# 面向物联网重编程的存储优化方法研究

邱杰凡 钱丽萍 黄 亮 陈庆章
(浙江工业大学 杭州 310014)

**摘 要** 未来在大多数物联网应用场景中,将允许使用者根据自己的需求复用当前已部署的物联网节点设备.从而要求节点设备具备对本地软件程序重编程的能力.然而,现有重编程方法普遍存在着传输和重组开销过高的问题.在工程实践中,通过使用大容量的扩展存储单元升级代码,可以有效避免由于传输代码引起的传输开销.因此,当需要对物联网节点软件频繁升级时,重编程的能量开销将主要取决于在节点本地读写代码引起的重组开销.为此,我们提出一种基于分页机制的重编程存储优化方法——AdvCache.该方法的核心思想是将相似的代码段以函数为单位聚合为函数页,放入低功耗的缓存中保存并执行.通过提高缓存的使用率和命中率,减少对高功耗存储单元的读写次数.此外,为了应对节点软件可能发生急剧变化的情况,我们提出了一种基于函数页调用关系的缓存替换算法,通过引入程序结构因素,进一步提高缓存命中率.实验表明 AdvCache 能够有效地降低对高功耗存储单元的读写操作数量.与先前的工作相比,AdvCache 的缓存命中率提高了 22%.

**关键词** 物联网;无线重编程;分页机制;程序结构;代码重建;传感器网络;信息物理融合系统
**中图法分类号** TP391 **DOI 号** 10.11897/SP.J.1016.2017.01888

# Research On Storage Optimization of Reprogramming Applied in Internet of Things

QIU Jie-Fan QIAN Li-Ping HUANG Liang CHEN Qing-Zhang
(*Zhejiang University of Technology, Hangzhou* 310014)

**Abstract** In the future deployment of Internet of Things (IoT), the function of a wireless node will be reconfigurable or re-programmable with respect to different application scenarios, which requires the wireless node to update its local software by over-air reprogramming. However, there exists a high energy overhead in the current reprogramming approaches caused by transmitted and rebuilt codes. In the field deployment, large-capacity extended memory is usually employed to store those updating codes without retransmitting them, so the reprogramming overhead is largely decided by the codes rebuilding consumption. Hence, we propose a reprogramming storage optimization approach based on paging mechanism—AdvCache for rebuilding codes. Our approach divides similar functions into several function pages which are stored and executed in a low-power cache. With the help of paging mechanism, the utilization and hit rate of cache are improved. Furthermore, in order to deal with the drastic change of node software, we introduce the program structure into cache replacement algorithm based on the function page. The experiment results show that AdvCache successfully reduces the rebuilding overhead and improves the average hit rate by 22% over earlier works.

**Keywords** IoT; over-air reprogramming; paging mechanism; program structure; rebuilding code; sensor network; Cyber-Physical System

收稿日期:2015-10-05;在线出版日期:2016-02-28.本课题得到国家自然科学基金(61502427,61379122,61502428,61379023)和浙江省自然科学基金(LY16F020034,LR16F010003)资助.**邱杰凡**,1984 年生,男,讲师,主要研究方向为嵌入式系统、传感器网络及物联网.E-mail:qiujiefan@zjut.edu.cn.**钱丽萍**,女,1981 年生,副教授,主要研究领域为无线通信和网络技术.**黄 亮**,男,1987 年生,讲师,主要研究方向为无线通信和物联网.**陈庆章**,男,1955 年生,教授,博士生导师,主要研究领域为传感器网络、定位技术和物联网.

## 1 引　言

在一些无线传感器网络的应用中，使用者对节点部署环境并不熟悉.在这些应用中，需要经过一段时间的数据采样收集和分析，当使用者获取足够信息后，才能开发出更有效的功能模块.这些功能模块一般通过无线重编程(over-air reprogramming)技术加载到节点上，从而实现节点本地软件的优化.此外，一些传感器节点可能肩负着多重任务，比如嵌入到楼宇的传感器节点潜在具备探测声波以及监测震动破坏的能力.但是受限的存储资源导致节点无法同时实现上述两种任务.而选择为不同的任务单独部署网络基础设施是浪费且没有效率的.也可以通过重编程实现任务的切换.

进而，随着物联网的发展，越来越多的物联网应用示范项目已经在多个领域取得了成功.不过应该看到这些物联网系统与传统传感器网络类似，往往都是为解决某一领域的特定问题或者满足特定区域的具体需求而构建的.在构建的过程中，各个网络由于采用了不同的硬件设备、软件模块甚至编程方法，形成了众多异构的物联网.网络之间相对独立，缺乏有效的协同.因此越来越多的研究者认为[1-3]：物联网应该是以连接和协调众多异构计算网络(如传感器网络和车联网)为目的的全局性网络.物联网的未来发展趋势是整合网络中的各种节点设备资源，由使用者根据自己的需求动态地组合复用设备，因此需要各种节点设备具备无线重编程的能力，以便动态地调整自身功能.

然而，物联网基础设施中包含大量电池供电的嵌入式节点设备，如果频繁地使用重编程技术升级节点设备上的软件程序，势必影响节点的使用寿命.这也是限制重编程技术大规模应用的主要瓶颈.无线重编程的能量开销主要来自于节点间发送和接收代码时产生的传输开销，以及节点上读写存储单元时产生的重组开销.目前大多数无线重编程方法对如何降低传输开销进行了深入的研究，其中比较有代表性的增量式重编程方法[4-11]，通过传输新旧代码的差异代码完成软件升级，可以有效减少传输开销.

这类增量式重编程方法通常需要在节点本地存储单元中重建新程序镜像.以较为流行的 TelosB 节点为例，其微控制单元(Microcontroller Unit，MCU)采用了内部 Flash＋RAM 的存储结构.当执行代码重组时，首先将差异代码读入节点板载专用非易失的外部 Flash 芯片中，与旧程序整合后生成新程序，最后将新程序写入 MCU 的内部 Flash 中.如表 1 所示，读写 Flash 具有较高的能量开销.多种传感器节点的存储单元读写能耗问题在参考文献[12-13]中亦有体现.尽管不同节点会导致能量开销数据上的差异，但总体上，重组开销相比于传输开销已经很难被忽略.

进而，在物联网应用中，可能存在着多个使用者在同一时段复用节点设备的情况.由于可以使用容量较大的外部 Flash 保存代码，因此在实际对节点升级的过程中，如果根据不同使用者的需求在几个程序镜像中反复切换，则可以预先将差异代码保存在大容量存储单元中，避免重复数据的传输，这时重编程开销将主要取决于在节点本地对新镜像的代码重组开销.

**表 1　TelosB 节点存储单元读写 1 KB 数据能量开销**

(单位：uJ)

| 读外部 Flash | 写外部 Flash | 读内部 Flash | 写内部 Flash | 读 RAM | 写 RAM |
|---|---|---|---|---|---|
| 1015 | 2458 | 785 | 1850 | ＜50 | 126 |

由表 1 可知，重组开销主要是来自于 Flash 的读写操作，为了降低重组开销必须减少对高功耗 Flash 读写操作的次数.在先前的研究中，我们提出了基于代码缓存的重编程方法 EasiCache[14]，该方法是将频繁更新的代码段放入低功耗的易失性存储单元 RAM 中完成更新并执行，可以有效减少对 Flash 的读写操作.另一方面，EasiCache 以函数为单位缓存代码段，代码段的尺寸具有不确定性，容易造成存储空间的浪费，降低缓存的使用率.另一方面，EasiCache 的缓存机制主要考虑局部性原理，即代码的修改也具有局部性，利用重编程过程中的动态特征(如：已被更新的次数)，对未来需要更新的函数做出预测.而在具体物联网应用过程中，应用需求改变具有随机性，节点上的软件可能发生剧烈变化.且每次变化之后的初始阶段，由于无法提供足够多的重编程动态特征信息，会导致缓存命中率明显下降.

为了解决上述重编程过程中缓存使用率以及命中率下降的问题，本文提出了一种基于分页机制的存储优化方法 AdvCache.该方法分别从以下 3 个方面展开研究：

首先,针对函数尺寸不确定导致的存储空间浪费问题.在 AdvCache 中,提出一种基于函数相似度的分页机制.通过分析函数之间的调用关系,设计基于函数相似度的聚类算法,尽量将同时被更新的相似函数放入同一个函数页(function page)中,同时提高缓存的使用率和命中率.

其次,针对函数页的缓存特点,重新设计函数的调用和加载方式.使用寄存器相对寻址方式代替直接寻址方式,通过对函数页内函数地址的集中管理,可以有效减少由于缓存函数页导致的读写操作.

