器功能的重中之重。遗憾的是，NVPsim 并没有对外设进行合理有效的仿真。

NVPsim 基于 Gem5 的 Syscall Emulation（SE）模式，在这样一种模式下 Gem5 并不提供任何外设功能，而是将程序运行中产生的对外设的请求（一般包含在系统调用，system call 中）直接发送给仿真器运行所在的系统（linux）。这样一来，在使用 NVPsim 的过程中，用户只能够仿真处理器和内存的行为而不能触及外设，在某种程度上这使得 NVPsim 有着巨大的局限性，应用场景不足。

扩展性较弱

从工程的角度来说，NVPsim 并不像 Gem5 本身一样有着良好的可扩展性，这是由于 NVPsim 并没有对所有模块在掉/上电的行为进行通用的建模并进行合理的分层抽象，而是将整个系统掉、上电行为直接通过修改底层事件队列管理代码来进行描述。当用户需要和 NVPsim 不同的系统建模时，NVPsim 并不能提供有效的接口让用户方便地修改、扩展系统的行为，用户在这种情况下需要大量阅读了解底层代码，十分不便。在这个意义上 NVPsim 更像是一种特定的非易失处

理器的“专用仿真器”，而不是一个通用的仿真平台。

1.5 毕业设计内容

本次毕业设计的任务是基于 Gem5 设计一种非易失系统的仿真框架（或仿真平台）：Gem5-NVP，从而解决上文提到的当前仿真方式的不足。此仿真框架设计有如下目标：

1. 对所有模块在系统状态改变时的行为有通用的接口

这需要每一个模块都能够通过重定义相同的接口函数来完成对这个模块掉/上电行为的建模，且用户可以像添加 Gem5 模块一样简便地添加 Gem5-NVP 的模块，仅仅需要定义模块在系统状态发生变化时的行为。这个目标主要针对的是仿真框架中模块行为描述的自由性和仿真框架的可扩展性。

2. 支持外设的仿真，且易于从代码中调用

这个目标要求仿真平台中存在可以在 Gem5 仿真器 SE 模式下使用的外设模块，且这个模块要能够方便地被用户编写的测试程序（benchmark）调用，不牺牲编写 benchmark 的简易度。

3. 易于配置，使用户远离复杂的底层代码

这个目标要求仿真框架中的一些参数、模块能够像 Gem5 中的参数、模块一样通过简单的 Python 配置文件进行配置。

为了完成以上目标，本次毕业设计的主要工作任务有：

1. 对 Gem5 的所有模块的底层通用类进行重写，使得所有模块都获取描述有关能量行为的接口。
2. 为 Gem5 添加模块间的有关能量的“连线”（能量接口），即 Energy Port，这使得能量本身和有关能量的系统状态控制流（上电、断电请求）能够在模块间传递。此外，用户应该能够在 Python 配置文件中方便地连接这些连线（类似 Gem5 的内存系统中的 Port）。
3. 添加能量管理模块，这是一个非易失仿真器所必须的，这个能量管理模块要负责能量采集、系统状态管理、能量检测。
4. 添加仿真器的外设模块，这个外设模块需要能够在 SE 模式下运行，且可以使用高级语言非常方便地调用。

5. 基于此仿真平台（我们称之为 Gem5-NVP）对各种非易失系统进行建模、测试（比如不涉及外设的计算系统、涉及外设的信息采集系统、涉及外设的信息发送系统）。

第 2 章 软件仿真器架构

2.1 软件采取的 NVP 整体架构

软件仿真器 Gem5-NVP 的架构是建立在 Gem5 的系统调用仿真（System Emulation）模式的基础上的，因此其基本模型和 Gem5 的 SE 模式的结构类似。Gem5 拥有较为自由的模块连接方式，支持用户自定义模块行为，每一个模块拥有“内存”接口，能够向其他模块发送内存访问请求，包含地址、可能的写入值、请求大小等。尽管 Gem5 支持多种多样的模块连接方式，多数模型使用的就是传统计算机使用的体系结构，即中央处理器-总线-内存的结构。Gem5-NVP 不需要对这个结构进行修改，所需要做的是在这个结构的基础上添加有关非易失的模块和功能。

整个软件仿真系统的框图如图 2.1 所示：

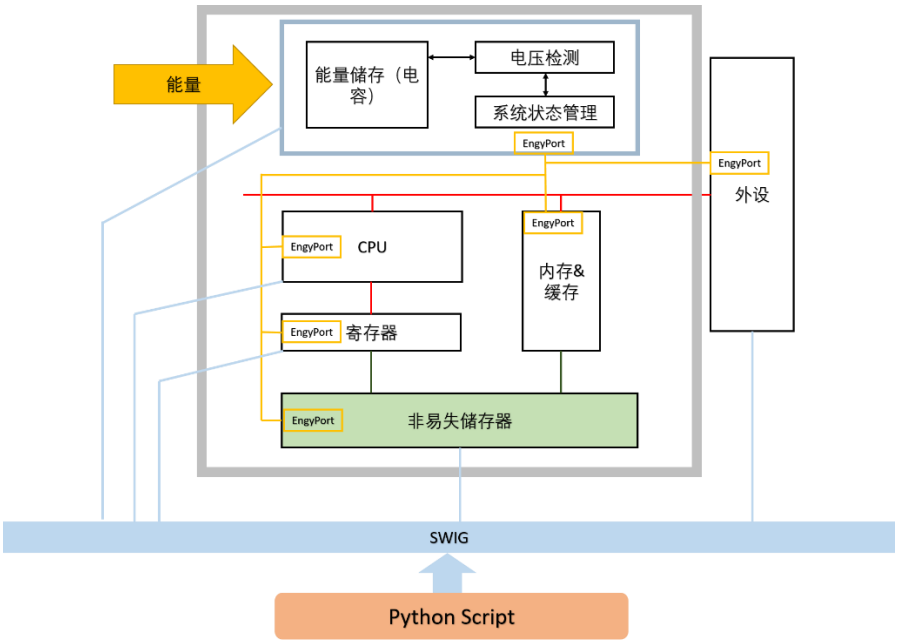


图 2.1 软件仿真系统结构

首先我们需要添加传统体系架构中不存在的能量采集和能量管理状态机模块，这部分模块负责收集能量，并在需要发生系统状态改变时通知/控制其他模块进行状态改变。这个模块的主要任务一是仿真从外界能量幅度（电压、电流、光照）到内部能量储存器的转换电路的行为，二是仿真系统能量状态机不断检查系统储存电压值并控制系统状态改变的行为。

此外，我们需要添加的是 Gem5 中不存在的能量信息通信功能，在非易失系统中，模块间有着很多必需的能量通信，比如说，能量管理模块需要通知系统其他模块断电或者上电，各个模块需要告知能量管理模块自身消耗的能量等，因此，我们需要为模块添加与其他模块进行通信的功能，即每一个模块在“内存”接口之外还需要有“能量接口”。

最后，由于 Gem5 的 SE 模式下不支持外设的行为，但我们的确在非易失系统中需要大量使用外设，为了日后仿真方便我们需要开发一个通用的外设模块来仿真非易失系统中外设的行为。

2.2 Gem5 类继承关系

图 2.2 描述了 Gem5 的类继承关系。从中可见 Gem5 中的每一个模块均为一个 SimObject 对象[14]，SimObject 对象为模块定义好了一些模块通用的功能。这些功能来源于 SimObject 的三个父类，分别为 EventManager、Serializable 和 Drainable。其中 EventManager 为 SimObject 提供了操作异步事件队列的功能，使得 SimObject 能够方便地在事件队列中添加、取消或者修改自身在其中插入的事件；Serializable 提供的功能是使得任何 SimObject 能够将当前状态输出为串行比特流，这个功能的作用是能够使得 SimObject 能够设置 checkpoint，当用户关心运行时某个状态时，能够将系统涉及到的所有 SimObject 的状态均保存成比特流形式的 checkpoint，日后的运行就不需要再次从头执行，只需读入 checkpoint 即可；Drainable 为 SimObject 提供的功能是能够使得 SimObject 能够正确结束自身的工作状态。

Gem5 通过以上方式为所有模块的开发打下了基础，在 SimObject 的基础上开发者开发了各式各样的类，典型的类有各种各样的内存类（MemObject），包含了各种建模的处理器（CPU）、各种建模的内存模块（SimpleMemory、DRAM 等等）

以及总线模块（CoherentXBar），这些模块的共同点是都需要向其他模块传输内存访问、读取、写入信息，事实上 MemObject 正是提供了这样的“内存”接口，正如“软件采取的 NVP 整体架构”一章中所说。

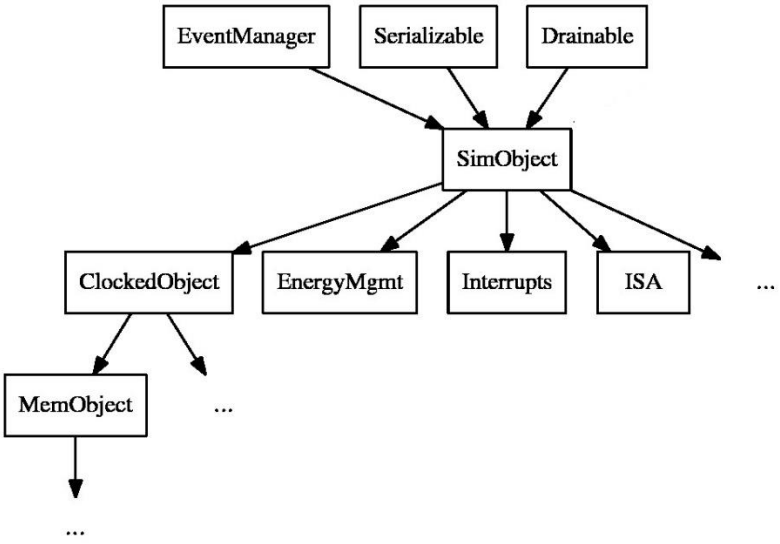


图 2.2 Gem5 类继承关系

2.3 为 Gem5 的仿真模块引入能量相关功能

在开发 Gem5-NVP 的过程中，一个内在的逻辑就是任何模块都应当有消耗能量的接口，同时应当能够接收系统有关能量的通知（如“开关机”等等），Gem5-NVP 不应该只针对少数我们关注的模块开发能量消耗和开关机功能，而是应给为所有模块提供通用的能量操作功能，基于此，我们认为和能量相关的功能同 EventManager、Serializable、Drainable 提供的功能处于同样的地位。为了实现这些要求，我们引入了“能量对象”（EnergyObject）作为 SimObject 的父类之一，如图 2.3 所示。

EnergyObject 类为模块提供了一个一些和能量有关的接口，这些接口将会在第三章中介绍。

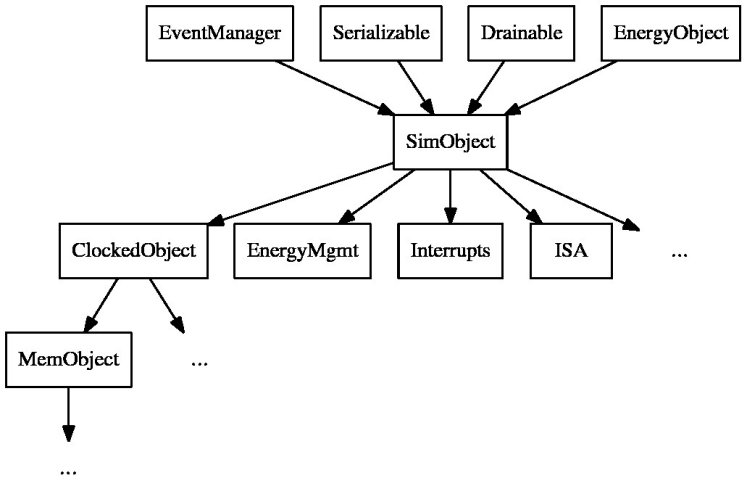


图 2.3 能量模块 EnergyObject

2.4 Python 控制端

除了上述 C++端功能之外，Gem5 还使用了 SWIG（Simplified Wrapper and Interface Generator）来对 SimObject 提供了对应的 Python 端接口，任意从 SimObject 中派生的类在 Python 端都有一个对应的同名类，在实际使用中，用户一般直接通过配置 Python 端对象的方式来配置这个 SimObject 的行为，为了仿真某一系统，用户需要为这个系统创建一个 Python 脚本，在这个脚本中定义这个系统中被使用的模块，配置这些模块的参数，并按照实际系统连接这些模块，这些模块（SimObject）存在一定的树状关系，比如说，CPU、总线、内存都是 system 模块的子模块，具体体现就是这些模块的对象是 system 模块对象的成员变量，定义好整个系统的参数和连接方式后，用户需要使系统的最上层 system 模块成为根模块 root 的成员变量，并告知 Gem5 对 root 模块进行仿真。在运行时，Gem5 会采用深度优先搜索的方式遍历模块树，为每一个 Python 模块建立对应的 C++后端模块，接下来同样采用深度优先搜索的方式来为每一个 C++后端模块进行初始化，在初始化的过程中，最初的事件被放入事件队列，这使得整个仿真过程开始。

