EasiCache：一种基于缓存机制的低开销传感器网络代码更新方法

邱杰凡[1),2)] 李 栋[1)] 石海龙[1),2)] 杜文振[1),2)] 崔 莉[1)]

[1)](中国科学院计算技术研究所 北京 100190)

[2)](中国科学院研究生院 北京 100049)

摘 要 随着应用环境越来越复杂多变，传感器网络需要具备远程代码更新的能力，对节点进行灵活地配置和升级以适应环境变化.然而过高的代码更新开销一直困扰着远程代码更新在传感器网络中的大规模应用.代码更新开销主要包括存储代码引起的重组开销和节点通信产生的传输开销.在工程实践中，作者发现重组开销甚至有可能超过传输开销成为主要的更新开销.为此作者提出了一种基于代码缓存机制的低开销远程代码更新方法——EasiCache.该方法通过代码缓存机制在低功耗RAM上动态保存并执行需要频繁更新的代码，尽量避免对高功耗闪存flash进行读写操作，从而有效降低了重组开销.此外，该方法通过函数级代码差异对比，降低了传输代码量，同时保存了代码缓存机制所需的程序结构信息，进一步降低了重组开销.实验结果验证了该方法在降低代码更新开销方面的有效性.

关键词 传感器网络；远程代码更新；代码缓存机制；函数级代码差异对比；物联网

中图法分类号 TP393 **DOI号**：10.3724/SP.J.1016.2012.00555

EasiCache: A Low-Overhead Sensor Network Reprogramming Approach Based on Cache Mechanism

QIU Jie-Fan[1),2)] LI Dong[1)] SHI Hai-Long[1),2)] DU Wen-Zhen[1),2)] CUI Li[1)]

[1)](*Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

[2)](*Graduate University of Chinese Academy of Sciences, Beijing* 100049)

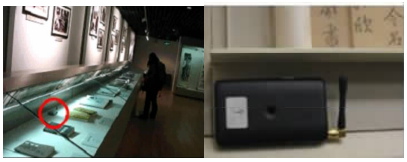
Abstract With applied environment of sensor networks becoming complicated and changeable, the over-air reprogramming is necessary for flexibly configuring and updating sensor nodes. However, too high overhead still restraints the large-scale application of the over-air reprogramming. The over-air reprogramming overhead includes the programming and the transmission overhead. In practice, we find the programming overhead may exceed the transmission overhead and thus provide EasiCache, a novel low-overhead reprogramming approach based on cache mechanism. Using the mechanism, the frequently changed codes are dynamically stored and executed in the low-power RAM instead of the high-power flash, which helping to lower the programming overhead. Additional, the approach uses the different code comparison between functions to reserve the program structure information and reduce the transferred code size. The experiment results demonstrate the EasiCache's effectiveness of lowering the over-air reprogramming overhead.

Keywords sensor network; over-air reprogramming; cache mechanism; different code comparison between functions; Internet of Things

收稿日期：2011-09-09；最终修改稿收到日期：2011-12-28.本课题得到“新一代宽带无线移动通信网”国家科技重大专项项目(2010ZX03006-003-02)、国家自然科学基金项目(61003293)、北京市自然科学基金项目(4112054)和中国科学院计算技术研究所知识创新项目(20106030)资助.**邱杰凡**，男，1984年生，博士研究生，主要研究方向为嵌入式系统、无线传感器网络.E-mail：qiujiefan@ict.ac.cn.**李 栋**(通信作者)，男，1979年生，博士，助理研究员，主要研究方向为无线传感器网络和Ad hoc网络.E-mail：lidong@ict.ac.cn.**石海龙**，男，1986年生，博士研究生，主要研究方向为分布式操作系统和物联网.**杜文振**，男，1989年生，硕士研究生，主要研究方向为无线传感器网络和物联网.**崔 莉**，女，1962年生，博士，研究员，博士生导师，主要研究领域为传感器技术、无线传感器网络和物联网.

1 引 言

在一个大规模无人值守的传感器网络中，受节点自身资源的限制和周围环境变化的影响，开发者很难在开发阶段全面考虑节点在部署后可能遇到的各种突发情况，因此需要通过远程代码更新对节点进行灵活地配置和升级. 以我们在故宫中部署的文物监测传感器网络为例[1]. 传感器节点一旦被部署到展柜中，展柜即被封闭，只有在换展时才能被取出，如图 1(a)所示. 其中一些传感器节点受到人流密度及展柜布置的影响，频繁发送数据，导致电池能量迅速耗尽. 利用远程代码更新，可以根据节点所处环境的不同，动态地调整节点上的休眠机制和数据保存机制，从而有效地延长节点的生命周期. 我们在太湖部署的蓝藻监测打捞感-执系统(Cyber-Physical Systems)[2]也存在类似问题. 感-执节点受波浪及天气影响较大. 特别是阴雨天气，经常发生数据被阻塞的情况，如图 1(b)所示. 利用远程代码更新可以在降雨之前更新路由策略，防止数据阻塞情况的发生. 另一方面，受季节变化的影响，蓝藻爆发的程度不尽相同，通过远程代码更新动态调整蓝藻打捞的调度策略可以提高打捞效率. 此外诸如普度大学的球场监测系统 eStadium[3]、香港科技大学的森林监测系统 GreenOrbs[4]等传感器网络，也都采用了不同的远程代码更新方法处理节点在部署以后遇到的各种突发情况.



(a) 故宫文物监测传感器网络

(b) 太湖蓝藻监测打捞感-执系统

图 1 我们已部署的传感器网络

随着物联网的发展，作为感知前端的传感器网络会越来越多地担负起局部数据处理工作. 在物与物能够智能交流的场景中，传感器节点需要作为信息装置融入到各种物体中，交互执行各种智能算法并产生有效的智能判断[5]. 这不仅要求节点具有数据采集和发送功能，更需要节点能够根据外部需求变化，主动调整自身功能，动态实现各种算法. 因此远程代码更新势必成为物联网中不可或缺的技术.

较高更新开销一直困扰着远程代码更新的大规模应用. 更新开销主要包括两方面：一方面是节点之间发送和接收更新代码时产生的传输开销；另一方面是节点上重建和存储代码时产生的重组开销. 目前大多数远程代码更新方法对如何降低传输开销进行了深入的研究，而对如何降低重组开销则关注较少.

以增量式代码更新方法为例，该类方法通过只传输新旧程序的差异代码，可以有效地降低传输开销[3,6-9]. 该类方法在代码重组时，通常要对外部 flash(由节点板载的扩展 flash 组成，如 TelosB[10]节点默认的扩展 flash 为 STM25P)和嵌入式芯片的内部 flash 进行读写操作. 如表 1 所示，较高的 flash 读写功耗导致这类方法的重组开销超过传输开销. 以我们在故宫中部署的传感器网络为例，在采用增量式代码更新方法后，我们将传感器节点上的程序从 2.2 版升级至 2.3 版需要传输 1560 字节更新代码，通过一个电流检测放大电路测量节点上存储器读写操作的电流、电压和时间，并计算开销. 在这个过程中，传输能量开销为 19.4 mJ，而重组开销竟达到 35.7 mJ，其中读写 flash 产生的开销占到重组开销的 98.2%以上.

表 1 TelosB 节点存储器读写 1000 Bytes 数据的平均开销

读操作	平均开销/μJ	写操作	平均开销/μJ
读外部 flash	1015	写外部 flash*	2458
读内部 flash	785	写内部 flash	1850
读 RAM	<50	写 RAM	126

注：* 写 flash 操作之前需要进行擦除操作，因此写 flash 开销包含擦除 flash 的开销.