最后,在考虑重编程动态特征的同时,将程序结构作为重编程静态特征引入函数页的缓存算法.该算法基于函数页之间的调用关系建立映射结构图,并利用马尔科夫链分析各个函数页在初始阶段被更新的可能性,可以有效提高连续升级初始阶段的缓存命中率.

本文第 2 节说明当前代码缓存方法可能存在的问题;第 3 节介绍 AdvCache 的设计与实现;第 4 节介绍实验场景并分析实验结果;第 5 节介绍相关工作;最后在第 6 节总结我们的工作.

## 2 问题描述

在我们的前期研究中,代码段是以函数为单位进行缓存.当函数被替换时,需要节点自适应地匹配合适的存储空间,放入相应的函数.然而随着重编程次数的增加,这种方法会引起代码碎片化的问题,并最终导致缓存使用率下降.为了量化缓存的碎片化程度,我们定义碎片化程度为

$$\text{碎片化程序}=\frac{\text{缓存中空闲空间尺寸之和}}{\text{缓存尺寸}} \quad (1)$$

我们使用 EasiCache 随机连续对 TelosB 节点上的软件升级,升级过程详见 4.1 节的介绍.TelosB 节点配置有一块拥有 10 KB RAM 以及 48 KB 内部 Flash 的 MCU.给定 RAM 缓存区大小为 8KB,图 1 给出了 EasiCache 在不考虑对缓存空间调整的前提下,连续 25 次升级,每次升级时的碎片化程度.

从图 1 中可以明显看出,当重编程次数超过 5 次以后缓存的碎片开始增加.经过 23 次以后,缓存区的碎片化程序已经达到了 30%以上.这意味着很多函数不能再被缓存到 RAM 中.只能通过重新调整缓存中所有函数的位置消除碎片.

此外,在实际执行代码缓存的过程中,被替换出

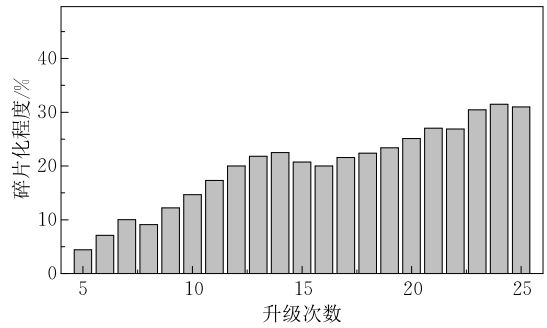


图 1 连续升级场景的碎片化程度

缓存区的函数将被尽量保存到内部 Flash 的原始位置上.然而如果函数在被缓存期间尺寸增大,EasiCache 借鉴了文献[15]的做法,直接将尺寸增大的函数放入主程序(.text 段)的末尾空闲空间,以避免对其他函数位置的调整.但是内部 Flash 的容量是有限的,随着软件升级次数的增加,如果末尾空闲空间消耗殆尽,势必需要重新调整内部 Flash 中所有函数的位置.图 2 给出了当连续升级节点软件时,内部 Flash 的读写字节数.在连续升级过程中,由于只有被替换出的函数需要回写到内部 Flash 中,因此对 Flash 的读写次数较少,但是如果发生末尾空间用尽的情况(如第 9 次和第 20 次升级),对内部 Flash 的读写数量将急剧增加.

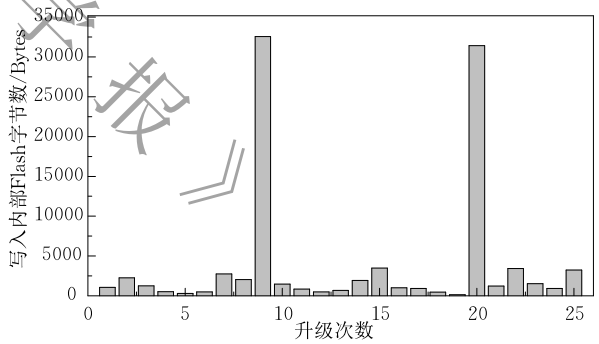


图 2 连续升级场景的写入内部 Flash 的字节数

为此,我们提出一种基于函数分页的存储优化方法——AdvCache,利用分页的思想,在降低碎片空间的同时,尽量避免由于回写函数导致的内部 Flash 空间耗尽的情况,减少对内部 Flash 的读写操作.

## 3 AdvCache 的设计与实现

本节主要由 3.1 节介绍的基于函数相似度的分页机制和 3.3 节介绍的替换算法组成.前者主要解决代码碎片化问题,后者通过分析程序结构提高函

数页的缓存命中率.同时,我们在 3.2 节和 3.4 节分别讨论了 AdvCache 中函数的调用与加载问题以及 AdvCache 的执行实现.

### 3.1　基于相似度的函数分页机制

由于节点重编程是对节点上某些参数以及功能的调整,修改对象仍然是相对独立的函数.且对某个功能的调整,往往是对多个相似函数的联合修改.为此,我们在 AdvCache 中提出一种基于相似度的函数分页机制.它通过将多个具有相似性的函数聚合成一个函数页(function page),使得每个函数页的尺寸相对固定.整个程序镜像将被平均地划分为多个以函数页为单位的代码段.同时,为了保证每个函数页可以被灵活地增量修改,参考 Koshy 等人[16]提出的方法,向每个函数页 $fp_x$ 的末尾增加一个尺寸为 $E_x$ 的溢出空间(slop region),专门用于函数尺寸增大时的位置调整,如图 3 所示.

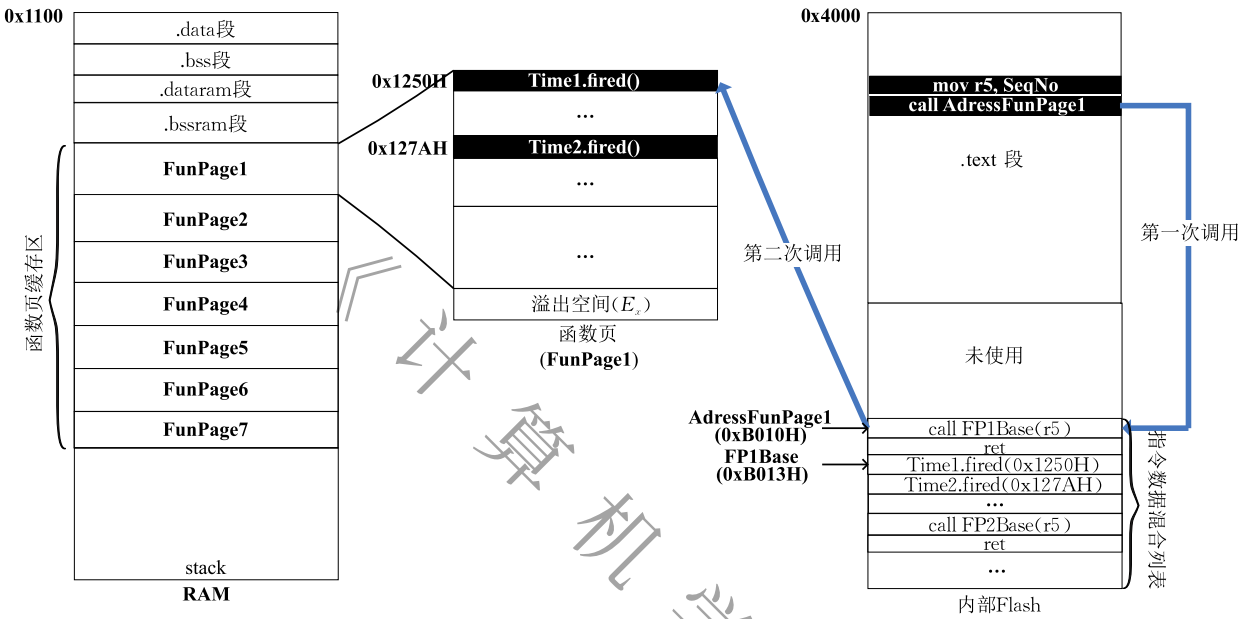


图 3　代码存储示意图及函数二次调用示意图(函数二次调用:位于函数页 FunPage1 中的函数 Time1.fired 地址为 0x1250,且已被缓存到 RAM 中.当保存在内部 Flash 上的某条指令要调用 Time1.fired 函数时,需要先跳转到指令数据混合列表中对应的函数页地址,地址为 0xB010H.再利用寄存器相对寻址方式获得保存 Timer1.fired 入口地址的内存单元地址(0xB013H),取得 Timer1.fired 的真实入口地址后,完成调用)

经过大量重编程实验,我们发现函数的修改通常具有单向性,即如果被调用函数(callee)发生了修改,会引起调用函数(caller)的修改,相反的情况则较少发生.举例来说,如果某个函数的入口地址发生了变化,则需要修改调用该函数的指令,这意味着必须对指令所处的函数做出修改.这种修改单向性,为我们描述函数之间的相似度提供了可能.