Gem5-NVP 中和能量有关的功能同样拥有从 Python 端进行控制的功能，提供了众多方便的 Python 接口，然而每一项功能的配置方式都有所不同，正如 Gem5 本身的诸多模块一样，和能量有关的 Python 配置细节将会在后几个章节中被分别详细介绍。

第 3 章 模块间能量信息交互

模块间能量信息的传输是一个软件非易失仿真器所有功能的基础，这一个部分包含了如下内容：模块间交换的能量信息有哪些类型，如何为模块添加能量信息传递的功能，模块之间的能量流路线是如何连接的。这一章节将会对这些问题提出解释。

3.1 “能量接口”

Gem5 中存在内存接口“Port”，描述了系统中的各个模块如何与其他模块进行内存访问相关的通信，借鉴这个概念，Gem5-NVP 中提出了“能量接口”这一概念，能量接口和内存接口存在诸多相似之处，也有一些不同，这一部分将对能量接口进行介绍。首先，能量接口在后端 C++代码中体现为“EnergyPort”类，能量接口能够互相连接，并能够互相发送信息。

3.1.1 能量信息

当需要发送信息时，能量接口会将如下简单的数据结构传送给对面的能量接口：

```
struct EnergyMsg
{
    int32_t type;
    double val;
};
```

此数据结构中 type 代表着此能量信息的类型，一般来说，type 为 0 意味着此能量信息代表发送端消耗了数值为 value 的能量，而其他数值的 type 对应的消息类型一般由能量管理模块 EnergyMgmt（后文中会进行介绍）中的系统状态机进行定义，一个最简单的例子是，系统默认的简易状态机的消息种类有两种，type=1 代表系统关机消息，type=2 代表系统开机消息，用户仿真的系统如果有其他状态或者其他信息，可以自行编写系统状态机并进行定义。

3.1.2 能量接口种类

在仿真能量相关系统过程中，不同模块所处的地位不同，例如，常规模块的地位都是能量的消耗者和能量消息的接受者，而能量管理模块是能量的采集者、消耗者和能量消息的发送者。这两种模块的能量信息处理方式是不同的，因此能量接口种类应该有两种，事实上能量接口有两种，主接口（MasterEnergyPort）和从接口（SlaveEnergyPort）。

在连接时，主接口和从接口之间能够相互连接，一个主接口可以连接多个从接口，但是一个从接口只能有一个连接的主接口，也就是说，主接口和从接口之间的连接关系是一种“一对多”关系。此外，主接口和从接口拥有的功能也不同，由于主接口是管理者，管理多个从能量接口，因此它能够“广播”能量信息，将信息发送给自己管理的所有从能量接口，而从能量接口应该可以告知单个主能量接口能量信息（如自身消耗了能量）。

表格 3.1 和表格 3.2 中描述了两种接口共有的和特有的成员变量和成员函数。

表 3.1 能量接口共有成员变量

成员变量	描述
port_id	此接口的接口号，调试用
port_name	此接口的名称，调试用
port_type	此接口的类型（主/从），调试用
owner	此接口的所有者，应该是一个 SimObject 对象，用于对象调用接口或者接口通知对象

表 3.2 能量接口共有成员函数

成员函数	描述
getPortId/setPortId	读出/写入接口号
getPortName/setPortName	读出/写入接口名称
setOwner	告知接口的所有者（SimObject）
handleMsg	处理接口收到的消息，处理方式为将消息告知接口的所有者（owner）

表格 3.3 从能量接口成员变量

成员变量	描述

master	此从接口对应的主接口

表格 3.4 从能量接口成员函数

成员变量	描述
setMaster	告知接口对应的主接口
singalMsg	向对应的主接口发送能量消息

表格 3.5 主能量接口成员变量

成员变量	描述
slave_list	此主接口的所有从接口队列

表格 3.6 主能量接口成员函数

成员函数	描述
bindSlave	将某一从接口放置在此主接口的从接口队列中，绑定从接口
broadcastMsg	向自身管理的所有从接口发送消息

3.2 能量接口与模块（SimObject）的关系

从上文的表格中可以看出，能量接口中存在“owner”变量，可以将其接收到的能量信息上传到上层的模块（SimObject）来处理，这是能量信息自下而上的传递方式，那么能量信息是如何自上而下从模块传递到能量接口的？上文中写道“能量模块”（EnergyObject）是为 SimObject 提供能量相关处理函数的父类，事实上，“能量模块”（EnergyObject）就是通过拥有能量接口的方式来使得模块（SimObject）所需要发送的能量信息传递到下层的能量接口并发送给其他模块的。在 Gem5-NVP 中，每一个“能量模块”（EnergyObject）拥有两个能量接口（EnergyPort），在两个能量接口中有一个是主能量接口，有一个是从能量接口，这样设计的原因是每一个模块既有可能成为能量的消耗者又能够成为能量的管理者，比如说，能量管理模块采集能量，控制系统状态的同事，自身也在消耗能量，对于这种模块，其自身的从能量接口会连接到自己的主能量接口上。

除了拥有两个能量接口之外，EnergyObject 还拥有一些成员函数来处理从能量接口中获得的能量消息或者发送某一些能量消息，表 3.7 和表 3.8 中列举了和能量模块（EnergyObject）中和能量接口相关的成员变量和成员函数。

表 3.7 能量模块成员变量

成员变量	描述
_seport	这个模块拥有的从能量接口，这个接口被模块用来在消耗能量时向其能量管理单元发送能量消息或者从能量管理单元获得系统能量状态变化消息
_meport	这个模块拥有的主能量接口，使用这个接口时，模块是作为管理者出现的，也就是接受其他模块的消耗能量消息或者发送系统状态改变消息，一般来说，只有能量管理模块（EnergyMgmt，后文会介绍）会用到这个能量接口

表 3.8 能量模块成员函数

成员函数	描述
getSlaveEnergyPort	获取模块的_seport，在模块初始化时用来连接各个模块的能量接口用
getMasterEnergyPort	获取模块的_meport，在模块初始化时用来连接各个模块的能量接口用
consumeEnergy	这个函数的作用是告知此模块的能量层面的管理者这个模块消耗了一定能量，这个函数会通过模块拥有的_seport 发送消息给管理模块。注：为什么没有单独提取出发送能量消息的函数？这是因为一个模块拥有主从两个能量接口，能量消息有可能从这两个接口中的任何一个发送出去，但是消耗能量的能量消息一定是从_seport 发送出去的，因此我们只提取出了消耗能量这一函数。
handleMsg	处理其他模块通过能量接口发送过来的能量消息，当模块拥有的能量接口接收到消息时，将会通过此函数通知模块，然而这个由于每一个模块的处理方式不同，这个函数实

际是一个虚函数，需要在具体的模块（比如某种建模的 CPU）中被重写。

3.3 Python 端配置

定义好了底端接口和代码之后，面临的问题就是如何连接这些模块，或者说，如何得知一个主能量接口连接了哪些从能量接口，一般来说，这种配置信息 Gem5 都是通过 Python 端（前端）来定义的，能量模块的连接方式同样也使用这种方法。编写 C++的连接两个能量模块的函数极为简单，只需要获取二者对应的能量接口并且连接就可以了，我们可以使用 SWIG（Simplified Wrapper and Interface Generator）将这个 C++函数包装成 Python 函数，来在 Python 端进行调用，然而，这种方法存在着很严重的问题。上文提到了，在完成 Python 端配置的最后 Gem5 才会初始化所有的后端（C++）对象，也就是说在运行仿真的配置脚本时 C++对象还没有被分配到内存中，如果直接调用这个 swig 函数的话，将无法获取对应模块（SimObject）的内存地址，造成错误。如果真的需要调用这个函数的话，需要在 Python 的开始仿真函数的初始化 C++对象后进行调用，这对于用户来说十分复杂，而且会破坏代码的层次性，将后端实现暴露给用户。为解决这个问题，可以在 Python 端引入“能量接口引用”类 EnergyPort（在 Python 端的类，有别于 C++端实现），这个类的作用是作为后端 C++能量接口的引用，这个类同样派生出主能量接口引用和从能量接口引用。在 Python 端同样每一个 SimObject 拥有一个主能量接口引用（命名为 m_engy_port）和一个从能量接口引用（命名为 s_engy_port）。

```
self.__dict__['m_energy_port'] = MasterEnergyPort(self)
self.__dict__['s_energy_port'] = SlaveEnergyPort(self)
```

我们可以直接为 Python 端的引用编写函数来相互连接，当我们在 Python 端连接两个引用，引用内部会记录此接口都连接了哪些其他接口，这样一来，我们就可以在 Gem5 初始化 C++端对象时根据引用记录下的信息来相互连接。

为了进一步使用户连接能量接口变得便捷，我们可以直接介入 Python 端 SimObject 的内建函数__setattr__，这个函数能够自定义 Python 为对象的成员变量赋值时的行为，可以在这个函数里判断被赋值的是否是能量接口引用，如果是的话则自动把赋值行为变为连接能量接口行为，这样一来，就可以通过如下简单的代码连接两个模块的能量接口（假设为 cpu 的从能量接口和 engy_mgmt 模块的主

能量接口）：

cpu.s_engy_port = engy_mgmt.m_engy_port

第 4 章 能量管理模块

本部分将会介绍为非易失处理器（NVP）引入的能量管理模块，能量管理模块是非易失系统与传统系统区别的一个重要部分，接下来本文将会从简介、能量收集、系统（能量）状态机几个角度来对能量管理模块进行介绍。

4.1 能量管理模块简介

能量管理模块（EnergyMgmt）如图 4.1 所示，它在系统中扮演的角色是收集能量、储存收集到的能量、不断采集当前能量储存器（电容）的电压，根据能量状况控制系统的其他模块进行启动、备份、恢复、断电等状态变化。能量管理模块会接收用户提供的能量配置文件（energy profile），这个文件内部的内容代表着环境中与非易失系统能量收集相关的物理量随时间的变化，这个物理量可能是光强、电压、或者是震动幅度，用户需要为能量配置文件定义时间单元，这代表着能量配置文件中两个相邻条目的采集时间差。能量配置文件是非易失仿真器运行所必需的，是非易失系统中能量管理模块采集能量的依据。