从表 1 可知，使用低功耗 RAM 代替高功耗 flash 存储代码可以有效降低重组开销. 但是由于 RAM 空间有限，一般不可能将所有需要更新的代码都放入 RAM 中，只能将一部分需要频繁更新的代码放入，然而如何确定需要频繁更新的代码目前尚未有相关研究. 当前可供参考的做法是由开发者在编程阶段指定放入 RAM 中的代码. 这种做法仍然是以简单的传感器节点作为应用对象，没有考虑在物联网中节点可能由于外界应用需求的变化而频繁地调整自身功能，从而改变最初程序中各部分代码更新的频度：一部分保存在内部 flash 中原本不需要更新的代码开始进行频繁更新，引起大量的读写 flash 操作；而一部分保存在 RAM 中原本需要频繁更新的代码则很长时间不进行更新，却占据

着有限的 RAM 空间. 针对这个问题，我们设计了一种基于代码缓存的低开销远程代码更新方法，EasiCache. 该方法在 RAM 上模拟出一块代码缓存区域保存需要频繁更新的代码. 这种代码缓存同传统意义上的高速缓存(Cache)相似，同样基于代码的局部性原理：通常外部物理世界的需求是渐变的，因此我们假设对代码的修改具有局部性. 即对完成某个功能的若干个函数(function)在一段时间内需要根据外界需求的变化连续地进行更新；不同之处在于，代码缓存机制并不是以提高程序的执行效率为目的，而是利用 flash 与 RAM 阶梯式的读写功耗差异，动态地使用低功耗的 RAM 保存并执行需要频繁更新的代码，避免对高功耗的 flash 进行读写操作，从而有效降低重组开销.

另一方面，EasiCache 作为一种增量式代码更新方法，采用了函数级代码差异对比技术. 该技术用于计算新旧程序间差异代码，通过传输这些差异代码可以有效降低代码更新过程中的传输开销；同时，该技术可以保留代码缓存机制所需的程序结构信息. 利用这些信息我们设计了重组操作，可以减少代码重建次数，进一步降低重组开销.

本文的贡献主要集中在以下几个方面：

(1) 为了降低重组开销，提出一种代码缓存机制，可以有效避免对高功耗 flash 的读写操作，并且设计了相应的替换算法.

(2) 为了降低传输开销，提出了一种与代码缓存机制相匹配的函数级代码差异对比技术. 它在减少更新代码传输量的同时，可以保留程序结构信息. 利用这些信息，我们定义了 3 种重组操作，进一步降低了重组开销.

(3) 为了验证 EasiCache 的有效性，我们设计了单次更新实验和连续更新实验，以验证 EasiCache 在降低更新开销方面的有效性.

本文第 2 节简要介绍研究背景；第 3 节给出 EasiCache 的组成结构与执行流程；第 4 节介绍 EasiCache 的实现；第 5 节设计实验更新场景并分析实验结果；第 6 节介绍相关工作；最后在第 7 节给出结论和未来的工作.

2　研究背景

我们的研究以 TinyOS 及 nesC 语言为基础，并以 TelosB 节点[10]作为硬件实现平台. nesC 语言是一种扩展的 C 语言，专门针对传感器网络特点进行了优化. 以 nesC 语言编写的程序能够被 nesC 编译器编译，并最终生成嵌入式芯片可执行的文件. 以采用嵌入式芯片 MSP430 的 TelosB 节点为例，图 2 显示了使用 nesC 语言编写的.nc 文件如何生成 MSP430 可执行的.ihex 文件. 由于.nc 文件仅仅包含了开发者的应用程序，如果直接比较新旧程序的.nc 文件并不能反映 TinyOS 系统模块的变化. .nc 文件对应生成的 C 语言文件 app.c 是 TinyOS 镜像(image)文件，它包含了整个 TinyOS 操作系统以及上层应用程序，但是由于很难确定单条 C 语言代码可能生成的指令类型以及指令条数，通常无法计算代码的修改地址. 在 EasiCache 中，将保存有程序结构信息的汇编文件(.s)作为生成差异代码的比较对象. 而最终生成的.ihex 文件将作为传输对象，它可以被嵌入式芯片直接执行.

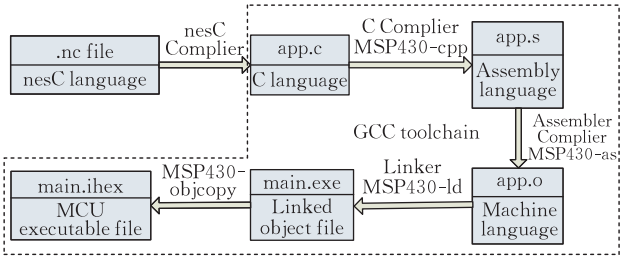


图 2　.ihex 文件生成流程图

当前大多数传感器节点的嵌入式芯片都采用了 flash＋RAM 的存储结构. 以 TelosB 节点为例，MSP430 的内部 flash 与 RAM 统一编址，表 2 显示了一种典型的 MSP430 存储地址空间①.

表 2　MSP430F1611 地址空间

硬件结构	地址空间
内部 flash (48KB)	中断向量表(32B) 地址：0xFFE0-0xFFFF 代码空间(48KB) 地址：0x4000-0xFFFF
RAM (10KB)	扩展 RAM (8KB) 0x1900-0x38FF 对映 RAM (2KB) 0x1100-0x18FF
信息存储器 (information memory)	256 Bytes 0x1000-0x10FF
引导存储器 (boot memory)	1KB 0xFFF-0xC00
RAM(对映 RAM)	2KB 0x9FF-0200h
外设	512 Bytes 0x000-0x1FF

① MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller [Online]. Available: http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf

默认情况下，.ihex 文件中的代码段(.text 段)被放入 0x4000 到 0xFFFF 的内部 flash 中，全局变量段(包括.data 段和.bss 段)被放入 0x1100 到 0x38FF 的 RAM 中.如果代码段大小超出内部 flash 容量，则需要将代码放入外部 flash 中.

3 EasiCache 概述

图 3 给出了 EasiCache 的组成结构，它主要由在计算机端执行的生成更新脚本(delta script)以及节点上运行的代码缓存机制两部分组成.生成更新脚本主要在计算机端实现，包括计算函数级的差异代码以及定义 3 种重组操作.通过传输包含差异代码及重组操作的更新脚本，EasiCache 实现了增量式代码更新.

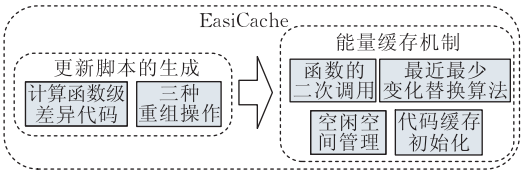


图 3 EasiCache 组成结构框图

EasiCache 中的代码缓存机制运行在节点上，由代码缓存初始化、函数的二次调用、空闲空间管理以及替换算法 4 部分组成.代码缓存初始化使开发者有机会在编程阶段预先将代码以函数为单位放入代码缓存中.函数的二次调用保证了函数在重新定位后仍能够被正确地调用.空闲空间管理则是为了提高存储空间利用率，通过空闲空间列表重新释放代码重复占用的存储空间.替换算法主要考虑当代码缓存耗尽时，采用何种策略选择回写到内部 flash 中的函数.