AdvCache 借鉴了推荐系统中常用的协同过滤思想描述函数之间的相似度.给定函数 $u$ 和函数 $v$,令 $N(u)$ 和 $N(v)$ 分别表示 $u$ 和 $v$ 中需要调用的函数.可以通过 Jaccard 公式计算函数 $u$ 和函数 $v$ 的相似度:

$$s_{uv}=\frac{|N(u)\cap N(v)|}{|N(u)\cup N(v)|} \tag{2}$$

或者可以通过余弦相似度计算:

$$s_{uv}=\frac{|N(u)\cap N(v)|}{\sqrt{|N(u)||N(v)|}} \tag{3}$$

如果采用余弦相似度定义每两个函数之间的相似度,会产生大量的计算开销.特别是,如果将计算相似度的工作移植到嵌入式设备上,会给嵌入式设备带来很大的计算开销.而实际上很多函数之间根本没有调用过相同的函数,即 $|N(u)\cap N(v)=0|$.按照这个思路,可以首先计算出 $|N(u)\cap N(v)\neq 0|$ 的函数对 $(u,v)$,然后再利用余弦相似度公式计算这些函数对的相似度.

在得到每个函数对的相似度之后,我们加权计算每个函数页的整体相似度 $S_k$:

$$\begin{aligned} S(f_1,f_2,\cdots,f_i)&=\sum_{u=1}^{i}\sum_{v=1}^{i}\beta_{uv}s_{uv}\\ \beta_{uv}&=\frac{1}{size(u)+size(v)} \end{aligned} \tag{4}$$

$\beta_{mn}$ 是权重值,与函数 $u$ 和函数 $v$ 的尺寸成反比例关系.即如果 $u$ 和 $v$ 的尺寸在当前函数页中所占的比重较小,且相似度较高,则函数页整体相似度较

高;反之,如果函数 $u$ 和函数 $v$ 相似度较高是由于函数 $u$ 和函数 $v$ 尺寸较大引起的,应该降低当前函数页的整体相似度.

函数分页的目的是尽可能将相似的函数放入同一个函数页中,因此将函数页的划分转化为一个优化问题.如式(5)所示:

$$\max \sum_{k=1}^{N} S_k(f_i, f_{i+1}, \cdots, f_j),\ i \in [1, K]$$
$$\text{s.t. } sz(fp_x) = sz(f_i) + \cdots + sz(f_j) + sz(E_x)$$
$$\sum_{x=1}^{N} sz(fp_x) < s_{\text{memory}} \tag{5}$$

优化的目标是使每个函数页的相似度最大化.其中 $S_k$ 为第 $k$ 个函数页中所包含多个函数的相似度函数.$K$ 为原始程序镜像中函数的个数,$N$ 为程序镜像中函数页的个数,且 $N$ 与函数页的划分密切相关.$sz(fp_x)$ 是单个函数页的尺寸,可以看到它是由多个相似函数 $(f_i, f_{i+1}, \cdots, f_j)$ 与溢出空间 $(E_x)$ 之和决定.由于溢出空间的存在,将导致分页后的镜像文件尺寸大于原始镜像文件,为了保证分页后的镜像文件能够被内部 Flash 容纳,需要增加限制条件使得函数页的总尺寸不能超过内部 Flash 的容量 $(sz_{\text{memory}})$.在第 4.4 节的实验中,我们将分析溢出空间对于存储开销的影响.

### 3.2 函数的调用与加载

在我们的前期研究 EasiCache 中,由于需要将代码以函数为单位放入或取出缓存,函数的入口地址势必会发生变化,如果修改每条调用该函数的指令,会引起大量读写操作.为此,我们采用了一种集中式管理的方式,将所有函数的入口地址保存到指令数据混合列表中,通过两次立即数寻址方式完成对函数的调用.

而在 AdvCache 中,若干函数已经被聚合到一个函数页中,当函数页整体被放入缓存区时,函数页包含的所有函数的入口地址都将发生变化.为此,需要在原有函数二次调用的基础上,采用更加高效的方式完成对函数地址的管理.如图 4 所示,我们采用了一种改进的指令数据混合列表.

| 函数数量 | 更新次数 | 函数页地址函数 | | 函数入口地址 | | |
|---|---|---|---|---|---|---|
| 5 | 1 | call FP1Base(r5) | ret | fun1 | fun2 | … |
| 10 | 3 | call FP2Base(r5) | ret | fun12 | fun23 | … |
| 7 | 8 | call FP3Base(r5) | ret | fun27 | fun9 | … |

图 4 改进的指令数据混合列表

列表的每一项对应一个函数页.每项的开头两个元素为函数页所包含的函数数量以及当前函数页被更新的次数.在函数页地址函数(AddressFunPage)中包含一条采用寄存器相对寻址方式的函数调用指令(call adss1(r5))以及一条返回指令(ret).最后一项由当前函数页中所有函数的真实入口地址组成.由于函数被划分到函数页后,其在函数页中所处的相对位置顺序可以被预先固定下来.因此,在指令数据混合列表中按照这个顺序保存每个函数的真实入口地址.

当调用某个函数时,实际上并没有直接跳转到真实的函数,而是首先跳转到函数页地址函数.如图 3 所示,在正式调用函数页地址函数之前,将函数在函数页中的位置序号(SeqNo)赋给寄存器 r5.利用 r5 作为变址寄存器,可以连续调用函数页中的所有函数,最终通过对基址(adss1)与寄存器 r5 内容求和,得到函数的真实入口地址,并正确调用相应函数.

当函数入口地址发生变化时,直接在混合列表中修改函数的地址,避免了修改每条调用该函数的指令.进而,一旦函数页被放入 RAM 的缓存区或者被回写到内部 Flash 时,函数页中所有函数的入口地址都将发生变化.这时根据新函数页的地址与原函数页的地址之间的差值,计算出函数页中所包含的每个函数的新入口地址,并对混合列表中的所有函数的入口地址连续进行修改.

由于在第二次调用中采用了寄存器相对寻址方式,需要由 AdvCache 自动在每一条函数调用指令前添加一条寄存器赋值指令.这个过程对于编程者来说是透明的.如果通过修改汇编文件(.s 文件)或者可执行文件(.exe 文件)添加该指令,可能会引起寄存器冲突.为此,需要由 AdvCache 在预编译阶段对镜像文件(app.c 文件)执行修改,即利用汇编语言混合编程添加该赋值指令.并最终交由编译器自适应地调整寄存器的使用,避免寄存器冲突.

### 3.3 函数页的替换策略

为了有针对性地缓存代码,我们在前期的研究中将代码的更新次数以及更新程度作为是否替换当前函数代码段的主要依据[14],并提出了最近最少变化(Least Recently Changed,LRC)替换算法.使用替换因子 $Rf(k,i)$ 决定在第 $k$ 次重编程时,第 $i$ 个函数是否被替换出缓存:

$$Rf(k,i) = C_k^{\alpha} \cdot CRF_i^{(1-\alpha)},\ \alpha \in [1,0] \tag{6}$$

其中,$CRF_i$ 表示第 $i$ 个函数在第 $k$ 次重编程过程中被修改的程度,而 $C_k$ 则表示第 $i$ 个函数被更新的次数.并利用比例因子 $a$ 调节修改程度和更新次数对

替换因子的影响.可见 LRC 算法的核心是基于重编程以往经验对未来动态地做出预测,即重编程的动态特征.

另一方面,通过大量实验观察,我们发现实际上函数之间的调用关系也与代码的修改存在着密切关系.在之前的讨论中,我们已经提到,对函数的重编程修改往往具有从被调用函数(callee)到调用函数(caller)的单向性.说明程序结构也会对重编程产生影响,我们称之为重编程的静态特征.