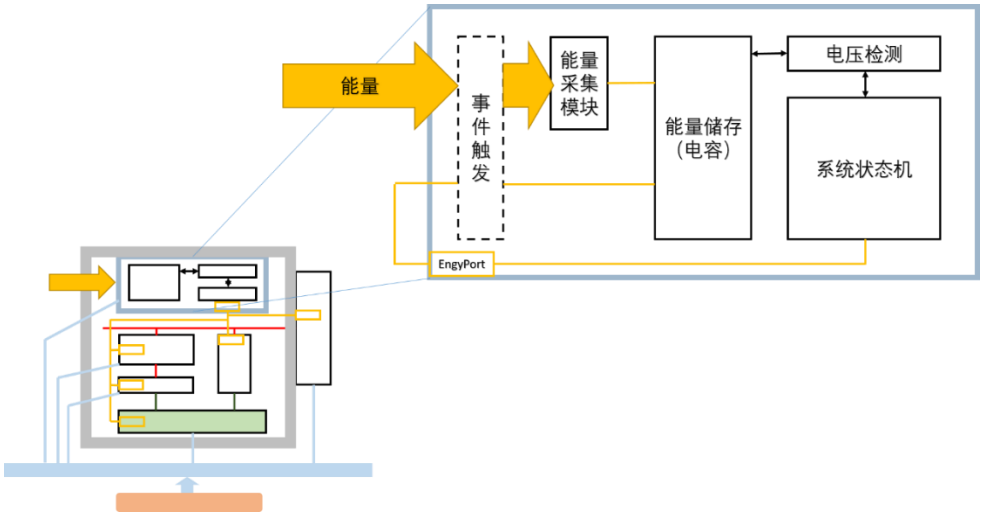


图 4.1 能量管理模块

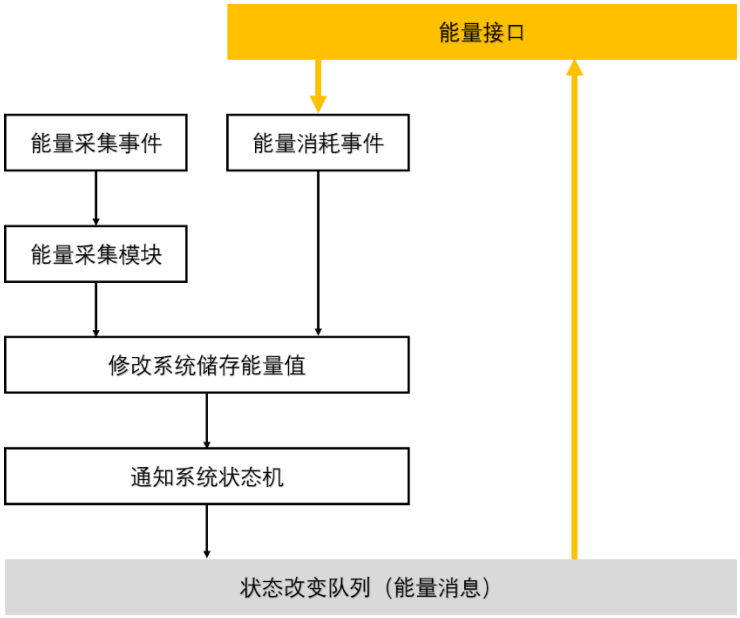


图 4.2 能量管理模块工作流程

能量管理模块的工作模型如图 4.2 所示，从图中可以看出，能量管理模块的工作事件有两条触发路线：

第一条触发路线是能量的收集，收集能量的事件是周期性触发的，也就是说用户可以规定一定的时间作为能量管理模块的能量收集时间单元，每经过这样一段时间，能量管理模块将从能量配置文件（energy profile）中读入一个外界环境能量强度条目，得到此时外界的能量强度，接下来能量管理模块将读取此时电容的电压值，并将外界能量强度和内部电容电压值两个参数告知能量收集模块，能量收集模块计算出收集后电容电压返回给能量管理模块，能量管理模块修改电容电压值，并将电压值的改变告知系统状态机。

第二条触发路线是能量管理模块所管理的其他模块消耗了能量，如上一章所说，模块可以通过触发“consumeEnergy”函数来消耗能量，一旦调用了这个函数，这个消息将会通过能量接口传送给能量管理模块，能量管理模块将所消耗的能量从电容中扣除，计算出电容电压的变化，并且将电压改变告知系统状态机。

根据上述描述，能量管理模块负责有关能量管理的整体流程控制，一些具体的工作，比如外界能量（能量配置文件 energy profile 中的条目）是如何转换成电容中的能量的，或者系统状态是如何改变的，是由能量收集模块和系统状态机模块控制的，下文将分别具体地介绍这两个部分并介绍整个能量管理模块的 Python 端配置方式。

4.2 能量收集功能

在非易失系统中，都会存在较为复杂的电路来完成能量收集的工作，这些电路将会将环境中的待收集的目标能量首先转换成电能，再经过一定的整流、变压、负载匹配等步骤转换成能够储存到能量储存装置（一般是电容）中的电能（图 4.3）[4]。Gem5-NVP 是一个行为级仿真器，因此无意对复杂的电路细节进行仿真，仅此忽略了电能转换的中间步骤，在 Gem5-NVP 看来，外界环境中的某种能量强度经过一系列的函数变化被成为了电容中电能的一部分，因此 Gem5-NVP 的能量收集模块在一次能量收集时得到的信息有两个：外界能量强度，电容此时状态（电压），能量收集模块根据某一些规则将计算出此次能量收集后电容的状态（电压），并且将这个数值返回给能量管理模块。

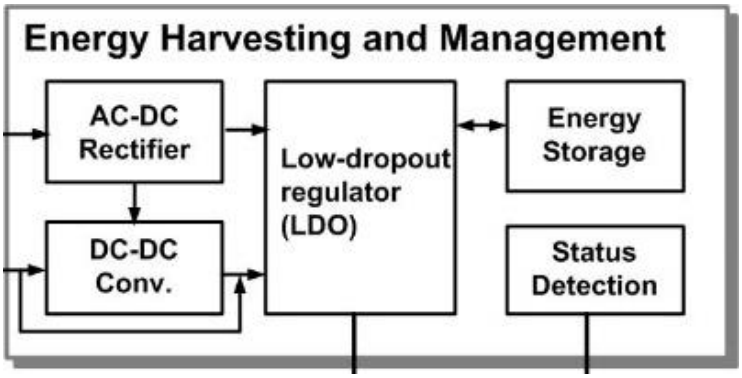


图 4.3 常见能量采集模块[4]

Gem5-NVP 的一个原则是，如果非易失体系架构中的某一些方面值得被研究或是研究的热点，则将这一个部分在软件仿真器中编写为可插拔的模块，以方便用户进行功能的改变和仿真。显然，能量收集单元是一个当前的研究热点，因此能量收集模块是一个可插拔的模块。Gem5-NVP 为能量收集模块提供了基类 BaseHarvest，这个类的类成员函数 double energy_harvest(double energy_harvested, double energy_remained) 需要在子类中被重写，也就是说，用户如果想自定义能量收集的方式，就需要从 BaseHarvest 中派生一个简单的类，并重写计算函数 energy_harvest，最后将一个这个类的对象“插入”能量管理模块。

如果用户不进行任何配置，Gem5-NVP 提供的能量收集模块是将外界能量强度以线性的方式累加到电容中的简单能量收集模块 SimpleEnergyHarvest。

4.3 系统状态机模块

从图 4.2 中可以看出，每当系统电容中储存的能量发生变化时，能量管理模块将通知一个系统状态机。这个系统状态机的主要功能是，随时监测系统此时的能量状态（电容的电压），并根据这个电压的变化来判断系统是否需要发生状态改变（比如是否开始备份、是否休眠、是否开机等），而一旦需要进行状态改变时，状态机将在能量接口上向被能量管理模块管理的其他模块发送状态改变的消息。这样一来，可以给系统状态机抽象出两个任务：任务一是定义系统的状态和状态转换，并在运行时实时监控系统能量并维护这个状态机的行为；任务二是为系统的状态改变定义响应的能量信息，也就是定义上文所述的能量信息 EnergyMsg 除了通知能量消耗（type 为 0）之外还能够通知哪些信息（type 为其他值的意义），在运行时，系统状态机会将系统的状态改变时对应的状态信息通过能量管理模块发送给系统的其他模块。

由于系统状态机也是非易失处理器的一个研究重点，系统状态机模块和能量收集模块相似，都是可插拔的，也就是说，Gem5-NVP 为其提供了拥有通用接口的基类 BaseEnergySM，用户需要根据自身需求来进行继承，并实现某些成员函数的行为。

基类提供了接口 void update(douhble _energy)，这个成员函数是一个虚函数，需要在子类中被重新定义。接口通知了系统能量的变化，参数_energy 是变化后的电容电压，根据发送的这个能量值和此前发送的能量值，用户可以在系统状态机类内部维护系统的状态，进行必要的状态更新，当系统状态机需要通知系统进行状态改变时，需要将对应信息放入一个 EnergyMsg 中，并通过基类提供的函数接口 void broadcastMsg(const EnergyMsg &msg)通知能量管理模块发送这一条信息。需要值得注意的是，系统的状态种类、系统的状态迁移都是由这个系统状态机模块进行定义的，因此对应状态改变的信息也需要这个模块进行定义（比如说 EnergyMsg::type 为 1 代表什么，为 2 又代表什么），当用户完成系统状态机设计后，还需要编写被能量管理模块所管理模块的 handleMsg 函数，根据定义的能量消息类型在接收到消息时进行合适的操作。

当用户未自定义系统状态机模块时，默认的系统状态机是 SimpleEnergySM，这个状态机只有开机、关机两个状态，当系统能量由负到正时控制系统开机，当

系统能量由正到负时控制系统关机。此外，Gem5-NVP 还提供了拥有不同开关机阈值的简易状态机，称为 TwoThresSM，可以在 Python 端由用户直接配置使用。

4.4 能量管理模块类成员介绍

这一部分简单介绍能量管理模块（EnergyMgmt）拥有的成员变量和成员函数。

表 4.1 能量管理模块成员变量

成员变量	简介
time_unit	能量配置文件和能量采集的时间单元
energy_remained	系统电容的电压值
energy_harvest_data	从能量配置文件中读取的所有环境能量强度条目
state_machine	指向系统状态机的指针
harvest_module	指向能量收集模块的指针
event_harvest	触发能量收集的周期性事件
event_msg	向被管理模块发送状态变化信息的事件
msg_togo	系统状态机告知发送的信息
path_energy_profile	能量配置文件的路径

表 4.2 能量管理模块成员函数

成员函数	简介
consumeEnergy	能量消耗的处理器
broadcastMsg	发送系统状态改变消息的函数
broadcastAsEvent	以事件形式发送系统状态改变的函数（不直接调用 broadcastMsg 是为了避免出现程序出现同时性缺陷）
handleMsg	接收并处理能量信息（从基类继承）
energyHarvest	从能量配置文件中读取条目并触发能量采集事件
readEnergyProfile	初始化是读取能量配置文件的函数

4.5 Python 端配置

能量管理模块 EnergyMgmt 在 Python 端也存在着对应同名实现，用户需要创建系统的能量管理模块的对象：

system.energy_mgmt = EnergyMgmt()

接下来可以通过四个参数对能量管理模块进行配置：

表 4.3 能量模块 Python 端参数

变量	说明
path_energy_profile	能量配置文件（energy profile）的路径
energy_time_unit	能量配置文件和能量采集的时间单元，如“10us”
state_machine	使用的系统状态机，默认为 SimpleEnergySM
harvest_module	使用的能量采集模块，默认为 SimpleHarvest

如：

system.energy_mgmt.path_energy_profile = ‘energy_prof’

system.energy_mgmt.energy_time_unit = ‘10us’

system.energy_mgmt.state_machine = SimpleEnergySM()

system.energy_mgmt.harvest_module = SimpleHarvest()

最后，我们需要将所有被控制模块的从能量接口与能量管理模块的主能量接口连接，比如，在某个系统中如果需要管理 cpu 的上电、掉电行为，需要配置为：

system.cpu.s_energy_port = system.energy_mgmt.m_energy_port

第 5 章 外设行为建模

这一部分将会介绍在 Gem5-NVP 中引入的外设模块（称为虚拟外设，Virtual Devices），虚拟外设能够帮助我们进行有关非易失处理器与外部硬件进行交互的过程。事实上，对于外设的仿真可能是一个软件非易失仿真器最重要的功能，因为绝大多数非易失处理器的工作方式都是与外设交互，由于非易失处理器功率低，运算速度慢，因此非易失处理器所在的节点一般不处理大规模的运算负载，而是作为终端获取环境信息并进行简单的预处理，常用的非易失节点，比如心脏起搏器监测节点、山体应力监测节点、水流监测节点，都遵从这个工作模式，因此，非易失处理器的行为是和外设紧密相关的，一个可用的非易失处理器软件仿真平台必须拥有对外设进行建模的功能。