节点部署之前，开发者通常可以预见部分需要频繁更新的代码.通过代码缓存初始化将这部分代码预先保存在代码缓存中.图 4 给出了在节点部署之后，对节点进行代码更新的执行流程.首先在计算机端确定需要更新的差异函数，计算新旧函数间的差异代码并生成相应的重组操作.随后计算机通过串口将包含函数差异代码与重组操作的函数更新脚本(delta script)后发往网关节点.网关节点收到更新脚本之后，以无线多跳(multi-hop)方式将更新脚本发往待更新节点.

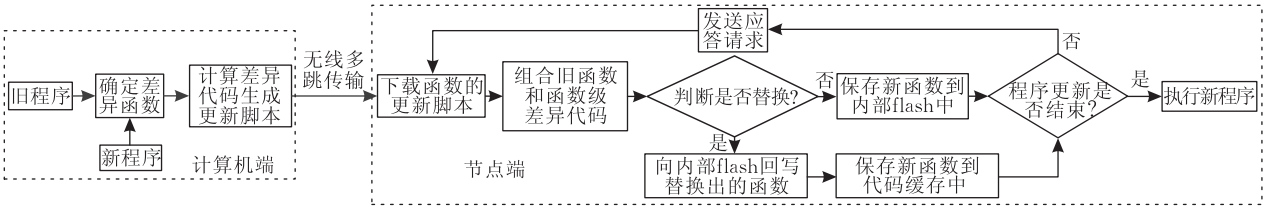


图 4 EasiCache 执行流程图

待更新节点将更新脚本下载到 RAM 中的函数组装区(function assembler)，并根据更新脚本中的重组操作，将差异代码与旧函数代码进行组合后生成新函数的代码.随后使用替换算法判断当前新生成的函数是否需要保存在代码缓存中.如果需要，则将替换出的若干函数回写至内部 flash；否则，新函数直接写入内部 flash.最后节点发送携带有当前新函数奇偶校验码的应答消息，请求继续更新函数.如果没有函数再需要更新，则整个程序的代码更新过程结束，节点开始执行新程序.

4 EasiCache 的设计与实现

本节主要由 4.1 节介绍的生成更新脚本和 4.2 节介绍的代码缓存机制两部分组成.前者主要解决传输开销过高的问题，后者主要解决重组开销过高的问题.

4.1 更新脚本的生成

在计算差异代码之前，首先确定哪些函数需要更新.EasiCache 采用二级比较机制，即通过比对新旧代码的奇偶校验码和 MD4 码的方式确定需要更新的函数.

首先对旧程序中的所有函数分别生成奇偶校验码以及 MD4 码，并对旧程序本身再生成一次奇偶校验码和 MD4 码.由于生成 MD4 码的计算开销较大，当获得新程序的代码后，并不立即生成新程序的 MD4 码，而是首先生成新程序的奇偶校验码.对新旧程序的奇偶校验码进行比较，如果不同，直接开始比较新旧程序中的各个函数；否则计算新程序的 MD4 码，继续与旧程序的 MD4 码进行比较.

新旧函数的比较与新旧程序的比较类似.首先将新程序中每个函数的奇偶校验码与旧程序中对应

函数的奇偶校验码进行比较，如果不同，则确定需要更新的函数；然后对所有奇偶校验码相同的函数分别生成 MD4 码，继续比较.

在确定需要更新的函数之后，可以直接计算新旧函数之间的差异代码. 我们使用 Python 语言编写计算差异代码的程序. 输入为新旧程序的汇编文件(app.s)；输出为汇编语言组成的差异代码文件(Diff.s). 差异代码文件(Diff.s)中包括：以函数为单位的新旧程序的差异代码，差异代码距离函数第一条指令的指令偏移数以及需要进行的重组操作. Diff.s 文件内容如图 5 所示.

```
main: call    #__nesc_atomic_start    /* 4_Rep */
      mov.b   r15, @r4                /* 5_Del */
      call    #Scheduler__init        /* 17_Ins */
      call    #PlatformInit__init     /* 29_Ins */
                    …
```

图 5　Diff.s 中的部分差异代码

/* */中的数字表示相对于函数第一条指令的指令偏移数. 由于每一条汇编指令对应的机器指令长度是一定的，所以在得到指令偏移数之后，可以计算得到差异代码相对于函数起始地址的偏移量. Del、Ins 和 Rep 分别表示 3 重组操作：删除操作、插入操作和替换操作. 替换操作不会改变函数的大小，可以直接写入新代码覆盖旧代码. 插入操作或删除操作会改变旧函数的大小. 由于保存了程序的结构信息，当一个函数的更新仅仅涉及替换操作或删除操作时，可以通过写入无条件跳转指令(JMP)或者空指令(NULL)来实现删除操作，并通过直接覆盖原始代码实现替换操作，避免函数的代码重建，有效降低重组开销. 如果对函数的更新包含了插入操作，则需要对这个函数进行重建.

如图 2 所示，Diff.s 文件中的差异代码经过链接器(MSP430-ld)链接和嵌入式代码格式转换器(MSP430-objcopy)最终生成嵌入式芯片可执行的代码(.ihex 文件). 这些可执行的代码、重组操作以及修改地址组成了代码更新所需的更新脚本. 另外，需要特别指出的是由于上述生成更新脚本的过程全部由计算机完成，不会消耗传感器节点的能量，也不会给传感器网络增添额外的开销.

4.2　代码缓存机制

由于读写 RAM 与 flash 的功耗差异巨大，EasiCache 通过代码缓存机制将需要频繁进行更新的代码保存在 RAM 中，减少对 flash 的读写操作.

4.2.1　代码缓存初始化

开发者可以在编程阶段将特定函数放入特殊段，并通过链接器(MSP430-ld)设置特殊段的起始地址①. 由于 MSP430 的 RAM 与 flash 统一编址，通过设定特殊段的地址，可以将程序的部分函数保存在 RAM 中并执行. 使用属性(attribute)对需要放入特殊段的函数进行标记.

图 6 中显示了两个被标记的函数，它们被指定存放在代码缓存区域(.cache 段)中，而不是默认的.text 段. 另外如果开发者可以预见到若干全局变量未来可能需要频繁修改，也可以使用相同的方法，将这些全局变量分别放入.dataram 段(初始化全局变量段)或者.bssram 段(未初始化全局变量段)，再通过设定这两个特殊段的地址将它们放入 RAM 中.

```
__attribute__(section(".Cache")) static Boot__booted (void)
__attribute__(section(".Cache")) static Timer0__fired (void)
```

图 6　使用属性(attribute)标记代码

由于 RAM 中默认存放了.bss 段和.data 段，为了避免.bssram 段、.dataram 段、代码缓存区域(.cache 段)、函数组装区域(Function Assembler)以及管理列表区域(Management List)与 RAM 默认存放的段发生地址冲突，需要进行两次编译. 在第一次编译时，设定所有区域和段位于相距较远的地址上. 由于代码缓存区域、函数组装区域以及管理列表区域的大小是我们预先设定的，在进行一次编译后，只需要计算.bas 段、.data 段、.bssram 段以及.dataram 段的大小，确定所有段和区域的最终地址后，再进行一次编译.

经过代码缓存初始化后，内部 flash、外部 flash 和 RAM 的分配情况如图 7 所示. RAM 中地址自低到高依次存放了.data 段、.bss 段、.dataram 段、.bssram 段、函数组装区域、管理列表区域、代码缓存区域(.cache 段)以及系统堆栈.