为此,可以将程序镜像中的所有函数页按照函数页之间的调用关系建立一张如图 5 所示的关系网.在图 5 中,每个函数页可以被视为一个顶点 $V$,每个顶点通过调用关系形成边 $E$,整个程序镜像转化为一个有向图 $G(V,E)$.假设程序镜像是由 $N$ 个函数页组成,则顶点个数为 $N$.当前函数页是否容易被修改,主要依赖于是否有很多函数页被当前函数页调用.如果一个函数页包含多个对其它函数页的调用,则其被修改更新的概率更高,应该更长时间地留在缓存中.基于上述假设,我们赋予每一个函数页缓存权重值 $W(fp_x)$,决定是否将函数页替换出缓存,如式(7)所示:

$$W(fp_x)=\sum_{(fp_x,fp_j)\in E}\frac{W(fp_j)}{IN(fp_j)} \tag{7}$$

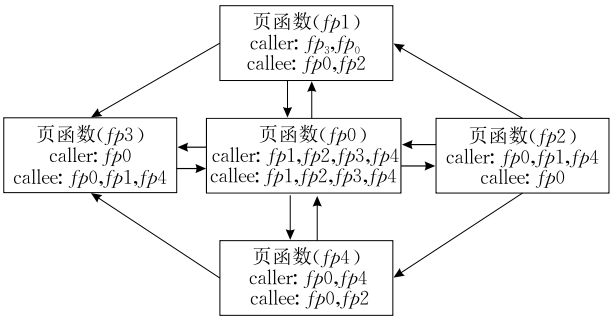


图 5　程序镜像中函数页的调用关系
(调用者(caller)和被调用者(callee))

函数页 $fp_i$ 调用了函数页 $fp_x$,则函数页 $fp_x$ 的缓存权重值 $W(fp_x)$ 由函数页 $fp_i$ 的权重值确定.$IN(fp_j)$ 是调用函数页 $fp_j$ 的函数页个数,即函数页 $fp_j$ 作为被调用者(callee)被调用的次数.定义缓存权重矩阵 $\boldsymbol{W}$ 是一个 $N$ 维列向量,其中每个分量代表一个函数页的权重值,如式(8)所示:

$$\boldsymbol{W}=(W(fp_1),W(fp_2),\cdots,W(fp_N)) \tag{8}$$

根据式(7),可以定义整个镜像程序的函数页相关性矩阵 $\boldsymbol{S}$ 如下:

$$\boldsymbol{S}_{xj}=\begin{cases}\dfrac{1}{IN(fp_x)}, & \text{if } (fp_x,fp_j)\in E\\ 0, & \text{其他}\end{cases} \tag{9}$$

由式(7)和(9),可以将缓存权重矩阵表示为

$$\boldsymbol{W}=\boldsymbol{S}^T\boldsymbol{W} \tag{10}$$

由于函数页之间的调用关系与具体的应用场景和分页策略密切相关,具有很强的随机性,因此可以将相关性矩阵 $\boldsymbol{S}$ 视为一个随机过程矩阵.其实质是一个马尔科夫模型,其中每一个函数页对应一个马尔科夫状态,函数页被其他函数页调用的过程是一个马尔科夫转化,这个转化表示当前函数页被修改时会在多大概率上引起调用它的函数页的修改.随着转化的不断进行,每个函数页的权重值将趋于稳定.一些作为调用者(caller)主动向外发起调用的函数页将具有更高的权重值.具有高权重值 $W(fp_x)$ 的函数页将在每次重编程过程中具有更高被修改的概率,将会被保留在缓存中.

综上所述,重编程的动态特征与静态特征都会对缓存的命中率产生影响,为此需要兼顾两种特征.重新定义每个函数页的替换权重值 $\overline{W}(fp_x)$ 为

$$\overline{W}(fp_x)=\left(1-\frac{1}{C_k}\right)*\frac{Rf(k,fp_x)}{Rf_{\max}(k,fp)}+\frac{1}{C_k}*\frac{W(fp_x)}{W_{\max}(fp)} \tag{11}$$

通过对反映重编程静态特征的 $W(fp_x)$ 以及反映重编程动态特征的 $Rf(k,fp_x)$ 归一化,以及引入更新次数 $C_k$,均衡对最终替换权重值的影响.很明显,函数页替换因子 $\overline{W}(fp_x)$ 在更新初期受到重编程静态特征即程序结构影响较大,随着更新次数 $C_k$ 的增加,$\overline{W}(fp_x)$ 退化为换因子 $Rf(k,fp_x)$,重编程动态特征对函数页替换权重的影响将越来越明显.

### 3.4　AdvCache 的执行实现

AdvCache 的执行实现将基于传感器网络操作系统 TinyOS[17],并可以分为两个阶段:对程序镜像编译的编译阶段,以及在部署之后对节点重编程的运行阶段,如图 6 所示.

如图 6(a)所示,在程序编译阶段,为了实现函数的分页机制,需要执行两次编译.在第一次编译过程中,首先针对 TinyOS 死代码消除机制做出修改,保留与当前需要执行函数相关的所有函数.然后,通过读取可执行文件(.exe)的符号表,建立函数表.它是由函数名、函数尺寸以及当前函数的调用关系三部分构成,如图 7 所示.

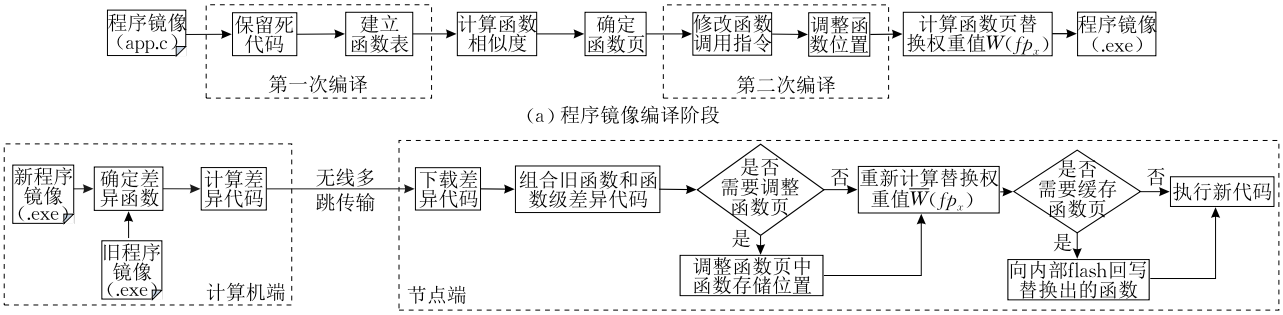


图 6 AdvCache 执行流程图

| 函数名 | 尺寸 | 被调用的函数及地址 | | | 调用的函数及地址 | | |
|---|---|---|---|---|---|---|---|
| *fun*0 | 97 B | *fun*1 | *fun*2 | … | *fun*6 | *fun*8 | … |
| *fun*10 | 102 B | *fun*12 | *fun*23 | … | *fun*25 | *fun*1 | … |
| *fun*7 | 234 B | *fun*27 | *fun*9 | … | *fun*0 | *fun*7 | … |

图 7 函数表

函数表由 AdvCache 在程序镜像编译后自动生成,并根据新的程序镜像进行调整. 利用函数之间的调用关系,可以计算出所有函数两两之间的相似度,用于确定函数页划分方案. 依照这个方案,重新调整所有函数的位置,并为每个函数页预留相应的溢出空间.

依据新位置,第二次重新编译源文件. 在预编译阶段,先修改函数的调用方式并加入寄存器赋值指令,随后调整函数的存储位置并修改函数表中对应函数的地址. 最终根据函数页的调用关系,计算出每个函数页的初始替换权重值,并生成分页的程序镜像(.exe 文件).

如图 6(b)所示,在节点部署之后,AdvCache 根据使用者编写的新程序镜像自动执行重编程任务. 首先将在计算机端根据新旧镜像以函数为单位生成差异代码. 随后,计算机通过串口将包含差异代码与重编程操作的增量脚本(delta script)发往汇聚节点(Sink). 汇聚节点收到增量脚本之后,以无线多跳(multi-hop)方式将增量脚本发往待升级节点. 在待升级节点收到包含差异代码的增量脚本后,将增量脚本下载到 RAM 中与旧函数代码组合,生成新函数.

如果新函数的尺寸超过了旧函数的尺寸,则需要在调整其所在函数页中相关函数的位置,以便放入尺寸增大后的新函数. 重新在计算当前函数页的整体替换权重值之后,判断是否需要缓存当前函数页. 如果需要,则将函数页整体放入缓存区. 最终,软件重启节点,开始执行新的程序镜像.

## 4 实验结果与分析

为了验证存储优化对重编程的影响,我们将在一个连续升级场景中测试 AdvCache 的性能. 连续升级场景是相对于单次升级场景而言,即在不重新部署节点的情况下,利用重编程技术连续升级节点上的软件程序. 现有的大多数重编程方法由于不涉及缓存机制,通常只进行单次升级实验. 为了更好地测试缓存机制对重编程的影响,我们将把实验重点放在连续升级的场景中.