本部分会先对虚拟外设进行简介，接下来介绍虚拟外设的概念与工作模式，最后介绍虚拟外设的地址解析与使用方式。

5.1 虚拟外设简介

虚拟外设也是外设，正如 Gem5 的文件夹“src/dev”中的外设一样，它通过总线与 CPU 联系在一起，且和 CPU、内存一样拥有同样的内存访问协议，也和传统的 Gem5 外设一样拥有一定的物理地址空间。然而，虚拟外设与 Gem5 传统的外设有着一些区别，正是由于这些区别的存在，引入虚拟外设才是必要的。下面首先将介绍 Gem5 传统外设在进行非易失处理器仿真时面临的问题，接下来介绍虚拟外设与传统外设的不同（实际上是为了解决问题）。

Gem5 传统外设涵盖了计算机外设的方方面面，从 PCI 设备到鼠标、键盘应有尽有，然而 Gem5 传统外设虽然全面并且精细，却并不易于使用。在 Gem5 中，只有完整系统仿真模式（Full System Mode，FS Mode）才能够调用外设，这是因为一般而言外设的工作流程一般涉及到操作系统，比如说：外设会触发中断，这时就需要操作系统有线程切换管理功能；外设需要用对应的物理地址调用，这时需要操作系统在虚拟地址空间上开辟一段专用的空间供程序来调用外设。但是，运行 FS 模式需要用户寻找兼容特定指令集的系统镜像，同时正确配置全部硬件，

再将需要运行的程序再这个系统镜像中编译运行，这无疑是十分复杂的，如果用户仅仅是为了仿真某些特定任务在非易失系统下的工作效率，那么以上复杂的配置使不必要的。

此外，Gem5 传统外设并没有对发生断电、或者电力恢复时的特性进行建模，由于硬件种类太多，对非易失、易失特性建模的难度十分巨大，这为非易失仿真器中使用这些传统外设带来了额外的难度。

基于以上的原因，我们需要引入能够在系统调用仿真模式（Syscall Emulation Mode， SE Mode）下能够方便进行调用，且能够对大部分外设的行为进行通用有效建模的外设，此外设还需要能够对能量有关的事件（上电、掉电）进行建模，供用户进行非易失处理器仿真用。这个外设就是所谓的“虚拟外设”（Virtual Devices）。

首先，虚拟外设并不像传统外设一样进行实际有意义的操作，传统外设每一个都有专属的工作，比如说网卡就会向外部发送网络包，磁盘外设就仿真磁盘的寻址、写入、读出功能，虚拟外设并不进行上述工作，它只是装作繁忙，这是因为用户在仿真非易失系统中外设时往往只在意外设的时间片调度，而不在意外设真的做了什么。

其次，虚拟外设能够工作在 SE 模式下，这是因为 Gem5-NVP 引入了虚拟地址解析功能，使得用户在 SE 模式下没有操作系统时同样可以将某些虚拟地址映射到外设的物理地址上，并进行调用，且虚拟外设在触发中断时可以不经过操作系统（涉及到中断向量等），直接在 CPU 上触发跳转。

最后，虚拟外设对能量相关的行为进行了通用的建模，能够有效仿真外设在发生断电、上电时的行为。

5.2 虚拟外设工作流程

虚拟外设的设计原则是使用非常少的参数就能对外设的整个工作流程进行建模，用户往往并不关心外设真的做了什么，而只关心外设工作时需要的时间和能量，因此，有关虚拟外设的参数大多都与时间和能量相关。

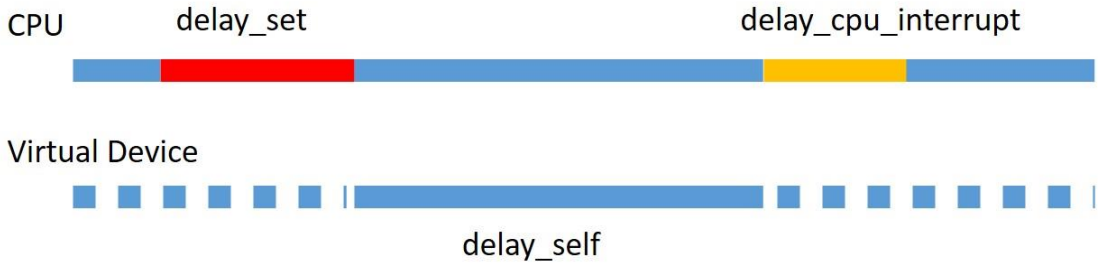


图 5.1 外设在常规情况下工作流程

图 5.1 是一个虚拟外设在常规情况下（能量充足，不发生断电重启）时的工作流程。首先，CPU 会操作和虚拟外设相关的虚拟内存地址（如何映射后文会提到），写入这些内存的作用是模拟 CPU 配置外设的过程，这段配置时间在图中为“delay_set”。当虚拟外设接收到请求时，如果其本身并没有在工作，则进入工作状态，工作时间称为“delay_self”，当工作完成后，虚拟外设会向 CPU 触发一个中断，在这个中断中 CPU 完成接下来的收尾工作，中断消耗的时间称为“delay_cpu_interrupt”。

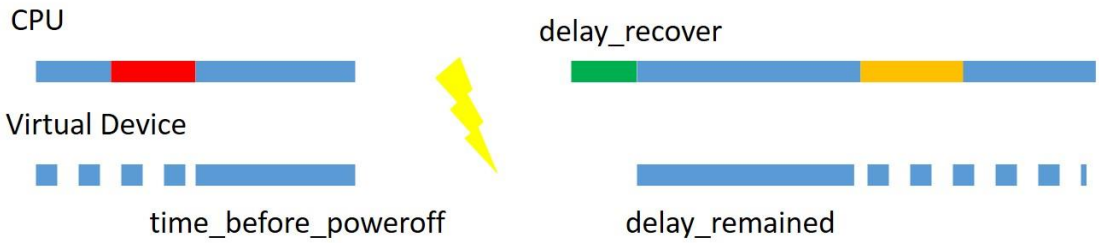


图 5.2 外设在断电重启时的工作流程

图 5.2 描述了当出现断电重启事件时虚拟外设的工作流程。当发生断电时 cpu 和虚拟外设都停止工作，当重启后，CPU 可能会需要一些时间来重新初始化这个外设，这段时间称为“delay_recover”，这段初始化时间结束后，外设继续工作，工作时间为“delay_remained”，值得注意的是，当外设属于工作可以被打断的类型时，图 5.2 中 delay_remained=delay_self-time_before_poweroff，也就是说，delay_remained 不是被提前确定的，这段时间应当是完成剩余工作的时间。

综合以上的描述，一个虚拟外设的行为主要由下表（表 5.1）中参数确定。

表 5.1 外设建模参数

参数	描述
delay_self	虚拟外设完成工作需要的时间
delay_set	CPU 最初调用外设初始化所需时间
delay_recover	当断电重启后 CPU 重新配置外设所需时间
delay_remained	外设从断电重启后继续完成工作所需时间
delay_cpu_intrerrupt	虚拟外设触发的中断所需时间
is_interruptable	外设是否是可以被打断的，如果是，则 delay_remained 会在运行时被确定

5.3 外设地址解析

一般来说，CPU 调用外设的方式是将请求通过写入外设对应的地址的，正常来说，外设对应的地址是由操作系统（Operating System）来确定的，但是虚拟外设工作在 SE 模式，不存在操作系统也不存在对外设的调用，为了使 CPU 能够调用虚拟外设，我们需要手动为虚拟外设进行地址解析。在运行有关虚拟外设的仿真之前，用户需要定义两个地址空间，第一个是虚拟外设的物理内存地址空间，这段空间不应当和内存的地址空间冲突，第二个空间就是虚拟外设在虚拟地址空间中的内存范围，这段虚拟的内存空间的大小需要和外设的物理内存空间拥有相同的大小。

虚拟外设的地址空间中的第一个字节为控制字节，其结构如下：

```
|-high----------------------low-|
|-4 bits-|correct|finish|work|set|
```

这个字节的最低位被用来调用虚拟外设，程序可以将这一位写入 1，这意味着 CPU 此时要调用虚拟外设，这个字节的其他位为只读，代表着虚拟外设的工作状态。

5.4 Python 端配置

虚拟外设同样有前端（Python）和后端（C++）两部分的代码，Python 端中虚拟外设的类名为 VirtualDevice，其成员变量如下包含了前文描述的虚拟外设的所有时间参数。

为了仿真虚拟外设的行为，除了创建虚拟外设对象之外，用户还需要在系统中开辟虚拟外设对应的物理内存空间与虚拟内存空间，并将二者对应，系统对象 System 中被引入了 vdev_ranges（虚拟外设物理地址空间）与 vaddr_vdev_ranges（虚拟外设虚拟地址空间）两个成员变量，供用户定义外设的内存地址空间，在确定好这两个变量后，用户还需要将物理地址空间中的对应条目分配给特定的外设对象的 range 成员变量。

以下为一个配置虚拟外设的例子：

```
system.vdev = VirtualDevice()
system.vdev.cpu = system.cpu
system.vdev.range = system.vdev_ranges[0]
system.vdev.delay_self = '10ms'
system.vdev.delay_cpu_interrupt = '100us'
system.vdev.delay_set = '200us'
system.vdev.delay_recover = '100us'
system.vdev.is_interruptable = 0
system.vdev.port = system.membus.master
system.vdev.s_energy_port = system.energy_mgmt.m_energy_port
```

5.5 程序端配置

用户编写的可执行程序如果想调用虚拟外设，可以直接访问外设的虚拟地址范围，使用的函数为内存地址映射的 mmap 函数。

以下为访问虚拟地址处于 1000MB 处的虚拟外设的例子：

```
size_t s = sizeof(uint8_t);
void * addr = (void*) 0x3e800000;
uint8_t *p = (uint8_t*) mmap(addr, s, PROT_READ|PROT_WRITE,
                             MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
*p = 0x01;
```

第 6 章 测试与仿真

本章节将对 Gem5-NVP 进行一些测试，有些测试有助于验证 Gem5-NVP 与现实系统仿真出信息的一致性，并且能够为现实系统的研发提供一些线索。

6.1 DFS 系统仿真

6.1.1 简介

DFS（Dynamic Frequency Selection）为动态频率选择系统的简称，这种处理器会随着工作时的负载和外界能量变化而改变自身的工作频率，在非易失处理器领域，DFS 处理器的意义在于，当外界能量强度不够高（但也不至于过低导致大部分时间都让非易失处理器在休眠），不足以支持固定频率的处理器长时间持续运行时，处理器会面对频繁的掉电、上电，断电重启的恢复时间会导致系统运行的效率过低，如果系统能够在能量较少时使用降低频率的方式来降低功耗，则有可能避免频繁的断电重启，从而增加完成任务的效率。

基于以上观点，下面将对动态频率系统与传统非易失系统进行测试，测试将使用一种典型的能量配置文件（energy profile）——平稳带有白噪声的收集配置文件。测试中将不断修改输入的能量配置文件的强度（整体与一定的系数相乘），并观察两种系统的运行八皇后 benchmark 的时间。

6.1.2 仿真参数设置

表 6.1 中给出了此次仿真的参数：

表 6.1 DFS 探究时仿真参数

参数	取值
CPU 类型	单周期 atomic
内存	512MB DDR3 1600
DFS 系统高点频率	1MHz

DFS 系统低点频率	500kHz
传统系统频率	1MHz
开机能量阈值	20000 单位
关机能量阈值	10000 单位
DFS 系统频率转换频率	20000 单位
能量配置文件时间单元	10us
CPU 高频率下功率	1.5 单位/us
CPU 低频率下功率	0.75 单位/us
CPU 断电恢复所需周期	1000

6.1.3 仿真结果

我们在一张图表中给出绘制以能量配置文件平均功率（单位/us）为横轴，以运行所需时间为纵轴的结果图，如图 6.1

从中可以看出两个趋势，第一个显而易见的趋势是二者随着外部能量的提高运行时间缩短的速度都越来越慢，而第二个趋势是 DFS 系统相对传统系统的提高比例随着能量越来越充足会先上升再下降，如图 6.2。