4.2.2　函数的二次调用

执行删除或者插入操作之后会导致函数代码量发生变化. 特别是执行插入操作后，可能会增加函数的代码量. 由于函数是连续存放的，如果将代码量增加后的新函数仍保存到旧函数地址上，势必会覆盖与它相邻的函数，严重时会导致整个程序崩溃，所以必须将新函数转存到其它地址上. 转存之后，这个函数的入口地址(entry address)发生改变，

① GNU Binutils: loader and linker[Online]. Available: http://sourceware.org/binutils/docs-2.21/ld/index.html

意味着对这个函数进行调用的所有指令都需要进行更新，而更新这些指令可能会引起较大的更新开销.

为此需要使用函数的二次调用解决这个问题：调用函数时，并不是直接对函数进行调用，而是先跳转到指令数据混合列表(Instrument and Date List)，由指令数据混合列表中的函数调用指令调用真正的函数，调用过程如图 7 所示.

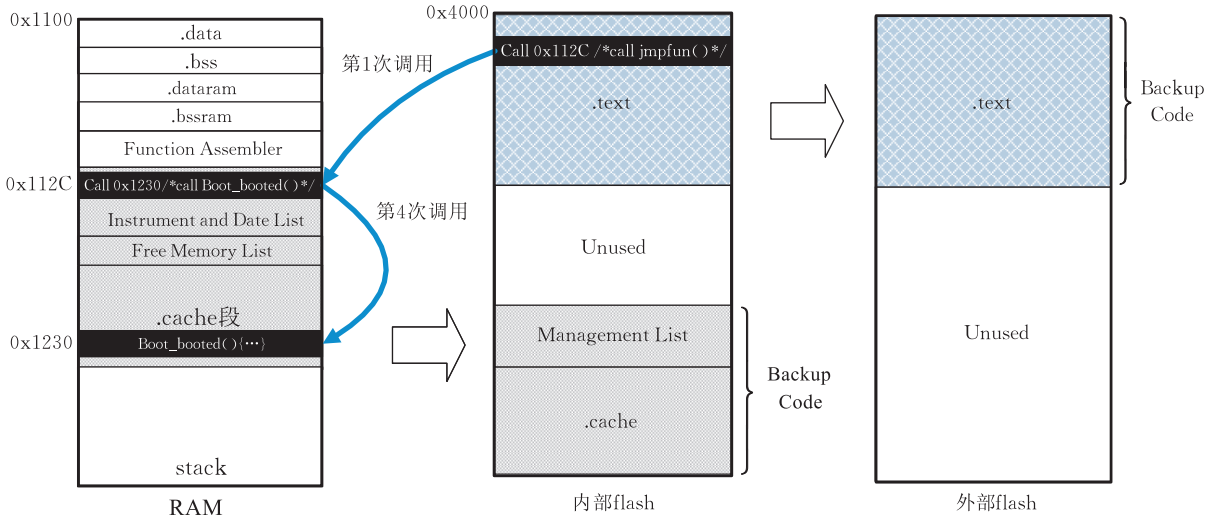


图 7 代码存储示意图及函数二次调用示意图. 函数二次调用：保存在代码缓存中的函数 Boot_booted 地址为 0x1230. 当保存在内部 flash 上的某条指令要调用 Boot_booted 函数时，需要先跳转到指令数据混合列表中的调用指令(位于 0x112C 的地址上)，其中包含了 Boot_booted 的真实地址. 再由这条调用指令调用保存在缓存区域中的 Boot_booted 函数.

如图 8 所示，指令数据混合列表由函数信息数据以及对函数的调用指令组成. 嵌入式芯片存储资源的稀缺性(例如 MSP430 的存储空间小于 64 KB)要求我们必须尽量复用存储空间. 函数调用指令包含了函数当前的存储地址，因此将它与函数的信息数据进行空间复用. 函数的信息数据包含了函数编号、函数代码量以及更新次数. 更新次数指示当前函数已被更新的次数，同时通过正负数区分当前函数是否已被保存在代码缓存中(正数表示当前函数保存在内部 flash 中，负数表示保存在代码缓存中).

函数编号	函数的尺寸/B	更新次数	调用指令	函数的入口地址	返回指令
31	66	−1	call	0x1560	ret
48	26	3	call	0x4a18	ret
126	26	−2	call	0x5160	ret

图 8 指令数据混合列表

指令数据混合列表保存在管理列表区域中. 列表中每条函数调用指令的位置是固定的，如图 7 中，在指令数据混合列表中调用 Boot_booted 函数的指令地址固定为 0x112C. 当某个函数的入口地址发生了变化，只需要修改指令数据混合列表中调用这个函数的指令，而不再需要修改其他对这个函数的调用指令. 函数的二次调用实际上是一种对函数地址的集中式管理，只需要修改指令数据混合列表中的一条调用指令，就可以完成对所有调用指令的修改. 函数的二次调用相当于在调用函数时增加一条跳转指令. 在实验部分的第 5.3 节，我们将分析增加一次跳转对程序执行效率产生的影响.

4.2.3 空闲空间的管理

函数代码量在更新后有可能增加，这时需要将更新后的函数放入新的地址. 如果直接将新函数放入代码缓存或者内部 flash 的末尾空间，会造成嵌入式芯片上的存储资源被迅速耗尽. 为了节约有限的存储资源，当新函数被放入代码缓存或者回写到内部 flash 中时，可以将原函数所占用的空间重新释放，保存其它新函数.

我们设计了缓存空闲空间列表(Free Cache List)和 Flash 空闲空间列表(Free Flash List)管理空闲空间，并将这两个列表保存在管理列表区域. 如图 9 所示，缓存空闲空间列表和 Flash 空闲空间列表分别记录了代码缓存和内部 flash 中空闲空间的起始地址及大小. 当新函数需要重新选择地址保存时，可以通过空闲空间列表查找合适的存储空间. 转存之后，需要修改空闲空间列表对应的项，并将对应的原始函数的起始地址和函数大小记录到空闲空间列表，释放对应原函数的存储空间.

缓存空闲空间起始地址	空闲空间的大小/B
0x1500	8
0x1542	14
…	…
0x168A	86

(a) 缓存空闲空间列表

Flash 空闲空间起始地址	空闲空间的大小/B
0x4160	89
0x5272	32
…	…
0xB878	22640

(b) Flash 空闲空间列表

图 9　空闲空间列表(列表的最后一项记录了末尾空间起始地址和空间大小)

4.2.4　代码缓存的替换

节点接收到函数更新脚本后,根据指令数据混合列表中函数更新次数的正负,确定旧程序中相应的函数是否已经被保存到代码缓存中.如果代码缓存已经保存了这个需要更新的函数,称之为命中.将需要更新的旧函数代码与差异代码一起放入函数组装区域进行组合,重建新函数.

代码缓存区域的容量通常在节点部署之前确定.其确定原则是不能够影响 RAM 中.bss 段、.data 段以及系统堆栈正常使用.如果代码缓存即将被耗尽,而重建函数的代码量超出了代码缓存的剩余容量,则需要将保存在缓存中的部分函数回写到内部 flash 中,腾出空间存放新函数,这个过程称为替换.