### 4.1 实验场景设置

实验以 TinyOS 操作系统与 TelosB 节点作为软硬件平台,全面测试 AdvCache 的性能. TelosB 节点配备 2.4GHz 的 CC2420 无线芯片以及一块 16 位 MSP430F1611 微控制芯片. 为了验证 AdvCache 的有效性,我们分别设计了两组连续升级的场景.

在第 1 组升级场景中,将 TinyOS 标准例程 Oscilloscope以及 5 个应用于故宫博物馆温湿度监测的节点软件程序 EasiRoute(版本从 v0.1～v0.3)作为实验对象,如表 2 所示. 按照图 8 所示的字母顺序,连续对节点上的软件程序升级.

**表 2 测试传感器节点软件程序说明**

| 程序名称 | 功能描述 | 源文件尺寸(单位:行) | 可执行代码(单位:字节) |
|---|---|---|---|
| Oscillsocope | TinyOS 标准例程,采集环境数据后,直接转发给汇聚节点. | 26663 | 18810 |
| EasiRoute_v0.1 | 博物馆温湿度监测传感器节点程序,具有路由机制,可自组建树形网络. | 30519 | 20124 |
| EasiRoute_v0.11 | 在 v0.1 的基础上,增加数据缓存机制,在扩展外部 Flash 芯片保存感知数据. | 41158 | 22516 |
| EasiRoute_v0.18 | 在 v0.11 的基础上加入时间同步协议,判断当前时间. 解决博物馆夜间断电,数据拥堵问题. | 44282 | 26678 |
| EasiRoute_v0.21 | 在 v0.18 的基础上加入负载均衡算法. 同时,将删除节点的数据采集功能,节点仅作为数据路由节点. | 42152 | 22424 |
| EasiRoute_v0.3 | 增加 UDP 协议,支持上位机通过互联网访问传感器节点. | 48252 | 29902 |

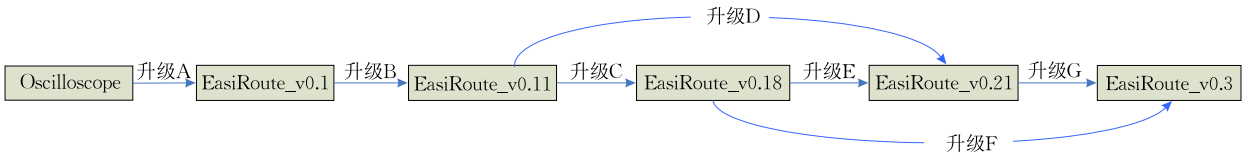


图 8　节点软件程序连续升级流程图

升级 A：Oscilloscope 是一个感知数据采集传输程序，升级到 EasiRoute_v0.1 之后，加入路由机制.

升级 B：升级博物馆监测程序，从 v0.1 版到 v0.11 版.升级后，新增加缓存功能，将感知数据保存在扩展外部 Flash 中.

升级 C：升级博物馆监测程序，从 v0.1 版到 v0.18 版.升级后，修复一个 BUG.解决了由博物馆夜间断电引起的，传感器节点向汇聚节点频繁发送数据的问题.

升级 D：版本从 v0.11 升级到 v0.21.增加一个负载均衡算法，根据节点的电池电量来调整节点的休眠时间.

升级 E：版本从 v0.18 升级到 v0.21.升级之后的新版本将移除节点的感知功能.传感器节点作为路由节点，仅负责转发来自其他节点的数据.

升级 F：版本从 v0.18 升级到 v0.3.本次升级过后，节点安装 UDP 协议，支持上位机可以通过互联网访问对应节点.

升级 G：版本从 v0.21 升级到 v0.3.

在第 2 组连续升级场景中，仍然使用以上 6 个软件程序，但采取随机升级的策略，不再固定升级的顺序.我们将从缓存命中率、能量开销以及存储开销 3 个方面测试 AdvCache 的性能，并在最后讨论 AdvCache对程序执行效率的影响.

### 4.2　缓存效率

首先，按照图 8 中的字母顺序升级节点软件，并测试缓存命中率.缓存命中率定义为命中函数的尺寸与需要更新函数的总尺寸之比.RAM 缓存区容量设定为 6144 B，函数页默认尺寸为 512 B.如图 9 所示，将 AdvCache、EasiCache 以及最近最少使用(Last Recently Used，LRU)算法进行对比.其中 EasiCache 默认采用最近最少替换算法(Least Recently Changed，LRC).

AdvCache 的缓存算法由于考虑了重编程的静态特征即程序的结构，在升级的初期具有较好表现.例如在首次升级 A 中，由 TinyOS 例子程序 Oscilloscope 升级到 EasiRoute_v0.1 时，由于数据

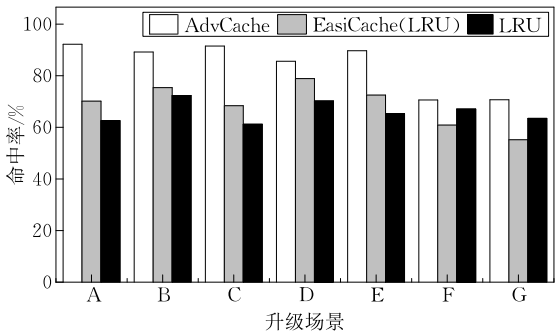


图 9　字母顺序连续升级场景中的缓存命中率

采集功能以及无线收发功能在 Oscilloscope 中已经实现.此外，通过对 TinyOS 死代码的消除[18]，汇聚树路由协议的模块实际上已经被烧写到节点，升级 A 更多的是在应用层激活这个模块，并根据 Easi-Route_v0.1 的要求对相应函数做出调整.

AdvCache 通过分析函数页之间的调用关系，可以在连续升级的初期更加精确地预测更新“热点”函数页，并将它们放入缓存中，其缓存命中率达到了 92.2%.而 EasiCache 使用的 LRC 算法以及 LRU 算法需要根据函数被更新的信息判断函数是否被替换，在升级的初期(升级 A 到升级 G)，二者的平均缓存命中率低于 75%.在升级 A 的过程中，仅有 70.17%和 62.56%.

升级 F 与升级 G 较为特殊，由于实现 UDP 协议的模块没有被写入，需要大量更新原本保存在内部编程 Flash 中的函数，最终导致 3 种缓存算法的命中率大幅下降.

为了分析重编程动态特征对缓存的影响，我们在另外的随机连续升级场景中，将节点上的软件程序，随机连续升级到以上 6 个程序.记录 5 次升级的命中率，求平均值，并将其定义为随机连续升级的平均命中率.图 10 显示了前 35 次升级的平均命中率.很明显，AdvCache 的表现在升级的初期(前 15 次升级)，平均命中率达到了 90.3%，优于 EasiCache 和 LRU 的 76.3%和 74.4%.

但是随着升级次数的增加，一方面大量代码段被重复加载到节点上，EasiCache 和 LRU 在获取足够多的编程动态特征信息以后，命中率不断提高.由

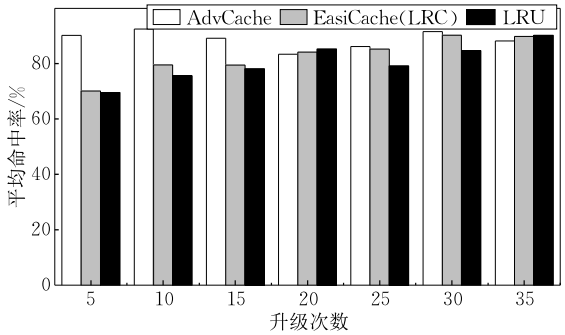


图 10 随机顺序连续升级场景中的平均命中率

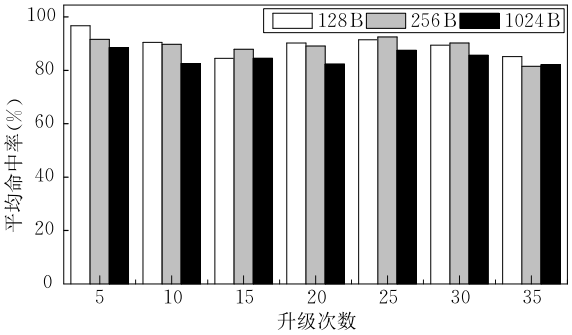


图 11 不同函数页尺寸的平均命中率

于 AdvCache 需要考虑重编程的动态特征以及静态特征，在 16 次～20 次的平均命中率较 EasiCache 和 LRU 分别下降了 0.85%和 1.8%. 而随着升级次数的增加，调节因子 $1/C_k$ 的存在使得 AdvCache 的函数页替换权重将逐渐退化为替换因子 $Rf(k, fp_x)$，其表现接近 EasiCache 的 LRC 算法.