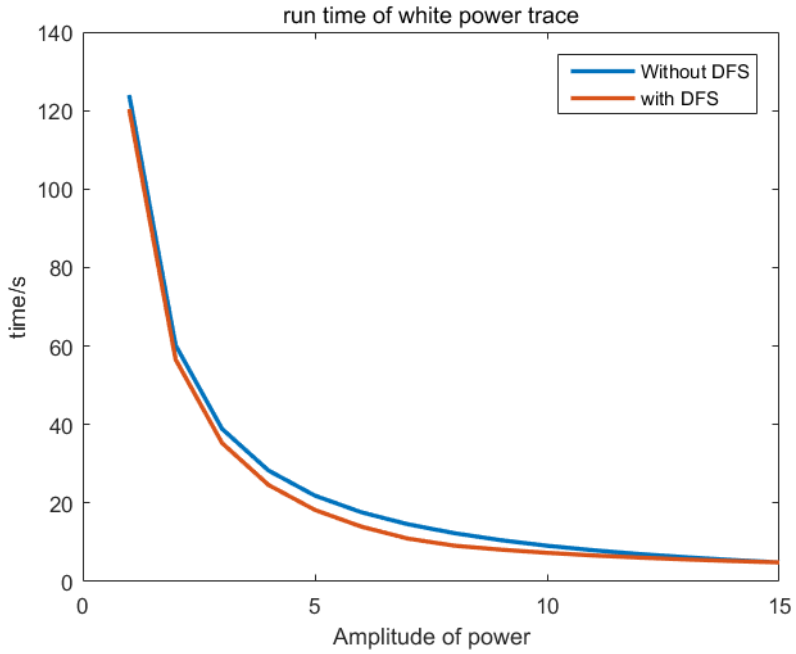


图 6.1 运行时间随外界能量强度变化

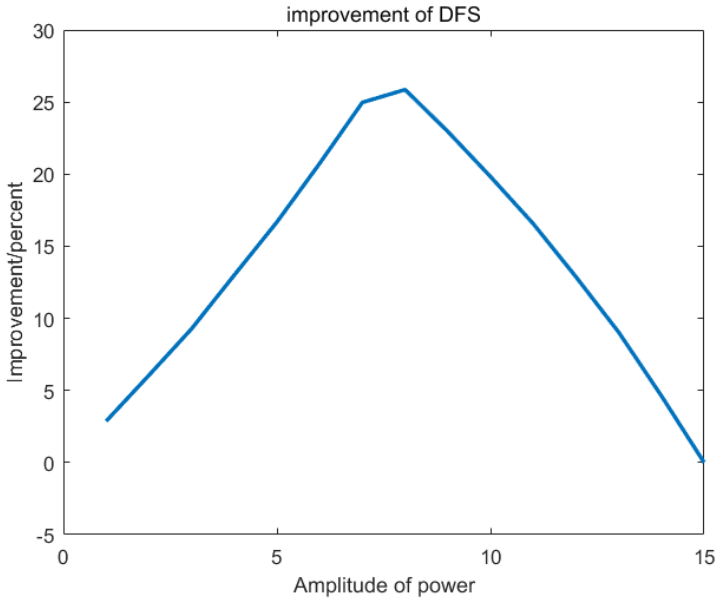


图 6.2 DFS 性能提升随外界能量强度变化（百分比）

6.1.4 分析

第一个趋势：随着外部能量的提高运行时间缩短的速度都越来越慢。这是因为能量提高到一定程度后系统的断电重启的次数大大降低，当断电重启接近消失时继续提升外部能量对运行时间影响就不大了。

第二个趋势：DFS 系统相对传统系统的提高比例随着能量越来越充足会先上升再下降。这个原因是当外界能量非常低时，无论系统是否有 DFS 常态都是休眠，当积累一定能量后才会短促运行，这时是否有 DFS 功能对程序运行效率影响不大，影响运行时间的主要是能量强度大小。当外界能量升高到一定程度时，有 DFS 的系统会由于能够在能量不足时降低频率，可以大大减低断电重启次数，而无 DFS 系统要频繁面临断电重启带来的恢复时间，这个时候 DFS 对系统性能的提高是最有效的。当外界能量继续升高，使得能量能够使得系统维持在高频点运行时，是否有 DFS 功能对系统而言就无关紧要了，因此 DFS 对系统性能的提高此时慢慢降低。

6.2 非易失外设仿真

6.2.1 简介

外设在非易失系统中占有重要的地位，对于外设的优化有几种方向，其中一种方向就是使外设变为非易失的，也就是说当发生断电重启的时候外设不需要CPU 进行任何干涉，自身即可保存自身的状态，并且完成断电自启。下面将会使用不同强度的能量配置文件（不同强度的方波）测试非易失外设对系统整体究竟有多少性能提升。外设的非易失性由当断电恢复后 CPU 需要多长时间来重新配置外设决定。使用的标准能量配置文件的平均功率约为 0.5 单位/us，在使用过程中会将能量配置文件与一定系数相乘获得不同外界功率强度。

本次仿真使用的 testbench 为连续请求外设工作 100 次的中断触发程序。

6.2.2 仿真参数

表 6.2 为本次仿真使用的参数。

表 6.2 非易失外设仿真参数

参数	取值
CPU 类型	Atomic
CPU 频率	1MHz
开机能量值	20000 单位
关机能量值	10000 单位
内存	512MB DDR3 1600
外设初始化时间（delay_set）	200us
外设工作时间（delay_self）	10ms
外设中断时间（delay_cpu_interrupt）	100us
系统功率	1.5 单位/us
能量配置文件时间单位	10us

6.2.3 外设恢复时间对性能的影响

分别将能量配置文件与一定系数相乘获得不同的外界能量强度，接下来仿真外设从断电重启中恢复所需的 CPU 配置时间（delay_recover）变化对程序运行效率的影响。图 6.3 为不同外界能量强度下程序运行时间随着 delay_recover 的变化。

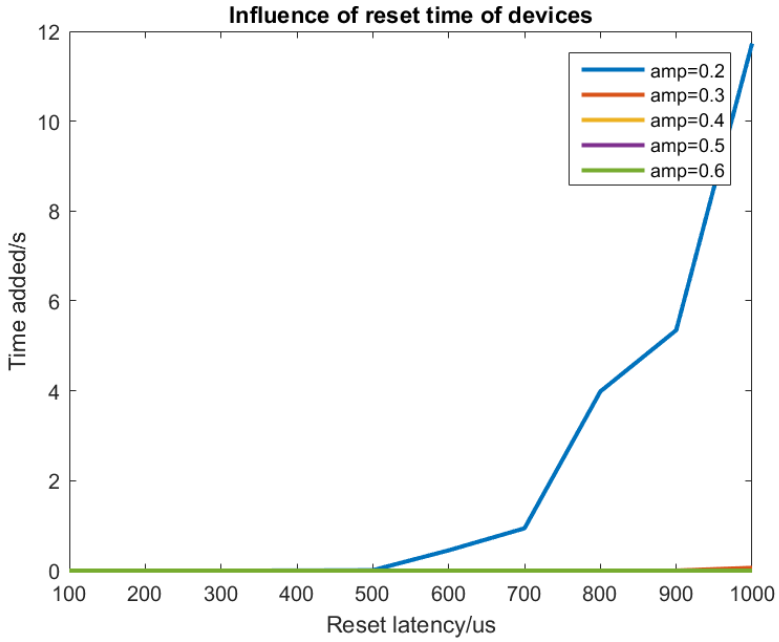


图 6.3 外设重新配置时间的影响

6.2.4 分析

从图 6.3 中可以看出，当能量充足时外设重新配置时间对程序运行效率的影响较低，而当能量不足导致程序运行中有大量断电重启时，外设恢复所需的 CPU 时间影响就越大，这是因为当 CPU 在断电后重新配置外设的时间变长后，给整个系统留下了更大的断电窗口（从开始重新配置到外设工作完毕），从而导致更多的外设重新配置时间，这是一个正反馈调节过程。

这一结论的启示是，在能量缺乏的环境下工作的系统对外设恢复占用的 CPU 时间更为敏感，这样的系统更加需要进行非易失外设的研究（非易失外设即断电重启后不需要占用 CPU 进行重新配置的外设）。

第 7 章 结论

本次毕业设计基于将 Gem5 仿真器扩展成为了 Gem5-NVP，一个支持能量行为与非易失系统的软件仿真平台。这个软件仿真平台拥有如下特点：

1. Gem5-NVP 为每一个模块都扩展了通用的能量相关接口，为模块间的能量通信定义了一套协议，这使得用户能够简单的仿真硬件的能量相关行为，比如消耗能量、开机、关机、休眠等等；
2. Gem5-NVP 为模块间的能量传递协议编写了方便易用的前端，使得用户能够使用前端的 Python 脚本轻松定义非易失系统中各个模块的连接关系和能量传递方向；
3. Gem5-NVP 对非易失系统的能量管理模块进行了建模与抽象，在 Gem5 中引入了能量管理模块，并编写了一些重要硬件（比如 CPU）在断电、重启时的行为；
4. Gem5-NVP 拥有非常出色的可扩展性，一些和非易失行为有重大关联的部分都支持用户使用派生的方式进行方便的行为重定义（比如系统状态机），由于 Gem5-NVP 基于 Gem5 平台，用户能够极为方便地在系统中加入自定义的新模块并完成需要的仿真。

以上特点使得 Gem5-NVP 成为了目前世界上唯一的拥有良好的易用性和扩展性的非易失软件仿真平台。事实上，“使用 Gem5-NVP 测试动态频率选择系统特点”目前成为了清华大学电子工程系本科生“现代计算机体系架构”课程的大作业之一，这证明了 Gem5-NVP 的较低的入门门槛和良好的可用性。此外，Gem5-NVP 的文档已经上线，即将被国内外多个研究小组使用，在未来可能会成为拥有完整的用户、文档、样例的生态体系。

插图索引

图 1.1　非易失处理器结构……3
图 2.1　软件仿真系统结构……..12
图 2.2　Gem5 类继承关系……14
图 2.3　能量模块 EnergyObject　……...15
图 4.1　能量管理模块……..22
图 4.2　能量管理模块工作流程……..23
图 4.3　常见能量采集模块……..24
图 5.1　外设在常规情况下工作流程……..30
图 5.2　外设在断电重启时的工作流程……..30
图 6.1　运行时间随外界能量强度变化……..34
图 6.2　DFS 性能提升随外界能量强度变化……...……...35
图 6.3　外设重新配置时间的影响……..37

表格索引

表 3.1　能量接口共有成员变量……17
表 3.2　能量接口共有成员函数……17
表 3.3　从能量接口成员变量……17
表 3.4　从能量接口成员函数……18
表 3.5　主能量接口成员变量……18
表 3.6　主能量接口成员函数……18
表 3.7　能量模块成员变量……19
表 3.8　能量模块成员函数……19
表 4.1　能量管理模块成员变量……26
表 4.2　能量管理模块成员函数……26
表 4.3　能量模块 Python 端参数……27
表 5.1　外设建模参数……31
表 6.1　DFS 探究时仿真参数……33
表 6.2　非易失外设仿真参数……36

参考文献

[1] Atzori L, Iera A, Morabito G. The Internet of Things: A survey[J]. Computer Networks, 2010, 54(15):2787-2805.

[2] Perton M, Audoin B, Pan Y D, et al. Energy harvesting vibration sources for microsystems applications[J]. Measurement Science & Technology, 2006, 17(12):R175-R195.

[3] Parks A N, Sample A P, Zhao Y, et al. A wireless sensing platform utilizing ambient RF energy[J]. Journal of Pharmacology & Experimental Therapeutics, 2013, 294(2):331-333.

[4] Ma K, Zheng Y, Li S, et al. Architecture exploration for ambient energy harvesting nonvolatile processors[C]// IEEE, International Symposium on High PERFORMANCE Computer Architecture. IEEE, 2015:1-1.

[5] Akinaga H, Shima H. Resistive Random Access Memory (ReRAM) Based on Metal Oxides[J]. Proceedings of the IEEE, 2010, 98(12):2237-2251.