传统缓存代码替换算法通常关注保存在缓存中的代码是否被执行.如:最近最少使用(Least Recently Used,LRU)算法以代码被执行的次数作为是否替换的主要因素.在 EasiCache 中,缓存代码的替换以函数为单位进行,如果仅仅以函数被更新次数为替换的主要因素,有可能出现这样一种情况:函数 F_m 每次升级都需要更新,命中率较高,但函数 F_m 代码变化较小;而函数 F_n 被更新次数较少,命中率较低,但发生更新时代码变化较大.这时有可能将函数 F_n 替换出去,一旦对函数 F_n 进行更新,仍然需要对 flash 进行大量的读写操作.为此我们设计了最近最少变化算法(Least Recently Changed,LRC)将函数变化次数以及函数变化的程度同时考虑.算法中的相关参数见表 3.

表 3　参数表

参数名	描述
P_{old}/P_{new}	更新前的旧程序/更新后的新程序.
$F_{(old,i)}/F_{(new,i)}$	旧程序/新程序中的函数 F_i.
CRF_i	函数 F_i 的变化率.
C_k	在第 k 次升级时,函数已被更新的次数.
$Rf_{(k,i)}$	函数 F_i 的替换因子.决定在第 k 次升级时是否需要对函数 F_i 进行替换.
a	比例因子,调节函数变化次数和函数变化程度对替换因子的影响.

首先定义反映函数 F_i 变化情况的变化率 CRF_i:

$$CRF_i=\frac{SizeofChag(F_{(old,i)},F_{(new,i)})}{SizeofChag(P_{old},P_{new})},\ i=1,2,3,\cdots,n \tag{1}$$

函数 $SizeofChag(x,y)$ 计算 x 更新到 y 发生改变的代码总量.变化率 CRF_i 越大说明函数 F_i 的变化程度越高,相对于其它函数对整体程序的更新贡献越大.

在第 k 次升级时,最近最少变化(LRC)算法将函数被更新的次数和函数变化的程度统一考虑,并计算函数在第 k 次升级时的替换因子:

$$Rf_{(k,i)}=C_k^{\alpha}\cdot CRF_i^{(1-\alpha)},\ \alpha\in[1,0] \tag{2}$$

如果需要更新的函数 F_i 没有保存在代码缓存中,且它的替换因子 $Rf_{(k,i)}$ 大于已经保存在代码缓存中的若干个函数的 $Rf_{(k,j)}$,则需要计算替换出这若干个函数后腾出的缓存空间是否足够保存函数 F_i.如果可以,则将这若干个函数回写入内部 flash 中,并将函数 F_i 保存在缓存中.反之如果仍然没有足够的缓存空间,则将函数 F_i 写入内部 flash 中.LRC 算法的核心思想是对不经常发生变化或者变化较小的函数进行替换,而尽量将经常变化或者变化较大的函数保存在代码缓存中.同时为了能够适应不同更新,可以通过合理设置比例因子 a,调节函数被更新次数与函数变化程度对替换因子的影响.

4.2.5　代码缓存的安全性

由于代码缓存使用易挥发性的 RAM 保存代码,当节点断电重启时可能导致 RAM 中的关键代码丢失,严重时会使程序崩溃,因此需要通过备份代码来消除潜在的程序安全隐患.如图 7 所示,我们将 RAM 中代码缓存和管理列表的内容备份到内部 flash 中,将内部 flash 中.text 段的内容备份到外部 flash 中.当节点重启之后首先将外部 flash 中的.text 段内容读入内部 flash,然后将内部 flash 中代码缓存和管理列表的内容读入 RAM 中,完成程序恢复.

为了尽量避免对 flash 的读写操作,并不是每一次升级都进行代码备份.只有当程序需要进行重大升级时,才进行代码备份,并记录备份程序的版本号.当节点完成程序恢复后,如果发现当前程序的版本号过低,可以要求计算机端重新发送最新程序的更新脚本,并更新至最新的程序.

5 实验结果与分析

我们设计的实验包括单次更新和连续更新两种.单次更新实验测试 EasiCache 的传输开销以及对程序的某个特定部分进行更新的重组开销.连续更新实验测试代码缓存机制的有效性.在现有代码更新方法中,由于不存在代码缓存机制,因此通常只进行单次更新实验.而我们为了更好地验证代码缓存机制的有效性,把实验重点放在对连续更新场景的测试上.最后我们通过实验分析 EasiCache 对程序执行效率的影响.

5.1 单次更新实验

我们在单次更新时,设计了 6 种场景,包括:

更新 1. 修改程序 Blink 中的全局变量,改变 LED 的闪烁频率.

更新 2. 修改程序 Blink 中的单条指令,关闭一个 LED.

更新 3. 在程序 Blink 中删除函数 Blink_Boot_booted 中的连续两行代码,同时删除函数 Blink_Timer0_fired 中的一行代码.

更新 4. 在程序 Blink 的函数 BlinkC_Timer0_fired 中插入两行代码.

更新 5. 在程序 Blink 中插入一个控制 LED 闪烁模式的函数 Control_LED_Pattern.

更新 6. 将程序 Blink 更新为程序 CntToLed.程序 CntToLed 通过 3 个 LED 显示计数变量 counter 最后 3 位的值.

在单次更新实验中,我们将 EasiCache 与当前的远程代码更新方法 Deluge[11]、Elon[12] 以及 Hermes[7] 进行比较.其中 Deluge 是 TinyOS 标准远程代码更新方法.Elon 通过设定 TinyOS 的组件(component)为可替换(replaceable),在编程阶段将可能频繁更新的代码保存在 RAM 中并执行.Hermes 基于比特级差异对比技术,有效降低了传输代码量.实验结果显示 EasiCache 在大多数情况下优势明显.

表 4 显示了在更新 1～6 中 Deluge、Elon 以及 EasiCache 需要传输的代码量.观察发现 Deluge 的传输开销十分巨大.这是由于在使用 Deluge 进行更新时要传输整个 TinyOS 镜像(image)以及代码更新协议.以更新 1 为例,当使用 Deluge 进行代码更新时,传输的 Blink 镜像本身仅占总传输代码量的 11.5%,而真正要更新的代码只占总传输代码量的 0.043%.Elon 和 EasiCache 都可以单独对变量进行修改[12],因此两种方法在更新 1 中的传输开销很低.在更新 2～4 中和更新 6 中,Elon 没有采用代码差异比较,当新旧程序中的若干个函数不同时,需要传输整个函数.而 EasiCache 通过函数级代码差异对比只需要传输差异代码,因此传输代码量明显少于 Elon.在更新 5 中,整个函数被插入旧程序中.新旧程序的差异是整个函数,因此 EasiCache 与 Elon 传输代码量相同.

表 4 各种更新场景下代码更新所需传输的代码量

(单位:Byte)

方法	代码量					
	更新 1	更新 2	更新 3	更新 4	更新 5	更新 6
Deluge	23110	23110	23114	23116	23296	25008
Elon	8	58	62	42	86	156
EasiCache	8	14	16	20	86	132

表 5 显示了在更新 1～6 中,Hermes、Elon 以及 EasiCache 的重组开销.3 种更新方法都可以独立对存储在 RAM 中的全局变量进行写改,在更新 1 中重组开销基本相同.Hermes 在更新代码时,会首先将更新脚本存入外部 flash,然后将重建后的新程序回写到嵌入式芯片的内部 flash.这样做导致重组效率低,重组开销大.由于 Elon 采用了直接把保存在 RAM 上的函数进行整体替换的策略,重组开销主要来自于向 RAM 写入函数代码,因此重组开销较小.EasiCache 通过删除操作和替换操作可以在不重建代码的情况下直接对函数完成更新,因此在更新 2 和更新 3 中重组开销明显少于 Elon.在更新 4 和更新 6 中,EasiCache 执行插入操作,需要重建函数代码,因此重组开销与 Elon 相差不大.而在更新 5 中由于代码差异主要是整个函数,函数级差异对比对降低重组开销效果不明显,重组开销与 Elon 基本相同.