在缓存区容量(6144 B)不变的前提下，我们测试了不同函数页尺寸对缓存命中率的影响，如图 11 所示. 当函数页尺寸较小时，可以更加灵活地将拥有高替换权重值的函数页放入缓存. 因此在升级的初期，128 B 函数页获得了较高的平均命中率. 但是较小的函数页尺寸意味着一些尺寸较大的函数无法被分页. 因此在 15 次连续升级之后，128 B 函数页平均命中率开始下降. 反之采用较大尺寸的函数页(如 1024 B)，在升级初期(1～10 次升级)与末期(30～35 次升级)的命中率都较为稳定. 需要指出，得益于 128 B 函数页的灵活性，第 17、19 以及 40 次升级的缓存命中率超过 97%，导致平均命中率较 512 B 有小幅提升. 然而从缓存命中率稳定性的角度出发，推荐使用介于二者之间的函数尺寸(512 B)，在升级的初期和末期都具有较好的表现.

### 4.3 重组开销

重编程的能量开销主要来自无线传输代码的传输开销以及在对节点本地存储单元执行读写操作的重组开销. 由表 1 可知，对节点 MCU 内部易失性存储单元 RAM 的读写能量开销要远远小于非易失性存储单元 Flash. 以 TelosB 节点为例，对 MCU 内部 Flash 以及板载外部 Flash 的单位字节的读写能量开销分别约为 RAM 的 14.6 倍和 19.3 倍. 因此在考察重编程的重组开销时，重点将放在对 MCU 内部 Flash 以及扩展外部 Flash 执行读写操作的数量.

我们将 Tiny Module-Link[15] 以及 EasiCache 作为比较对象. 其中 Tiny Module-Link 已经考虑了存储单元读写开销问题，提出在低功耗的 RAM 中以函数为单位完成代码重组，但被更新的函数仍然需要回写到内部 Flash 中. EasiCache 与 AdvCache 则直接使用低功耗 RAM 缓存代码，可以有效减少对内部 Flash 的读写操作.

表 3 和表 4 分别显示了从升级 A～升级 G 的连续升级场景中，3 种重编程方法对 MCU 内部 Flash 的读写操作数量. 在实际应用中，由于 AdvCache 与 EasiCache 可以直接在 MCU 内部完成对函数的升

**表 3 完成升级需要从 Flash 读取的代码量/Bytes**

| | | 升级 A | 升级 B | 升级 C | 升级 D | 升级 E | 升级 F | 升级 G |
|---|---|---|---|---|---|---|---|---|
| Tiny Module-Link | 读外部 Flash | 3278 | 1252 | 1578 | 1314 | 872 | 4214 | 4562 |
| | 读内部 Flash | 3025 | 2456 | 2894 | 2848 | 3010 | 35780 | 33678 |
| EasiCache | 读外部 Flash | — | — | — | — | — | — | — |
| | 读内部 Flash | 902 | 614 | 918 | 594 | 828 | 36678 | 32172 |
| AdvCache | 读外部 Flash | — | — | — | — | — | — | — |
| | 读内部 Flash | 236 | 266 | 242 | 856 | 328 | 1112 | 1146 |

**表 4 完成升级需要写入 Flash 的代码量/Bytes**

| | | 升级 A | 升级 B | 升级 C | 升级 D | 升级 E | 升级 F | 升级 G |
|---|---|---|---|---|---|---|---|---|
| Tiny Module-Link | 写外部 Flash | 3678 | 1252 | 1578 | 1314 | 872 | 4214 | 4562 |
| | 写内部 Flash | 3214 | 2022 | 2760 | 3090 | 1320 | 30258 | 31142 |
| EasiCache | 写外部 Flash | — | — | — | — | — | — | — |
| | 写内部 Flash | 578 | 456 | 1342 | 1090 | 1278 | 30958 | 31588 |
| AdvCache | 写外部 Flash | — | — | — | — | — | — | — |
| | 写内部 Flash | 336 | 156 | 148 | 1512 | 752 | 2544 | 2626 |

级，因此不会对扩展外部 Flash 执行读写操作，而 Tiny Module-Link 则需要将增量脚本暂存到外部扩展 Flash 中. 另一方面，Tiny Module-Link 在每次升级时会将已更新的函数从 RAM 回写到 MCU 内部 Flash 中，因此对内部 Flash 的读写数量甚至超过了对外部 Flash 的读写数量，在 7 个升级场景中读写数量平均为 2781 B.

EasiCache 和 AdvCache 将代码放入低功耗的 RAM 中缓存，有效减少了对内部 Flash 的读写操作. 并且 AdvCache 在升级 A、C、E 三个场景中的平均缓存命中率较 EasiCache 高出 25.3%，这意味着更多的函数已经被保存到了 RAM 中，而不需要再读取内部 Flash 中保存的代码，相应地，写到内部 Flash 中的代码也更少. 比较特殊的是在升级场景 D 中，尽管 AdvCache 的命中率要高于 EasiCache，但是由于发生了一次函数页的替换操作，原本保存在内部 Flash 中的函数页需要与缓存中的函数页(512 B)交换，整体读写数量较 EasiCache 增加了 40.1%.

在最后两个升级场景中，由于增加了对 UDP 协议的支持，大量的函数被更新，且更新过后尺寸增大的函数需要被回写到内部 Flash 中. 由于 EasiCache 与 Tiny Module-Link 都采用了在主程序尾部空闲空间保存函数的策略. 这时，写入内部 Flash 函数的尺寸实际上超出了内部 Flash 的容量，必须重新调整保存在内部 Flash 中所有函数的存储位置，极大地增加了对 Flash 的读写操作. 而 AdvCache 由于采用了分页机制，且每一页都有相应的溢出空间，保证了对函数位置的调整将发生在函数页内部，因此较前两者的读写操作数量分别下降了 94.2%和 94.3%.

### 4.4　存储开销

受到函数尺寸不确定的影响，以函数为单位缓存代码，势必产生碎片化的问题. 图 12 显示了采用 AdvCache 和 EasiCache 在随机连续升级过程中缓存区的碎片化程度. 碎片化程度定义已由式(1)给出，缓存区容量默认为 6144 B.

AdvCache 采用分页之后，每个函数页为保证能够在页内完成函数位置的调整，需要增加一个溢出空间，同样会带来碎片化的问题. 因此在连续升级的初期，EasiCache 的碎片化程度明显高于 AdvCache. 然而由于 EasiCache 的缓存区函数与函数之间没有溢出空间的存在，因此随着函数更新的不断进行，其碎片化程度进一步提高.

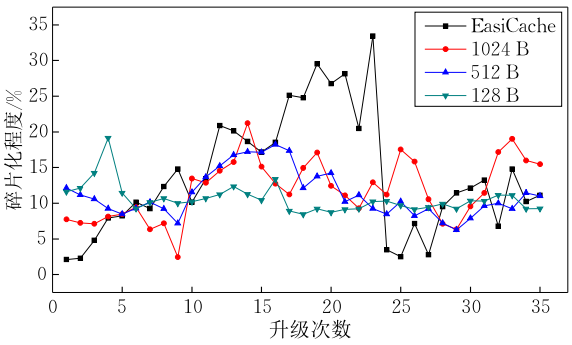


图 12　随机连续升级场景的碎片化程度
(EasiCache 与不同尺寸函数页)

在这个过程中，EasiCache 可以通过局部调整函数缓存位置的做法，降低碎片化程度. 如在使用 EasiCache 的第 7 次升级中，由于局部连续存储的函数中，有部分函数更新过后尺寸变小，可以通过调整这些函数的存储位置，使得更新过后尺寸变大的函数放入，因此碎片化程度较 EasiCache 第 6 次升级下降了 8%. 但是 EasiCache 的碎片化程度在总体上是不断上升的趋势. 在第 23 次升级的过程中，其碎片化程度超过了 30%，很难再缓存新的函数，因此不得不重新调整缓存区内所有函数的位置. 调整完成之后，碎片化程度下降到 3%以下.

此外，函数页的尺寸也会对碎片化程度产生直接影响. 在图 12 中，我们分别对比了采用尺寸为 128 B，512 B 以及 1024 B 的函数页之后各次升级的碎片化程度. 很明显，随着函数页尺寸的增加，多个函数更加容易被聚合到同一个函数页中，溢出空间的尺寸也较小，因此在升级初期，1024 B 函数页具有较少的碎片空间，缓存使用率也较高. 然而我们发现，多次升级之后，由于函数页尺寸较大，溢出空间相对于函数的保存位置较远，当多个函数尺寸增加时，调整过程也更加困难. 在大多数情况下，大尺寸函数页是通过对函数页内局部函数位置的调整来完成尺寸增加函数的放入，因此其碎片化程度也较高. 并且每次使用溢出空间调整函数位置，波及范围较大，能够大幅降低碎片化程度，表现为碎片化程度曲线波动较大，如图 12 所示.