[6] Sakimura N, Sugibayashi T, Nebashi R, et al. Nonvolatile Magnetic Flip-Flop for Standby-Power-Free SoCs[J]. IEEE Journal of Solid-State Circuits, 2008, 44(8):2244-2250.

[7] Ransford B, Sorber J, Fu K. Mementos: system support for long-running computation on RFID-scale devices[C]// Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2011:159-170.

[8] Balsamo D, Weddell A S, Merrett G V, et al. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems[J]. IEEE Embedded Systems Letters, 2015, 7(1):15-18.

[9] Gu Y, Liu Y, Wang Y, et al. NVPsim: A simulator for architecture explorations of nonvolatile processors[C]// Asia and South Pacific Design Automation Conference. IEEE, 2016:147-152.

[10] Li H, Liu Y, Zhao Q, et al. An energy efficient backup scheme with low inrush current for nonvolatile SRAM in energy harvesting sensor nodes[C]// Design, Automation & Test in Europe Conference & Exhibition. EDA Consortium, 2015:7-12.

[11] Li H, Liu Y, Fu C, et al. Performance-aware task scheduling for energy harvesting nonvolatile processors considering power switching overhead[J]. 2016:1-6.

[12] Zhang D, Liu Y, Sheng X, et al. Deadline-aware task scheduling for solar-powered nonvolatile sensor nodes with global energy migration[C]// Design Automation Conference. IEEE, 2015:1-6.

[13] Graphics, Mentor. ModelSim. //2007

[14] Binkert N, Beckmann B, Black G, et al. The gem5 simulator[J]. Acm Sigarch Computer Architecture News, 2011, 39(2):1-7.

[15] Wang Y, Liu Y, Li S, et al. A 3us wake-up time nonvolatile processor based on ferroelectric flip-

flops[C]// Esscirc. 2012:149-152.

致谢

本次毕业设计自 2016 年秋季学期期中到 2017 年春季学期期末，持续了约半年的时间，在这段时间内，很多老师和同学为我的毕业设计提供了帮助。

本次毕业设计中，我非常感谢我的指导教师孙忆南老师和电路所的刘勇攀老师。孙忆南老师无论在毕业设计的开题阶段、中期答辩还是收尾阶段均提供了巨大的帮助，与孙忆南老师的讨论我获得了确定工作方向的线索，完善了每一次答辩与报告。在毕业设计的过程中，我很荣幸进入了刘勇攀老师的研究小组，研究小组的学习会、交流会让我能够很快入门非易失处理器的研究，了解了当前最新的发展现状，此外，刘勇攀老师在每周均认真审视我的工作情况并提出切实有效的意见。

感谢电路所的武通达学长为我耐心解答有关非易失处理器、非易失系统的问题，并与我讨论非易失软件仿真器所需要的功能，这给我的毕业设计提供了很大的帮助。感谢李金阳学长为我提供了非易失芯片 THU1020n 的 testbench 与测试数据，以及感谢岳金山学长帮助我的研究内容成为了“现代计算机体系架构”课程的大作业之一。

最后感谢我的父母在我本科的最后阶段对我一如既往的支持。

声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果，尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作作出贡献的其他个人和集体，均已在文中以明确方式标明。

签名：张乐凡　　日期：06/12/2017

附录 A 外文资料的调研书面翻译

NVPsim：应用于非易失处理器架构探索的仿真器

摘要

非易失处理器（NVP）应用了非易失内存技术，在发生断电时保存运行时的信息。在能量采集系统中，这个功能使得 NVP 能够在间断的电源供应下进行连续的工作进展。这篇论文基于 gem5 建立了一个 NVP 仿真器，NVPsim，这个仿真器被已经流片的原型芯片的测试结果所验证，拥有良好的误差。刚进一步，为了阐述这个仿真器进行架构探索的能力，我们探究了选取不同非易失内存作为片上缓存、不同备份策略和不同能量缓冲大小的非易失处理器设计的性能和能量消耗。

介绍

能量采集技术被广泛应用于无电池设备的供电。然而，能量源往往有着先天的不确定性，这导致了采集到的功率不可避免地在有些时候低于设备的需求，造成间歇的电源中断。频繁的电源中断造成了微处理器中寄存器、易失的缓存中运行状态的丢失，导致了系统性能的下降。为了解决这一问题，非易失处理器（NVP）在最近被广泛认为是一种需要从外界进行能量采集的节点的解决方案。

非易失处理器使用片上的非易失储存器来在发生断电时保存运行线程的状态。当供电恢复，线程状态从非易失储存器中被恢复。Wang 等人将易失的触发器（Flip Flop）换成了铁电的触发器，从而实现了非易失处理器。这个处理器相比“MSP430FR 系列”微处理器休眠和唤醒时间分别小 30 倍和 103 倍。

非易失处理器体系结构的探索在之前的工作中被实现，Ma 等人探索了多种非易失处理器的设计，这些设计使用了不同观点微结构和不同的能量源来最大化非易失处理器的运行工作进展。然而，他们主要关注点在于备份非易失处理器的寄存器的策略，而忽视了片上缓存的设计。尽管 NVP 指令和数据的缓存可以使用非易失储存器，高的写入所需能量和延时会因为非易失储存器的特性而被引入。非易失 SRAM（nvSRAM）被提出以替换传统缓存中的 SRAM，以实现高性能和低能量消耗。Tsai 等人分析了比特级的写入 nvSRAM 写入数据量降低技术，Li 等

人提出了块级的能量利用率高的 nvSRAM 备份技术。然而，这些备份技术的有效性缺少在不同架构下的系统探究。事实上，随着核心数和内存数的提升差距变大，NVP 处理器的缓存设计变得越来越重要。它为非易失处理器的设计添加了新的维度。

NVP 处理器的设计探索需要一个合适的仿真工具，Ma 等人验证非易失处理器设计的方式是使用 Verilog，此仿真方式仿真整个体系架构速度非常慢，Dong 等人设计了 NVSim，以仿真不同非易失存储器的设计技术，但是没有考虑整个体系架构。Binkert 等人提出了 gem5，一个高配置自由度仿真框架，但是不对非易失处理器结构仿真提供支持。我们拓展了这个仿真框架，使之成为一个非易失处理器设计探索的合适仿真器。

这篇文章的目标有两层。首先，建立在现存 gem5 仿真框架下的一个体系结构级的非易失处理器仿真器被编写；其次，使用被提出的这个仿真器，我们进行了非易失处理器的设计探索。我们着重探索了非易失处理器片上缓存的设计，这在前人的工作中没有被提到。综上，此论文给出了如下贡献：

一个建立在 gem5 仿真框架的非易失处理器仿真器（NVPsim）被提出，经过验证，它是一个拥有高灵活性和高准确度的非易失处理器体系架构的探索工具。

我们使用 NVPsim 验证了使用不同非易失储存器来进行片上缓存、不同能量缓存的非易失处理器设计。

使用 NVPsim，我们验证了在不同缓存类型在不同策略下的备份运行状态的效率。

论文的其余部分如下：第二部分介绍了 NVPsim 基于的非易失处理器的模型买第三部分介绍 NVPsim 的设计，第四部分验证了 NVPsim 的准确性，第五部分进行了 NVP 的设计探索。

非易失处理器建模

我们参考一个已经流片的非易失处理器建立的非易失处理器模型，这个模型的概要在图 A.1 中给出，且由收集到的能量进行功能。在一个基于非易失处理器的系统，收集到的能量首先由一个 DC-DC 转换器进行蒸馏，接下来保存到一个电容C_{bulk}中，C_{bulk}作为能量的缓冲来保存收集到的能量并给非易失处理器功能，

在非易失处理器中，寄存器和片上缓存都使用的是非易失储存器。我们假定主内存使用阻性非易失储存器，因为主内存技术不在本文探讨范围内。

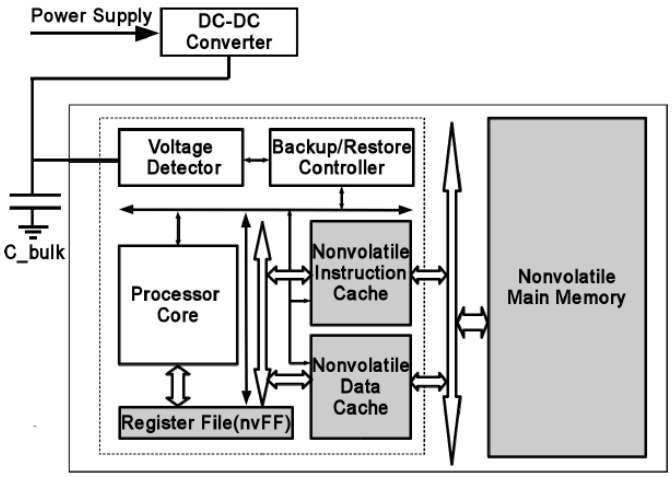


Fig. 1. Overview of the nonvolatile processor model.

图 A.1 非易失处理器概要

A. 电压检测器建模

当发生一次断电时，非易失处理器在关机前保存线程运行状态，当电力恢复时，非易失储存器在重新运行前恢复线程运行状态。电压检测模块会通过检测电容的电压来寻找可能的电能中断点和电能恢复点。图 A.2 阐述了电压检测器在时间域的行为。当电容的电压超过阈值电压$V_{restore}$时，电压检测器在时间t_{plh}后，t_1时发送一个“restore”信号给备份/恢复控制器。当这个恢复信号被备份/恢复控制器接收到时，非易失处理器将会在t_{rst}的时间内备份线程运行信息。非易失处理器在t_2重新开始运行。当电容电压低于阈值电压V_{backup}时，电压检测器会发送“backup”信号给备份/恢复控制器并在时间t_{phl}后在时间t_{bkp}完成备份。

注意到$V_{restore}$和V_{backup}都由一些电压窗设置技术，比如 MPPT 来决定，因此$V_{restore} > V_{bakcup}$是一定能得到满足的，V_{min}是 NVP 操作电压的最小值，在最坏的情况下，电容电压在恢复过程中可能低于V_{min}，这样的情况被认为是系统崩溃，系统将会回滚。

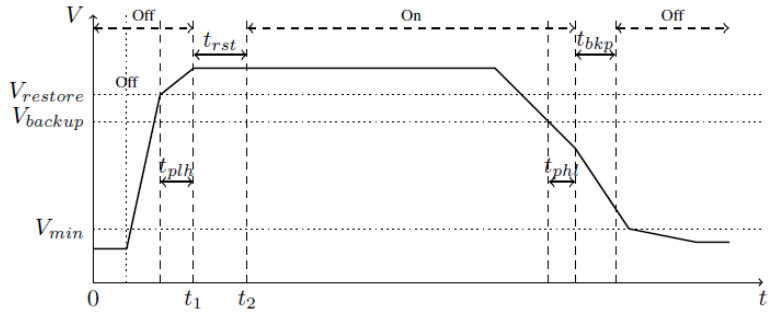


Fig. 2. Voltage detector modeling.