表 5 各种更新场景下代码更新所需的重组开销

(单位:μJ)

方法	重组开销					
	更新 1	更新 2	更新 3	更新 4	更新 5	更新 6
Hermes	0.035	8027.1	8157.2	8144.6	8256.7	12490.5
Elon	0.039	40.2	57.7	34.8	6.47	96.8
EasiCache	0.021	0.14	0.36	36.2	6.92	88.5

在单次更新的场景中,开发者通过缓存初始化可以将需要更新的函数在编程阶段预先保存在 RAM 中.如果设置合理,可以完全避免对 flash 的读写操作,因此重组开销较低.然而在一个连续更新的场景中,开发者不可能完全预测所有需要更新的函数,由此可能导致对内部 flash 进行大量读写操

作，增加重组开销，因此需要使用代码缓存机制动态保存更新频繁的函数.

5.2　连续更新实验

为了验证代码缓存机制的有效性，我们设计了一个连续更新的场景，即当节点的代码完成更新之后，立即再一次更新节点上的程序. 如图 10 所示，我们选取 4 个 TinyOS 的例子程序：RadioCnt、RadCntToLeds、BaseStation 和 BaseStation15.4 以及我们在故宫传感器网络节点上的程序：EasiRouter2.2 和EasiRouter2.3 作为实验更新程序. 程序 RadioCnt 与程序 RadCntToLeds 类似，后者加入了通过 LED 显示计数变量 counter 最后三位的功能. 程序 BaseStation15.4 与程序 BaseStation 都具有监测无线信道数据并向串口转发的功能，后者增加了接收串口数据以及通过无线信道发送数据的功能. 程序 EasiRouter2.3 最为复杂，加入了实用的路由机制. 程序 EasiRouter2.3 是程序 EasiRouter2.2 的升级版本，不同之处在于前者拥有更完善的休眠重启机制.

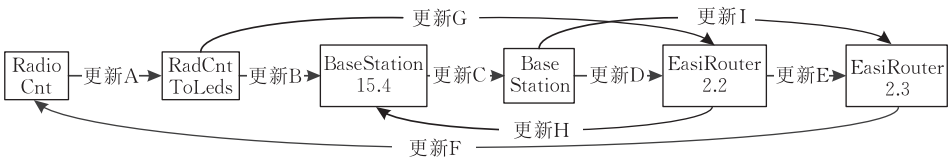


图 10　连续更新流程图

按照图 10 所示的 Order(A,B,C,D,E,F,A′,G,H,C′,I)顺序对节点上的程序进行更新. 我们定义缓存的命中率为命中函数的代码量与缓存容量之比. 默认设定缓存容量为 3 KB，比例因子 $\alpha=0.5$. 图 11 给出了在这个连续更新过程中，采用最近最少使用(LRU)替换算法和最近最少变化(LRC)替换算法的代码缓存命中率.

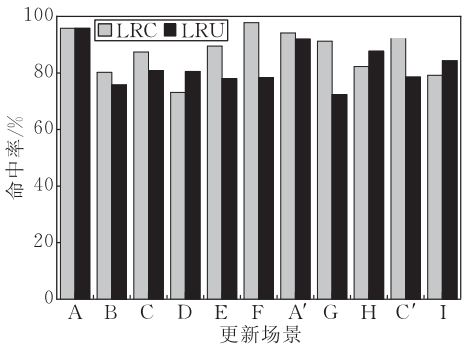


图 11　采用最近最少使用(LRU)算法和最近最少变化(LRC)算法的缓存命中率

节点上保存的起始程序是 RadioCnt. 由于程序 RadioCnt 与程序 RadCntToLeds 较为相似，更新 A 缓存命中率达到了较高的 95.2%. 这也说明通过代码缓存初始化合理地安排缓存中的代码，可以有效提高缓存命中率. 在更新 A 之前没有替换行为的发生，所以两种替换算法的命中率相同. 在更新 A 之后，缓存已经无法容纳所有需要更新的函数，部分函数需要被替换出并回写到内部 flash 中.

程序 RadCntToLeds 与程序 BaseStation 相似度较低，导致更新 B 的命中率明显下降. 程序 BaseStation 与程序 BaseStation15.4 较为相似，差异在于增加了串口和射频发送数据的功能，因此更新 C 的命中率相较于 B 有所回升，但仍然偏低. 从更新 B 和更新 C 可以看出，LRC 算法的命中率明显高于 LRU 算法的命中率，这是由于 LRC 算法能够将那些变化比较大且更新频繁的函数尽量保存在代码缓存中，而 LRU 算法只将函数被更新次数作为唯一的考虑因素. 在更新 D 中，程序 BaseStation 与程序 EasiRouter2.2 差异较大，需要更新的函数较多，命中率下降到 80%以下. 在更新 D 中，LRU 算法的命中率高于 LRC 算法. 这是由于 LRC 算法在上一次更新中(更新 C)替换出了若干更新比较频繁但是变化程度较小的函数(约占缓存总量的 8.1%)，而在更新 D 中这些函数成为了被更新的对象，导致 LRC 算法的命中率降到了 73.14%. 这种情况同样发生在更新 H 和更新 I 中. 在更新 E～G 和更新 C′中，LRC 算法的命中率普遍较高. 特别是在更新 G 中，LRU 算法的命中率下降到了 72.39%，而 LRC 算法的命中率仍然保持在 85%以上. 总体而言，LRC 算法的平均命中率较高，达到了 87.5%；LRU 算法的平均命中率为 82.2%.

表 6 列出了使用 LRC 算法($\alpha=0.5/\alpha=0.95$)和 LRU 算法对上述 6 个程序分别进行 20 次、30 次、45 次、60 次、100 次以及 150 次连续 随机更新的平均命中率(每次测试均重复 50 遍，取平均值). 当 $\alpha=0.5$ 时，除了在连续更新 20 次时，LRC 算法的平均命中率略低之外；在大多数情况下，LRC 算法的平均命中率均高于 LRU 算法. 当 $\alpha=0.95$ 时，LRC 算法的命中率已与 LRU 算法的命中率相差不大. 由式(2)可知，当 α 趋近于 1 时表示函数更新次数

C_k将在越来越大的程度上决定替换因子$Rf_{(k,i)}$，当$\alpha=1$时，LRC 替换算法与 LRU 替换算法等价.