相反尽管采用较小函数页，由于每个函数距离溢出空间较近，可以很方便地通过溢出空间来放入尺寸增大的函数，因此尺寸为 512 B 和 128 B 的函数页，相较于尺寸为 1024 B 的函数页，碎片化程度分别下降了 5.5%和 10.8%. 如图 12 所示，由于每次通过溢出空间调整函数位置时，波及范围较小，所以碎片化程度变化也较小，表现为碎片化程度曲线较

为平滑.

### 4.5 AdvCache 对执行效率的影响

为了方便对函数页的管理,AdvCache 采用二次调用方式完成对函数页内函数的调用.与直接调用函数相比,这种二次调用方式需要在第 1 次调用之前对寄存器(r5)赋值,并且在第 2 次调用中,用寄存器相对寻址方式找到函数的正确入口地址.以 MSP430 为例,每次函数调用,实际上增加了 1 条数据传输指令(mov)、1 条调用指令(call)和 1 条返回指令(ret).执行 3 条指令分别需要 3、8 和 5 个时钟周期,因此每次执行对函数的调用将增加 16 个时钟周期.

图 13 给出了 6 个传感器节点程序在采用二次调用后,执行效率受到的影响.受影响最大的是 EasiRoute_v0.3,其执行效率较原始程序下降了 17.9%.该程序由于加入 UDP 协议模块,程序的整体复杂度大幅提升,其函数总量较 EasiRoute_v0.18 增加了 11.2%.其他 5 个程序的执行效率下降的百分比均超过了 10%.很明显,函数的二次调用确实会对程序的执行效率产生较大影响.然而,从物联网软件升级的动态性角度考虑,采用函数二次调用能够有效较少对原始程序的修改,且为函数页的管理提供了便利,提高了重编程的效率.与重编程产生的能量开销相比,代码执行所需要的能量开销几乎可以忽略不计.因此,为了降低重编程能量开销,推荐采用二次调用方式实现对函数的调用.

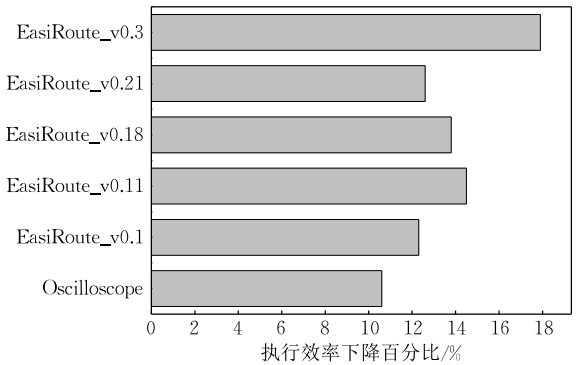


图 13 函数二次调用对程序执行效率的影响

## 5 相关工作

早期的重编程方法较少考虑能量开销问题或仅仅以间接方式来减少对网络整体的负担.例如 Deluge[19]需要同时将完整的代码镜像和软件升级协议加载到节点之上.Stream[20]针对 Deluge 的不足,通过预装重编程协议的方式减少更新时节点失效的时间.在 MobileDeluge[21]中,引入移动 Deluge 基站,保证待升级节点位于基站一跳之内,以提高代码传输链路的质量.MNP[22]中考虑了多跳传输方式对软件升级的影响,并提供了一种重编程发起方的选择算法,尽量选择邻居节点作为更新发起方,减少程序镜像被转发的次数.CORD[23]则应用了一种二阶段(two phase)策略来降低能量开销.新镜像首先被分发给在第 1 个阶段形成的节点集合,然后在第 2 阶段,由这个集合中的节点作为发起方向剩余节点转发新镜像.另外,针对物联网感知网络存在的离散性和动态性,一些研究者提出了基于编码的重编程方法[24-26],利用增加的冗余数据,保证单次传输的成功率,避免重传的发生.在这些基于编码的重编程方法中,方法之间的区别主要是采用了不同的编码原理或者编码方式.然而上述方法由于需要传输未经处理的程序镜像,甚至还要加入其它冗余编码,都存在着较为严重的能量开销问题.

为了降低对传感器节点重编程过程中的传输开销,一些研究引入了压缩算法,如 Sadler 等人[27]研究了针对节点设备的自适应压缩算法.Tsiftes 等人[28]则直接使用了 GZIP 对镜像进行了压缩.相应地,压缩程序镜像意味需要由节点本地完成对代码的解压,势必带来大量计算开销以及存储空间短缺问题.

目前,针对重编程过高的能量开销,已有研究者提出了增量式重编程[4-11],其核心思想是仅仅通过传输新旧代码镜像的差异代码来减少传输代码量.根据实现的手段不同,它们主要可以分为两类:一类是以提高代码相似度为手段来减少差异代码量.如 Zephyr[5]和 Hermes[4]试图通过固定函数和全局/静态变量的引用地址来提高新旧代码镜像的相似度.Shafi 等人[8]将软件克隆检测技术应用到新旧软件的监测上,最大限度地提升代码相似度.Li 等人[9]提出了一种升级感知(update-conscious)编译器.由于可执行代码的大部分内容是对 MCU 通用寄存器执行的操作,升级感知编译器的核心是在生成新镜像之前,通过读取原始镜像内容,对新镜像各个函数中通用寄存器的使用做出调整,以此提高新旧镜像的相似度.另一类,则是以设计更加有效的差异代码生成算法为手段,尽量匹配新旧镜像中的相同代码段.Hu 等人[7]借鉴了 Rsync 的思想,提出了 RMTD 算法,该算法借鉴了 Rsync 算法思想,时间复杂度从$O(n^3)$降到了$O(n^2)$,但空间复杂度则达到

了$O(n^3)$. 因此在处理尺寸稍大的镜像文件时，即使对于通用 PC 机也是很大的负担. Dong 等人整合 Zephyr 和 RMTD 的优点提出了 R2[10] 以及更加完善的 R3[11]. 相对于 Zephyr，R2 和 R3 使用了相对地址无关代码，即将所有的引用地址都初始化为零来保持新旧镜像的相似度. 同时对 RMTD 算法进行了改进，成功将空间复杂度降至 $O(n)$. 然而，增量式的重编程方法并没有将新镜像传输到节点. 因此节点需要根据已经保存的原始镜像重组新的代码镜像. 在这一过程中，不合理的代码重组方法可能引起大量对节点上的非易失性存储单元的读写操作，从而增加整体的重编程开销.

由于目前大多数已知的重编程研究主要是源自于传统的传感器网络，传感器节点上执行的任务通常不会经常的调整，基本不用考虑在频繁软件升级的情况下如何合理的对节点代码进行重组. 例如在增量式重编程方法 Zephyr[5] 中，尽管研究者在实验的最后对非易失性存储单元的读写能量开销进行了测试，并且对当前的代码重组方法提出了质疑，但没有进一步展开研究. Kim 等人提出了 Tiny Module-Link[15] 方法，该方法将新镜像以函数为单位放入了 RAM 中重组，重组之后新函数需要回写到内部非易失性存储单元保存. 此外，Koshy 和 Pandey[16] 试图通过给每个函数末尾添加溢出空间(slop region)，支持对保存在内部非易失性存储单元中的原始函数进行直接修改. 这样做的好处是当函数被更新后其代码量可能增长，而这部分增长可以被放入溢出空间中.

在另外一些研究中，即使提出使用易失性存储单元保存频繁升级的代码段，其目的也不是纯粹为了降低能量开销. 例如 Dong 等人[18] 提出了一种使用 RAM 保存更新代码的方法 Elon. 该方法的提出是为了解决由于电池电压下降导致的对非易失性存储单元读写操作失败的问题. 为此，Dong 等人尝试将最有可能升级的函数放入不受电压变化影响的 RAM 中保存. 当需要更新时，直接使用新的函数替换原始函数. 从严格意义上讲，Elon 并不是一种增量式重编程方法；并且，被放入 RAM 中的函数实际上是必须由开发者在开发阶段指定，当节点部署以后无法改变，不涉及缓存功能.