图 A.2 电压检测器建模

B. 备份/恢复控制器建模

备份/恢复控制器控制着寄存器和缓存的备份。在这个模型中，寄存器包含了非易失的触发器，寄存器中的内容如果不被备份到非易失触发器中将会丢失。当控制器接收到“backup”信号时，它将会向寄存器发送控制信号使所有的内容并行地被备份。

缓存的备份过程取决于非易失储存技术的选取，如果缓存使用的是需求是非易失储存器技术（比如 nvSRAM），备份/恢复模块需要使用在章节 3 中提到的策略控制缓存备份过程，否则缓存中的内容将不会被备份，因为在断电时数据的完整性可以被保持。当“restore”信号到达备份/恢复控制器时，控制器将会使用和备份时相同的方式来恢复寄存器和缓存中的数据。

C. 非易失处理器状态机建模

图 A.3 给出了非易失处理器的状态机模型。“s1”和“s2”代表了“running”和“run-backup”状态，当电容电压小于阈值电压V_{backup}时，状态机将会从“running”状态转移到“run-backup”状态。当“run-backup”状态消耗的时间

大于t_{phl}时，状态机将会从“run-backup”状态转移到“backup”状态（“s3”），当非易失储存器在t_{bkp}的时间里保存好线程运行数据后，状态机将从“backup”状态转移到“off”状态（“s4”）。状态机将维持在“off”状态，直到电容电压超过$V_{restore}$，这是状态机将会从转移到“off-restore”状态（“s5”），状态机在时间t_{plh}后将会从“off-restore”状态转移到“restore”状态（“s6”），当非易失处理器在时

间t_{rst}完成状态的恢复后，状态机会从“restore”状态转移到“rollback”状态（“s7”）。

“rollback”状态对系统的回转过程（跳转到上一条被“backup”信号打断未完成操作）进行了建模，当状态机在“rollback”状态时，它可能根据电容中的电压转移到不同的状态。如果非易失处理器还没有完成回转时电容电压就低于V_{backup}，则状态机进入“run-backup”状态，如果 NVP 能够完成回转并且没有发生电能中断，则状态机进入“running”状态。

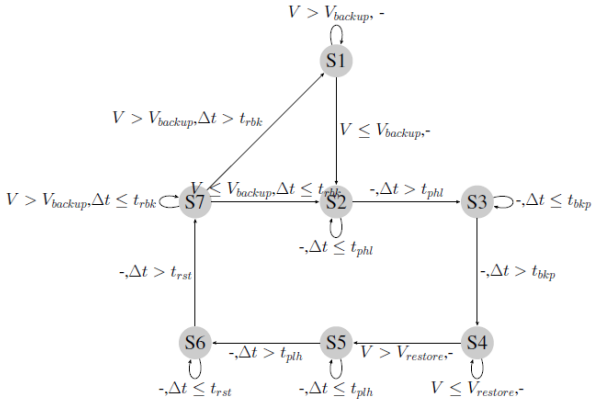


Fig. 3. The NVP state machine model. s1: running state; s2: run-backup state; s3: backup state; s4: off state; s5: off-restore state; s6: restore state; s7: rollback state. δt is the time duration since the system enters the corresponding state.

图 A.3 系统状态机建模

NVPsim 仿真器设计方法

这一章主要介绍 NVPsim 的设计方法，我们首先给出 NVPsim 的概述，接下来介绍电压检测模块、备份/恢复控制器和各个模块间的公布方式。

A. 仿真器概述

图 A.4 中给出了 NVPsim 的整体设计，第二章中介绍的非易失处理器模型指导了这一设计。NVPsim 是在 gem5 仿真框架的基础上建立的，TimingSimpleCPU 模型（一个平衡了仿真准确性和仿真速度的 CPU 模型）被用作 CPU 的模型，下述几个模块被添加：i）电压检测器，ii）备份/恢复控制器。一些 gem5 模块，比如事件管理模块和缓存模块被修改以便支持 NVPsim。

NVPsim 将能量供给文件、系统参数和仿真程序作为输入。能量供给文件决

定了输入的能量，NVP 的一些设置被系统参数设定（比如V_{backup}和电容大小）。NVPsim 使用 Mibench 基准程序系列、能量供给文件和系统设定进行仿真。NVPsim 的输出是不同硬件模块的能量消耗以及一个能够帮助我们评估性能和能量效率的数据统计文件。

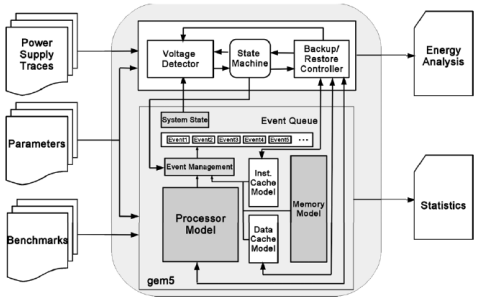


Fig. 4. Overview of NVPsim design.

图 A.4 NVPsim 设计概述

B. 电压检测器

给出当前电容的电压V_i后，电压检测器会使用差分算法计算下一个状态的电压V_{i+1}。电容中的电压由方程 A.1 决定。

$$E = \frac{1}{2}CV^2$$

对方程 A.1 的左右两端取时间的微分，得到：

$$\frac{dE}{dt} = CV\frac{dV}{dt}$$

使用简单的变换，方程 A.2 变为：

$$V_{i+1} - V_i = \frac{P_i \Delta t}{CV_i}$$

在方程 A.3 中，P_i是在第 i 步输入的净能量，而Δt是步间的时间间隔。P_i等于$P_{input_i} - P_{output_i}$，$P_{input_i}$和$P_{output_i}$是第 i 步的输入功率和输出功率。$P_{input_i}$是第 i 步的电能供给，$P_{output_i}$的动态计算是基于第 2 章中处理器的内部状态（如访问缓存）来计算的。比如，如果 NVP 在第 i 步的状态是“backup”，则P_{output_i}为备份/恢复模块计算出的备份功率。如果 NVP 在第 i 步的状态是“running”，则输出功率被当前处理器的内部状态（如缓存访问）来决定。

C. 备份/恢复控制器设计

备份/恢复控制器计算备份线程状态所需要的能量和时间，我们认为线程状态可以被比特级地并行备份和恢复，这被现存的硬件所支持。

假定某一级内存（比如缓存）中的 N 个比特被并行地备份，所需的总功率为$\mathrm{p}=\mathrm{P}_{\mathrm{bkp_1bit}}N$，所需时间是$\mathrm{t}=\mathrm{t}_{\mathrm{bkp_1bit}}$，$\mathrm{P}_{\mathrm{bkp_1bit}}$和$\mathrm{t}_{\mathrm{bkp_1bit}}$是备份一个比特所需的功率和时间，我们从 NVSim 的仿真结果与前人工作的文献中获得$\mathrm{P}_{\mathrm{bkp_1bit}}$和$\mathrm{t}_{\mathrm{bkp_1bit}}$。$\mathrm{P}_{\mathrm{max}}$是整个系统的功率上限，如果$\mathrm{P}_{\mathrm{bkp_1bit}}N$超过了$\mathrm{P}_{\mathrm{max}}$，这些字节将会分批次备份。令$\mathrm{k}=\frac{P_{bkp_1bit}N}{P_{max}}$，这些比特被分成了$\lceil k\rceil$个批次，这时$\mathrm{t}=\mathrm{t}_{\mathrm{bkp_1bit}}\lceil k\rceil$，$\mathrm{P}=\mathrm{P}_{\mathrm{bkp\_1bit}}N$，为了备份 N 个比特使用的功率和时间取决于P和$\mathrm{P}_{\mathrm{max}}$如果$\mathrm{P}<\mathrm{P}_{\mathrm{max}}$，则$[\mathrm{p},\mathrm{t}]=[\mathrm{P}_{bkp_1bit}N,t_{bkp_1bit}]$。

令$\kappa=\min\{\lceil k\rceil,\lfloor k\rfloor\}$，如果$\mathrm{P}>\mathrm{P}_{\mathrm{max}}$，计算结果将会由k决定，如果$\lfloor k\rfloor=\lceil k\rceil$，则$[\mathrm{p},\mathrm{t}]=[\mathrm{P}_{\mathrm{max}},t_{bkp_1bit}k]$，如果$\lfloor k\rfloor\neq\lceil k\rceil$，备份 N 比特所需的功率和时间将被分为 2 个阶段，如方程 A.4 所示。

$$\begin{cases}[p_1,t_1]=[P_{max},t_{bkp\_1bit}\kappa]\\ [p_2,t_2]=\left[P_{bkp_{1bit}}\left(N-\kappa\frac{P_{max}}{P_{bkp\_1bit}}\right),t_{bkp\_1bit}\right]\end{cases}$$

在第一个阶段，$\kappa\frac{P_{max}}{P_{bkp_1bit}}$个比特被分为κ个批次以$\mathrm{p}_1$的功率和$\mathrm{t}_1$的时间进行备份，第二个阶段，剩余的比特以$\mathrm{p}_2$的功率和$\mathrm{t}_2$的时间进行备份。恢复 N 个比特所需的功率和时间的计算方式同备份的计算方式相同，只需替换$[\mathrm{P}_{bkp_1bit}N,t_{bkp_1bit}]$为$[\mathrm{P}_{rst_1bit}N,t_{rst_1bit}]$。

如方程 A.4 所示，被备份或者恢复的比特数会影响备份或恢复的功率和时间，这些数据在涉及到寄存器等级时是固定的，如果缓存使用了在需求时进行备份的备份单元，这个数据将会动态变化，并且会被不同备份策略所影响。我们搭建了两个基本的备份缓存内容使用的备份策略：全备份和 LRU 部分备份。对于第一个策略，所有的缓存条目被备份，对于第二个策略，备份/恢复控制器将扫描所有的缓存条目并备份满足 LRU 条件的条目，剩余的缓存条目将不会被备份。

LRU 条件被如下定义：一个缓存条目B_i如果满足$\mathrm{T}_{\mathrm{backup}}-\mathrm{T}_\mathrm{i}\leq\mathrm{T}_{\mathrm{threshold}}$则符合这个条件，$\mathrm{T}_{\mathrm{backup}}$是备份/恢复控制器开始备份缓存的时间，$\mathrm{T}_\mathrm{i}$是缓存条目$\mathrm{B}_\mathrm{i}$上一次被访问的时间，$\mathrm{T}_{\mathrm{threshold}}$是系统参数中可以设定的阈值。由于运行指令数是和运行时间正相关的，我们将 LRU 备份策略中的时间都换成了指令树，LRU 部分备份策略的伪代码在算法 A.1 中给出。

Algorithm 1 LRU Partial Backup Strategy

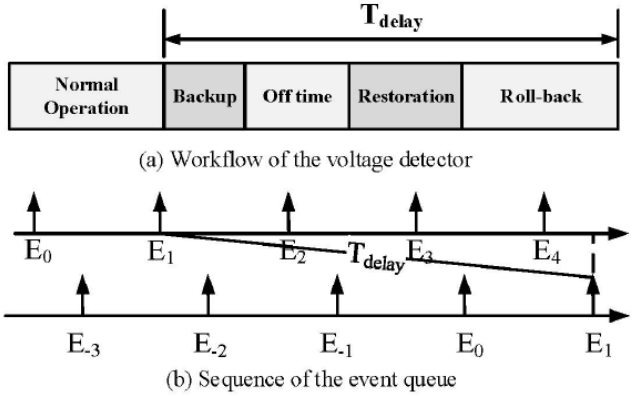
procedure LRUBACKUP(Cache,T_{backup},Threshold)
BackupNum $\leftarrow$ 0
for Block in Cache **do**
if Block.valid == True **then**
if T_{backup}- Block.T_i $\leq$ Threshold **then**
BackupNum$\leftarrow$ BackupNum + 1

算法 A.1 LRU 部分备份策略

D. 模块间的同步

Gem5 仿真框架是时间驱动的，事件之间的时间间隔是不确定的，然而，电压检测器检测断电、上电和更新 NVP 的状态的时间间隔是确定的，因此，我们需要进行事件队列和电压检测器之间的时间同步。我们修改 Gem5 仿真框架来实现这样的同步。

图 A.5 介绍了事件队列和电压检测器之间的同步。图 A.5(a)给出了电压检测器的工作流程，电压检测器走过的是“备份-关机-恢复”的过程，有一些操作不能被断电所打断，比如 CPU 和内存之间的相互作用。假定断电时有一个操作不能被打断，系统需要先在上电后重新运行它，因此回退时间呗包含在了T_{delay}中，整个个流程将事件队列中的事件都后推了T_{delay}，正如图 A.5(a)和 A.5(b)所示，从技术上来说，这个同步方式是通过用 POSIX 同步原语修改 Gem5 代码实现的。



(a) Workflow of the voltage detector

(b) Sequence of the event queue

Fig. 5. Synchronization between the event queue and the voltage detector.