表 6 LRC 算法与 LRU 算法的平均命中率

（单位：%）

算法	命中率					
	20 次	30 次	45 次	60 次	100 次	150 次
LRC($\alpha=0.5$)	84.41	85.60	85.25	88.79	84.18	86.54
LRC($\alpha=0.95$)	83.34	81.22	80.23	80.15	78.14	81.37
LRU	84.71	82.52	80.44	80.52	79.44	81.23

表 7 和表 8 列出了 EasiCache、Hermes[7] 和 Tiny Module-Link[13] 3 种更新方法按照 Order 更新过程中对 flash 进行读写操作的情况. Hermes 的重组过程包括将更新脚本写入外部 flash，与保存在外部 flash 的原始程序进行组合后产生新程序，最后将新程序的代码整体从外部 flash 读出并写入内部 flash. 实际上新程序的代码量决定了 Hermes 的重组开销，导致即使新程序相较于原始程序变化较小，也有可能引起较大的编程开销. 如在更新 F 中，程序 EasiRouter2.3 相较于程序 RadioCnt 功能十分完善，从更新脚本的尺寸上反映了对程序 EasiRouter2.3 进行较小的修改就可以完成更新，但是将新程序 RadioCnt 从外部 flash 搬移到内部 flash 仍然使更新 F 的重组开销维持在一个较高的水平. 另一种更新方法 Tiny Module-Link 充分考虑了这种代码搬移造成的高开销问题，提出将代码组合过程放入低功耗的 RAM 中进行，外部 flash 仅存放更新脚本[13]. 更新时需要将更新脚本读出与保存在内部 flash 中的原始代码进行组合. 同时由于 Tiny Module-Link 也是以函数为单位进行更新，所以更新时仅需从内部 flash 中读取需要更新的函数代码.

表 7 完成更新需要从 flash 读取的代码量

（单位：Byte）

方法		代码量								
		更新 A	更新 B	更新 C	更新 D	更新 E	更新 F	更新 G	更新 H	更新 I
Hermes	读外部 flash	13148	15884	16826	18192	19250	12026	18192	15884	19250
	读内部 flash	—	—	—	—	—	—	—	—	—
Tiny Module-Link	读外部 flash	1428	1392	1986	2814	2518	890	4410	2522	5014
	读内部 flash	2028	4958	1536	3782	1278	636	3152	1798	3122
EasiCache	读外部 flash	—	—	—	—	—	—	—	—	—
	读内部 flash	134	0	272	1266	466	0	1678	1078	2142

表 8 完成更新需要写入 flash 的代码量

（单位：Byte）

方法		代码量								
		更新 A	更新 B	更新 C	更新 D	更新 E	更新 F	更新 G	更新 H	更新 I
Hermes	写外部 flash	1562	4614	2010	2946	1596	926	4528	2640	4872
	写内部 flash	13148	15884	16826	18192	19250	12026	18192	15884	19250
Tiny Module-Link	写外部 flash	1428	1392	1986	2814	1890	890	4410	2522	5014
	写内部 flash	1944	6862	2478	5148	3044	426	6674	3846	7134
EasiCache	写外部 flash	—	—	—	—	—	—	—	—	—
	写内部 flash	578	2484	674	1768	1326	26	2426	2588	4242

EasiCache 为了进一步降低重组开销，使用低功耗 RAM 动态保存并执行部分需要频繁更新的函数，可以有效减少对内部 flash 的读写操作. 同时由于 EasiCache 使用 RAM 保存更新脚本，避免了对外部 flash 进行读写的操作，因此表 7 和表 8 中未列出 EasiCache 对外部 flash 进行读写操作的代码量. Hermes 不需要对内部 flash 进行读操作，故也未列出.

在更新 B 和更新 F 中，由于 EasiCache 的代码缓存中保存了所有需要更新的函数，因此不再需要向内部 flash 读取这些函数的原始代码，对内部 flash 的读取量降为 0. 而在其他更新中 EasiCache 读取内部 flash 的代码量也远远小于 Tiny Module-Link. 另外，EasiCache 可以在 RAM 上直接执行函数代码，不再像 Tiny Module-Link 需要将所有更新的函数代码回写到内部 flash 中，因此写入内部 flash 的代码量也明显少于 Tiny Module-Link.

在表 9 中列出了 Hermes 及 Tiny Module-Link 与 EasiCache 的重组开销之比. 可以看出 EasiCache 的重组开销明显小于另外两种更新方法. 例如在更新 F 中，Hermes 和 Tiny Module-Link 的重组开销分别达到了 EasiCache 的 763.3 倍和 91.03 倍. 这也表明在 EasiCache 中，程序更新代码量成为影响重组开销的主要因素. 当更新代码量增大时，EasiCache 的重组开销也随之增加. 例如在更新 I 中，新生成函数的代码量远远超出了代码缓存能够容纳的极限，只能向内部 flash 回写大量代码，这两种更新方法与 EasiCache 的重组开销之比下降到 6.7 和 3.29.

表 9　Hermes 及 Tiny Module-Link 与 EasiCache 的重组开销之比　　（单位：μJ）

方法	重组开销								
	更新 A	更新 B	更新 C	更新 D	更新 E	更新 F	更新 G	更新 H	更新 I
Hermes：EasiCache	35.33	12.37	34.92	13.20	20.96	763.6	10.21	8.84	6.70
Tiny Module-Link：EasiCache	8.64	4.66	8.33	5.24	5.61	91.03	4.87	2.94	3.29

5.3　EasiCache 对执行效率的影响

对函数进行二次调用处理后，每次函数调用多进行一次跳转，增加了程序的复杂度，会对程序执行效率产生影响.以 TelosB 节点采用的 MSP430F1611 为例.增加一次函数跳转意味着每次函数调用要多执行一次调用指令（CALL）和一次返回指令（RET）.

图 12 给出了 6 个更新程序在采用函数的二次调用后，执行效率受到的影响.受影响最大的是程序 BaseStation15.4 和程序 BaseStation，执行效率分别下降了 9.5%和 12.3%.由于这两个程序的大量工作是串口通信和射频通信，在多数时间里需要对与底层量硬件相关的函数进行调用，函数的切换较为频繁，而且这部分函数的代码量较少，执行时间较短，因此 CALL 指令和 RET 指令所消耗的时钟周期占程序执行总的时钟周期比重较大.当增加一次函数跳转后，执行效率受到的影响也相对较大.我们在故宫传感器网络节点上运行的程序需要经常进入休眠状态并重启，对与底层硬件相关的函数调用也较多，相对于另外两个没有休眠机制的程序 RadioCnt 和程序 RadCntToLeds 执行效率受影响较大.

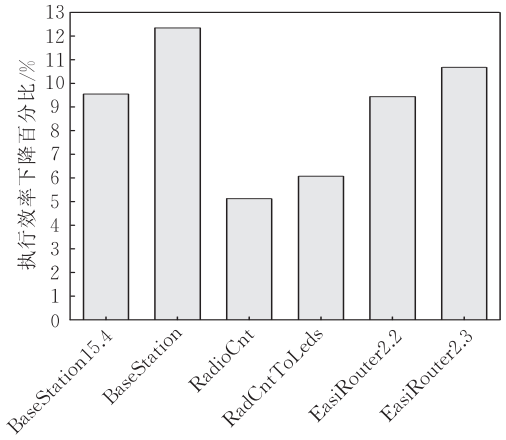


图 12　函数的二次调用对程序执行效率的影响

为了提高程序执行效率，我们可以不对与底层硬件相关的函数进行函数二次调用.但这样的后果是如果这部分函数被更新，则可能导致对调用这部分函数的指令进行大量修改，增加了代码更新开销.