## 6　结束语

本文介绍了一种应用于物联网节点重编程的存储优化方法 AdvCache. 该方法通过分页机制，将最有可能同时更新的相似函数聚合到一个函数页中，在降低缓存碎片化程度的同时，可以提高缓存的命中率. 并通过在每一个函数页中添加溢出空间实现了函数位置的局部调整，

另一方面，由于物联网应用需求改变具有随机性，节点软件程序可能发生剧烈变化. 而单纯考虑连续重编程过程中的动态特征，在软件连续升级初期，命中率较低. 为此，我们将重编程的静态特征即程序结构引入基于函数页的替换算法. 可以有效缓解由于软件程序升级初期，动态特征信息不足导致的命中率下降问题. 通过在连续升级场景中的实验，我们从缓存命中率、存储开销以及能量开销三个方面验证了 AdvCache 的有效性，并在最后讨论了 AdvCache 对于程序执行效率的影响.

在未来的工作中，我们将继续研究与函数分页机制相适应的差异代码生成方法，根据每个函数页的特征简化更新函数所需的操作指令数量，进一步降低物联网重编程的传输开销.

**致　谢**　本文的部分内容来自于第一作者博士期间在中科院计算技术研究所的研究工作，这里特别感谢崔莉研究员对本文工作的支持！

## 参　考　文　献

[1] Wu Jian-Jia, Zhao Wei. WInternet: From Net of Things to Internet of Things. Journal of Computer Research and Development, 2013, 50(6): 1127-1134(in Chinese)
(武建佳，赵伟. WInternet:从物网到物联网. 计算机研究与发展，2013，50(6)：1127-1134)

[2] Guinard D, Trifa V, Mattern F, Wilde E. Architecting the Internet of Things. Berlin: Springer, 2012

[3] Jara A J, Zamora M A, Skarmeta A F G. Glowbal IP: An adaptive and transparent IPv6 integration in the Internet of Things. Mobile Information Systems, 2012, 8(3): 177-197

[4] Panta R K, Bagchi S. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks// Proceedings of the 28th IEEE International Conference on Computer Communications. Rio de Janeiro, Brazil, 2009: 639-647

[5] Panta R K, Bagchi S, Midkiff S P. Zephyr: Efficient Incremental reprogramming of sensor nodes using function call indirections and Difference Computation//Proceedings of USENIX Annual Technical Conference. San Diego, USA, 2009: 1-14

[6] Jeong J, Culler D. Incremental network programming for wireless sensors//Proceedings of the 1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks. Santa Clara, USA, 2004: 25-33

[7] Hu J, Xue C J, He Y. Reprogramming with minimal transferred data on wireless sensor network//Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems. Macau, China, 2009: 160-167

[8] Shafi N B, Ali K, Hassanein H S. No-reboot and zero-Flash over-the-air programming for Wireless Sensor Networks//Proceedings of the 9th IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks. New Orleans, USA, 2013: 371-379

[9] Li W J, Zhang Y T, Yang J, Zheng J. Towards update-conscious compilation for energy efficiency in wireless sensor networks. ACM Transactions on Architecture and Code Optimization, 2009, 6(4), article No. 14

[10] Dong W, Liu Y H, Chen C, Bu J J. R2: Incremental reprogramming using relocatable codes in networked embedded Systems. IEEE Transactions on Computers, 2013, 62(9): 1837-1849

[11] Dong W, Mo B, Huang C, et al. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems//Proceedings of the 32nd IEEE Conference on Computer Communications. Turin, Italy, 2013: 14-19

[12] Shnayder V, Hempstead M, Chen B R, et al. Simulating the power consumption of large-scale sensor network applications//Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems. Sydney, Austrlia, 2007: 188-200

[13] Chu D, Popa L, Tavakoli A, et al. Mercury: A wearable sensor network platform for high-fidelity motion analysis//Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems. Berkeley, USA, 2009: 183-196.

[14] Qiu Jie-Fan, Li Dong, Shi Hai-Long, Cui Li. EasiCache: A low-overhead sensor network reprogramming approach based on cache mechanism. Chinese Journal of Computers, 2012, 35(3): 555-567(in Chinese)
(邱杰凡, 李栋, 石海龙, 崔莉. EasiCache:一种基于缓存机制的低开销传感器网络代码更新方法. 计算机学报, 2012, 35(3): 555-567)

[15] Kim S K, Lee J H, Hur K. Tiny module-linking for energy-efficient reprogramming in wireless sensor networks. IEEE Transaction on Consumer Electronics, 2009, 55(4): 1914-1920

[16] Koshy J, Pandey R. Remote incremental linking for energy-efficient reprogramming of sensor networks//Proceedings of the 2nd European Workshop on Wireless Sensor Networks. Istanbul, Turkey, 2005: 354-365

[17] Levis P, Madden S, Polastre J, et al. TinyOS: An operating system for sensor network. Ambient Intelligence, 2005: 115-148

[18] Dong W, Liu Y H, Chen C, et al. Elon: Enabling efficient and long-term reprogramming for wireless sensor networks. ACM Transactions on Embedded Computing Systems, 2014, 13(4), article No. 77

[19] Hui J W, Culler D. The Dynamic behavior of a data dissemination protocol for network programming at scale//Proceedings of the ACM Conference on Embedded Networked Sensor Systems. Baltimore, USA, 2004: 84-91

[20] Panta R K, Khalil I, Bagchi S. Stream: Low overhead wireless reprogramming for sensor networks//Proceedings of the 26th IEEE International Conference on Computer Communications. Anchorage, USA, 2007: 928-936

[21] Zhong X Y, Navarro M, Villalba G, et al. MobileDeluge: Mobile code dissemination for wireless sensor networks//Proceedings of the 11th IEEE International Conference on Mobile Ad-hoc and Sensor System. Pennsylvania, USA, 2014: 363-370

[22] Kulkarni S S, Wang L. MNP: Multihop network reprogramming service for sensor networks//Proceedings of the 25th IEEE International Conference on Distributed Computing Systems. Columbus, USA, 2005: 7-16

[23] Huang L, Setia S. CORD: Energy-efficient reliable bulk data dissemination in sensor networks//Proceedings of the 27th IEEE Conference on Computer Communications. Phoenix, USA, 2008: 1-14

[24] Hagedorn A, Starobinski D, Trachtenberg A. Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes//Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks. St. Louis, USA, 2008:457-466.

[25] Rossi M, Zanca G, Stabellini L. SYNAPSE: A network reprogramming protocol for wireless sensor networks using fountain codes//Proceedings of the 5th IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks. Santa Clara, USA, 2008: 188-196

[26] Du W, Li Z J, Liando J C, Li M. From rateless to distanceless: Enabling sparse sensor network deployment in large areas//Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems. Memphis, USA, 2014: 134-147

[27] Sadler C M, Martonosi M. Data compression algorithms for energy-constrained devices in delay tolerant networks//Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems. Baltimore, USA, 2004: 265-278

[28] Tsiftes N, Dunkels A, Voigt T. Efficient sensor network reprogramming through compression of executable modules//Proceedings of the 9th IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks. New Orleans, USA, 2014: 359-367

**QIU Jie-Fan**, born in 1984, Ph. D., lecturer. His research interests include embedded system, sensor network and Internet of Things.

**QIAN Li-Ping**, born in 1981, Ph. D., associate professor. Her research interests are wireless communication and networking.

**HUANG Liang**, born in 1987, Ph. D., lecturer. His research interests include wireless communication and Internet of Things.

**CHEN Qing-Zhang**, born in 1955, Ph. D., professor, Ph. D. supervisor. His research interests include sensor network, localization, and Internet of Things.

**Background**

In this paper, we proposed a reprogramming storage optimization approach based on paging mechanism. This approach mainly solves the high rebuilding overhead caused by writing the nonvolatile memory of sensor node during the reprogramming procedure.

In traditional sensor network, with respect to lowering the reprogramming overhead, the most of research works focus on reducing transmitted overhead such as incremental reprogramming approaches. However, in the Internet of Things, the software on sensor node is possibly updated frequently and drastically. The rebuilding overhead becomes more important.

In existing work, Kim S K et al. use the RAM to rebuild codes in order to avoid writing/reading the high-power external Flash, but give a less consideration for writing/reading the internal Flash. In our early works, we had proposed EasiCache which caches and executes the frequently changed codes in low-overhead RAM for reducing the rebuilding overhead. However, the uncertain size of cached codes leads a code fragment problem which lowers the cache utility and hit rate.

For this, we propose AdvCache—a storage optimization approach based on paging mechanism. It unifies the size of codes by integrate the similar functions into one function page. Furthermore, in order to deal with the drastic change of node software, we also introduce the program structure into AdvCache, which improves the hit rate of cache.

This research work is supported by the National Natural Science Foundation of China under Grant Nos. 61502427, 61379122, 61502428 and 61379023, and the Zhejiang Provincial Natural Science Foundation under Grant Nos. LY16F020034 and LR16F010003.