图 A.5 事件队列和电压检测器间的同步

NVPsim 验证

我们通过对比仿真结果和真实的非易失系统来验证 NVPsim 的建模。NVPsim 的系统参数的设定值如表 A.1 所示。非易失系统的运行时间在不同 benchmark 和能量供给文件下进行了测试，仿真中使用的能量供给文件是频率为 16kHz 的方波，表 A.2 给出了仿真的结果，在表 A.2 中，D_C代表了能量供给文件的占空比。仿真结果与实验结果的平均误差为 3.07%，最大误差是 9.31%，因此，我们认为 NVPsim 进行的建模是足以有效率地表现实际系统的行为的。

表 A.1 实际 NVP 平台参数

PLATFORM SETUP OF AN ACUTAL NVP SYSTEM

Processor Freq.	25MHz	Mem Size	512KB	Mem Type	FeRAM
MCU Power	160μW	RegFile Size	128B	RegFile Type	NVFF

表 A.2 仿真结果和实际测试对比

COMPARISON OF SYSTEM PERFORMANCE ON SIMULATOR AND ACTUAL PLATFORM USING SQUARE WAVEFORM POWER SUPPLY WITH DIFFERENT DUTY CYCLES

	FFT /ms			Sort /ms			Sqrt /ms		
D_c	Sim.	Mea.	Err.	Sim.	Mea.	Err.	Sim.	Mea.	Err.
30%	54.0	49.4	9.31%	360	330	9.21%	33.2	30.7	8.14%
40%	37.8	35.9	5.44%	252	239	5.44%	23.3	22.3	4.48%
50%	29.0	27.3	6.23%	193	182	6.04%	17.8	16.9	5.33%
60%	23.1	22.6	2.21%	153	151	1.32%	14.2	14.0	1.43%
70%	19.4	19.3	0.52%	130	129	0.78%	12.0	12.0	0.00%
80%	17.0	16.5	0.30%	112	110	1.82%	10.4	10.2	1.96%
90%	15.0	14.6	0.27%	99.9	97.6	2.36%	9.2	9.1	1.09%
100%	12.4	12.4	0.00%	82.5	82.5	0.00%	7.65	7.65	0.00%

使用 NVPsim 进行设计探索

实验的配置在表 A.3 中被给出，我们对比了不同 NVP 设计，这些不同设计基于 NVPsim，包含了片上缓存设计和电容大小。表 A.4 列出了 7 个不同的片上缓存设计的区别，我们选取了阻性 RAM（RRAM）和 nvSRAM 作为缓存的非易失储存器技术。我们使用了 nvFF 作为寄存器非易失储存器。RRAM 单元的参数是从 NVSim 中获得的，NVFF 和 nvSRAM 的参数是从前人工作中获得的。

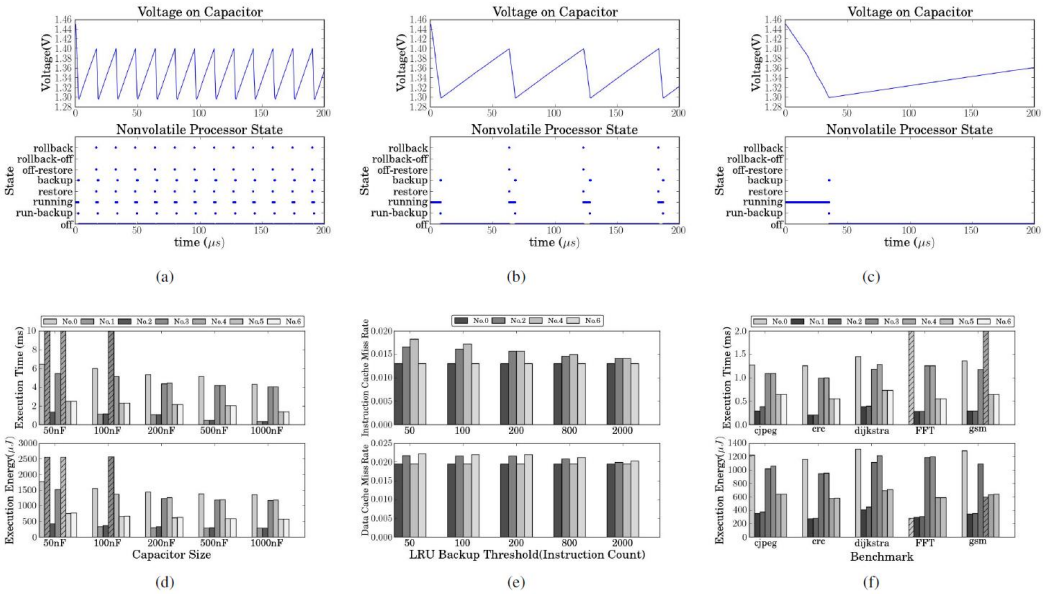


Fig. 6. Design space exploration using NVPsim. Fig. 6(a), Fig. 6(b) and Fig. 6(c) show the processor states when the capacitor sizes are 50nF, 200nF and 1000nF respectively. The power supply is a square wave: (1kHz, 5mW, 0.5 duty cycle). Fig. 6(d) shows the performance and energy statistics of NVP with different capacitor sizes and cache designs. The power supply is a square wave: (1kHz, 0.5mW, 0.5 duty cycle). Fig. 6(e) shows the cache miss rates affected by different strategies for backing up cache contents. Fig. 6(f) shows the performance and energy statistics of NVP under a real energy harvesting power supply.

图 A.6 设计探索

表 A.3 仿真器配置

SIMULATOR CONFIGURATION

Processor Frequency	1500MHz	Memory Size	16MB
Instruction Cache Size	8KB	Data Cache Size	8KB
Instruction Cache Assoc.	2	Data Cache Assoc.	2
$V_{restore}$	1.4V	V_{backup}	1.3V

表 A.4 非易失缓存设计

NVP CACHE DESIGN

No.	Data Cache Type	Inst. Cache Type	Strategy for Backup
0	RRAM	RRAM	N/A
1	nvSRAM	nvSRAM	Full Backup
2	nvSRAM	nvSRAM	LRU Partial Backup
3	RRAM	nvSRAM	Full Backup
4	RRAM	nvSRAM	LRU Partial Backup
5	nvSRAM	RRAM	Full Backup
6	nvSRAM	RRAM	LRU Partial Backup

A. 电容大小对非易失处理器性能的影响

我们首先探究了电容大小——一个 NVP 系统参数——对 NVP 系统性能和能量消耗的影响。如图 A.6(d)所示，运行时间随着电容大小的增加而降低，图 A.6(a)、图 A.6(b)、图 A.6(c)给出了电容变化下 NVP 在不同状态下的时间，在实验中，输入能量被设定得十分低以便模仿从实际能量采集设备中获取的能量。有趣的是，我们观察到在输入功率很低时，尽管电源供给并没有中断，处理器仍然会面临断电，这是由于处理器运行所需的功率超过了输入功率，电容电压持续减小，到达电压阈值，这是处理器开始备份系统状态和数据，一个大的电容意味着更大的能量缓冲，使得处理器能够在耗尽电容储存的能量前运行更长的时间，如图 A.6(c)所示。在另一方面，更大的电容需要更长的充电时间，因此处理器需要停留在关机状态更久，而一个小的电容使得处理器能够更频繁地重启工作，对电能的恢复反应更快。没有这样的仿真结果，我们不能够给出不同应用需求下是否需要一个更大的电容。电容的大小影响了输入功率比较小时的备份和恢复数量，当输入功率较大时，备份和恢复的数量主要由能量中断所决定。

B. 不同缓存设计的对比

实验结果表明使用 nvSRAM 缓存的非易失处理器在 6 中不同的设计中有着最佳的性能，这是由于 nvSRAM 有耕地的延时，并且访问所需能量更低。全部能量中的 55%被用来访问 RRAM 缓存，而只有 9%的能量被用来访问 nvSRAM 的缓存。对于 nvSRAM 缓存，37%的总能量由于漏电功率而损失，如果 nvSRAM 的漏

电功率能进一步被降低，nvSRAM 缓存消耗的能量会大幅降低。基于表格 4 中的第三个和第五个设计，我们发现 RRAM 更加适合作为只读的指令缓存，因为 RRAM 需要更高的写入能量，而且有更长的写入延迟。

部分备份策略的目标是降低峰值电流，提高系统的能量利用效率，应该使用仿真结果验证部分备份策略是否达到这一目标，作为一个例子，LRU 部分备份策略被这个仿真器测试，可以从仿真结果中看出，非易失储存器类型固定时，相比全备份，LRU 部分备份策略并没有提供更高的性能或者更高的能量效率，通过研究不同备份策略下的块缺失率，我们发现 LRU 部分备份策略由于丢失了活跃的即将被访问的缓存块增加了块缺失率，如图 A.6(e)所示，块缺失率在断电时不丢弃任何块时最低，这表明了设计非易失处理器时部分备份相比于全备份并不必要，这告诉我们一个体系结构级仿真工具对于设计探索是非常重要的。图 A.6(f)给出了不同 NVP 设计在实际的能量供给文件（太阳能）和不同 benchmark 下的性能。

结论

在这篇论文中，我们提出了一个非易失仿真器：NVPsim，它支持非易失仿真器的体系结构探索，这个仿真器被一个原型系统进行了验证。我们验证了 NVP 在不同片上缓存下的性能和能量数据，发现在系统需要收集能量的情况时，nvSRAM 作为非易失缓存的表现要好于 RRAM，此外，论文还阐述了非易失缓存的部分备份策略并不一定能够带来更好的性能，因此需要使用仿真器在不同配置和不同 benchmark 下进行详细测试。

原文索引

Gu Y, Liu Y, Wang Y, et al. NVPsim: A simulator for architecture explorations of nonvolatile processors[C]// Asia and South Pacific Design Automation Conference. IEEE, 2016:147-152.

综合论文训练记录表

学生姓名	张乐凡	学号	2013012229	班级	无35
论文题目	非易失处理器的仿真平台设计				
主要内容以及进度安排	2016秋季学期： - 12-14周 文献调研、问题抽象 - 15-18周 仿真框架编写（能量管理单元、模块间交换能量接口） 2017春季学期： - 1-2周 能量管理模块python端编写，完成模块掉、上电响应函数，完成CPU实际响应 - 3-5周 整体联调，简单demo，进行代码重构，分离能量管理模块的能量采集、系统能量状态机功能 - 6-7周 项目文档编写，项目wiki网站建立 - 8-10周 中期答辩，完成外设掉电响应，完成虚拟中断 - 11-13周 整理工作，进行一些仿真测试，设计benchmark - 14-16周 编写毕业设计论文，准备最终答辩 指导教师签字：孙凡南 考核组组长签字：徐海正 2017年 1月11日				
中期考核意见	对非易失处理器能量采集单元等关键模块进行了建模工作，初步完成仿真器框架与项目网页的建立。完成工作量达到总工作量的50%以上，取得了预期的阶段性成果。 考核组组长签字：徐海正 2017年 4月13日				

指导教师评语	论文针对应用能量采集环境下的非易失处理器设计这一研究热点，设计了一个可以评估非易失处理器系统性能的仿真平台。论文继承了Gem5仿真器的结构，添加了非易失处理器的电气特性，并对能量采集和管理单元和外设行为进行了抽象和建模。论文设计的仿真平台尝试解决了在非易失处理器系统设计前期性能评估和性能优化困难的难题，并通过对该该仿真器编写文档、开源代码等工作，初步构建了开放研究环境，具有较大的实际应用价值。论文写作较为规范，具有较强的逻辑性，反映作者掌握了相关领域的基础理论和专门知识以及进行科学研究的方法。 指导教师签字：孙小忆 2017 年 6 月 12日
评阅教师评语	论文实现了一个非易失处理器的软件仿真框架，此框架为用户提供了定义硬件能量行为的自由度，对研究非易失处理器体系架构的学术界有较大意义。本文结构规范，逻辑严密，对整个仿真系统的结构有着较为清晰的展示。 评阅教师签字： 2017 年 6 月 12日
答辩小组评语	论文书写规范，答辩阐述清楚，回答问题正确全面。反映论文作者张东凡具有较好的理论基础和较强的动手实践能力。答辩通过，建议其参加优秀论文评选。 答辩小组组长签字：徐淑正 2017 年 6 月 13日

总成绩：92

教学负责人签字：徐淑正

2017年 6 月 13日