6　相关工作

Deluge[11]是较早的代码更新方法.该方法在代码无线分发阶段需要传输更新协议和整个 TinyOS 镜像.当被更新节点收到新代码时直接写入外部 flash.当更新代码接收完毕后，通过 Bootloader 将代码读入内部程序 flash，进行硬件重启完成更新，因此更新开销大.针对 Deluge 的不足，文献[14-17]提出了不同的解决方案.Elon[12]以 TinyOS 为基础，将代码以组件（component）为单位放入 RAM 中执行，提高了更新效率，但它在更新时以组件作为传输和重组的基本单位，包含了大量无关代码.Kim 等人[13]提出的 Tiny Module-Link 主要考虑了由于读写外部 flash 引起的重组开销，将新程序代码重建全部放入 RAM 中进行，但是在完成重建后，仍然需要将所有被更新的函数回写到内部 flash 中.

增量式代码更新方法通过传输差异代码降低传输开销.其中 Hermes[7]采用 Rsync 算法计算字节级差异，但是由于它必须对外部 flash 进行读写，增加了重组开销.Koshy 和 Pandey[6]试图通过给每个函数末尾添加溢出空间（slop region）来存放插入的代码，尽量避免代码重建.但是溢出空间会导致大量无效的存储碎片，而且插入的代码量受到溢出空间大小的限制.

除了专门的远程代码更新方法之外，SOS[18]、Contiki[19]等操作系统使用了动态链接方法实现代码更新，但是这些操作系统在更新代码时需要传输符号表和重定位表，增加了传输开销，并且也无法对操作系统内核模块进行代码更新.Mate[20]和 ASVM[21]在节点上实现了虚拟机技术，可以在传输少量代码的情况下，完成代码更新.然而虚拟机代码是一种紧凑型代码（compact code），与本地码（native code）相比执行效率过低，并且表达能力有限.

7　结束语

目前大多数传感器网络代码更新研究仍然集中

在如何有效降低传输开销上，而对如何有效地降低重组开销的研究则较少.然而通过在故宫中布署传感器网络[1]的工程实践，我们发现有时重组开销能够超过传输开销成为代码更新的主要开销.

本文介绍了一种基于代码缓存机制的低开销远程代码更新方法 EasiCache.该方法通过代码缓存机制将部分代码动态地保存在 RAM 中，尽量避免对 flash 元件的读写，从而有效降低了代码更新的重组开销.同时在 EasiCache 中，针对代码更新中传输开销较高的问题，提出了与代码缓存机制相适应的函数级代码差异对比技术.通过单次更新实验和连续更新实验，我们验证了 EasiCache 在降低更新开销方面的优势.

在未来的工作中，我们将进一步研究与代码缓存机制相适应的替换算法，提高在动态变化场景中的缓存命中率.

参 考 文 献

[1] Li Dong, Hui Chun-Li, Huang Xi, Zhao Ze, Cui Li. Application case of wireless sensor networks: Museum. Communications of CCF, 2006, 2(5): 72-74(in Chinese)
(李栋，回春立，黄希，赵泽，崔莉. 无线传感器网络在故宫环境监测中的应用. 中国计算机学会通讯，2006，2(5)：72-74)

[2] Li D, Zhao Z, Cui L, Zhu H et al. The design and implementation of a surveillance and self-driven cleanup system for blue-green Algae Bloom on Lake Tai//Proceedings of the 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems. San Francisco, USA, 2010: 759-761

[3] Panta R K, Bagchi S, Midkiff S P. Zephyr: Efficient Incremental reprogramming of sensor nodes using function call indirections and difference computation//Proceedings of the USENIX Annual Technical Conference. San Diego, USA, 2009

[4] Dong W, Liu Y, Chen C, Bu J J. R2: Incremental reprogramming using relocatable codes in networked embedded systems//Proceedings of the 30th IEEE International Conference on Computer Communications. Shanghai, China, 2011: 376-380

[5] Sun Ning-Hui, Xu Zhi-Wei, Li Guo-Jie. Sea computing: A novel computing model of internet of things. Communication of China Computer Federation, 2010, (2): 39-43(in Chinese)
(孙凝晖，徐志伟，李国杰. 海计算：物联网的新型计算模型. 中国计算机学会通讯，2010，(2)：39-43)

[6] 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

[7] 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

[8] 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

[9] 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

[10] Polastre J, Szewczyk R, Culler D. Telos: Enabling ultra-low power wireless research//Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks. Los Angeles, USA, 2005: 364-369

[11] 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

[12] Dong W, Liu Y, Wu X, Gu L, Chen C. Elon: Enabling efficient and long-term reprogramming for wireless sensor networks//Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. New York, USA, 2010: 49-60

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

[14] 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

[15] 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

[16] Marrón P J, Gauger M, Lachenmann A, Minder D et al. FlexCup: A flexible and efficient code update mechanism for sensor networks//Proceedings of the 3rd European Workshop on Wireless Sensor Networks. Zurich, Switzerland, 2006: 212-227

[17] Rossi M, Zanca G, Stabellini L et al. 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. Francisco, USA, 2008: 188-196

[18] Han C, Rengaswamy R, Shea R, Kohler E, Srivasta M. SOS: A dynamic operating system for sensor networks//Proceedings of the 3rd International Conference on Mobile Sys-

tem, Applications, and Services. Seattle, USA, 2005: 163-176

[19] Dukels A, Gronvall B, Voig T. Contiki- A lightweight and flexible operating system for tiny networked sensors//Proceedings of the IEEE Workshop on Embedded Networked Sensors. Tampa, USA, 2004: 455-462

[20] Levis P and Culler D. Mat′e: A tiny virtual machine for sensor networks//Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, USA, 2002: 85-95

[21] Levis P, Gay D, Culler D. Active sensor networks//Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation. Boston, USA, 2005: 343-356

QIU Jie-Fan, born in 1984, Ph. D. candidate. His research interests include embedded operation system and wireless sensor network.

LI Dong, born in 1979, Ph. D., assistant professor. His research interests include wireless sensor networks and ad-hoc networks.

SHI Hai-Long, born in 1986, Ph. D. candidate. His research interests include distributed operation system and Internet of Things.

DU Wen-Zhen, born in 1989, M. S. candidate. His research interests include wireless sensor network and Internet of Things.

CUI Li, born in 1962, Ph. D., professor, Ph. D. supervisor. Her research interests include sensor technology, wireless sensor networks and Internet of Things.

Background

In this paper, we propose a novel low-overhead reprogramming approach — EasiCache. The approach mainly solves the high reprogramming overhead problem. The existing researches mainly focus on how to reduce reprogramming code size for lowering transmission overhead. Dong W et al propose transferring the changed functions by presetting replaceable component. Incremental reprogramming approaches such as Hermes merely transfer the different codes between the new and the old program instead of the entire program.

However, the programming overhead brought by storing and rebuilding codes is less considered. The Kim S K et al use the RAM to rebuild codes and avoid writing/reading the high-power external flash, but not give a consideration for writing/reading the internal flash. Our research focuses on reducing the programming overhead by cache mechanism which dynamically stores and executes the parts of program in the low-power RAM. By the mechanism, the frequently changed codes are not written into the internal flash, and thus not read from it. We also propose the different code comparison between functions to lower the transmission overhead and reserve the program structure information needed by the cache mechanism. According to the information, we design three programming operations which can avoid the conduct of code rebuilding. Compared with the previous approaches, EasiCache efficiently avoids the operation to high-power flash for lowering the programming overhead and further gets an advantage of the entire reprogramming overhead.

This research work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61003293, the Beijing Natural Science Foundation under Grant No. 4112054, the CAS Knowledge Innovation Program under Grant No. 20106030, and the National Science and Technology Major Project of China under Grant No. 2010ZX03006-003-02.