软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(10):3844–3857 [doi: 10.13328/j.cnki.jos.006308]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向星载计算机的双重索引数据压缩方法*

邓岸华[1,2], 乔 磊[2], 杨孟飞[3]

[1](西安电子科技大学 计算机科学与技术学院, 陕西 西安 710071)

[2](北京控制工程研究所, 北京 100190)

[3](中国空间技术研究院, 北京 100094)

通信作者: 乔磊, E-mail: 13811459711@163.com

摘 要: 随着星载计算机系统功能的日益复杂, 程序规模也在快速扩大. 在存储资源极其受限的背景下, 需要稳定、有效的代码压缩功能来保障星载软件的正常存储与运行. 混合压缩算法是目前无损数据压缩的主流算法, 具有压缩率高、代码规模和计算资源需求大的特点. 然而, 在航天星载计算机等嵌入式系统中, 由于其运行环境特殊, 需要较高的可靠性和抗干扰能力, 无法实现混合压缩算法应有的效果. 同时, 单一压缩模型压缩率较低. 针对以上问题, 在 LZ77 算法代码体积和内存消耗优势的基础上提出了改进方法: 为压缩过程设计一种新的匹配记录表以存储高价值数据索引来辅助压缩, 实现了原算法局部性优势与高价值数据全局分布的互补, 更大程度上减少了数据冗余; 结合动态填充、变长编码等进一步优化编码结构, 降低存储需求; 最终, 设计并实现了一种更加适合航天嵌入式环境的无损数据压缩算法(LZRC). 实验结果表明: (1) 新算法在比 LZ77 算法代码体积仅多出 3.5 KB 的条件下, 对软件代码的平均压缩比提高了 17%; (2) 新算法的运行内存需求仅为混合压缩算法的 12%, 代码体积也减少了 84%, 更加适合星载计算机系统.

关键词: 数据压缩; LZ77; 嵌入式操作系统; 资源受限; 匹配记录表; 多重索引

中图法分类号: TP311

中文引用格式: 邓岸华, 乔磊, 杨孟飞. 面向星载计算机的双重索引数据压缩方法. 软件学报, 2022, 33(10): 3844–3857. http://www.jos.org.cn/1000-9825/6308.htm

英文引用格式: Deng AH, Qiao L, Yang MF. Double Index Data Compression Method for Onboard Computer. Ruan Jian Xue Bao/Journal of Software, 2022, 33(10): 3844–3857 (in Chinese). http://www.jos.org.cn/1000-9825/6308.htm

Double Index Data Compression Method for Onboard Computer

DENG An-Hua[1,2], QIAO Lei[2], YANG Meng-Fei[3]

[1](School of Computer Science and Technology, Xidian University, Xi'an 710071, China)

[2](Beijing Institute of Control Engineering, Beijing 100190, China)

[3](China Academy of Space Technology, Beijing 100094, China)

Abstract: As the functions of on-board computer systems become more and more complex, the scale of programs is also growing rapidly. A stable and effective code compression function is required to ensure the storage and operation of on-board software under the background of extremely limited storage resources. Hybrid compression is currently the mainstream algorithm for lossless data compression, and it has the characteristics of high compression rate, large code size, and large demand for computing resources. However, embedded systems such as spaceborne computers require reliability and anti-jamming capabilities, and hybrid compression cannot play its amazing effect. At the same time, the compression rate of a single model is too low to meet the demand. This study proposes an improvement method based on the advantage of the low resource requirement of LZ77 algorithm, the details are as follows. The new algorithm uses a new matching record table for the compression. This table stores high-value indexes to assist in compression, which

* 基金项目: 国家自然科学基金(61632005, 62032004, 61802017)

收稿时间: 2020-09-28; 修改时间: 2020-11-04, 2020-12-08, 2021-01-06; 采用时间: 2021-01-26

realizes the complementarity between the local advantages of the original algorithm and the global distribution of high-value data, and reduces data redundancy to a greater extent. In addition, the new algorithm combines dynamic filling, variable length coding, and other methods to further optimize the coding structure and reduce storage requirements. Finally, the lossless data compression algorithm (LZRC) is designed and implemented, which is more suitable for the aerospace field. Experimental results show that: (1) the code size of the new algorithm is 3.5 KB more than the code size of the original algorithm, and the average compression ratio of software has increased by 17%; (2) compared with hybrid compression, the runtime memory of the proposed algorithm is only 12%, and the code size is also reduced by 84%, which is more suitable for on-board computer systems.

Key words: data compression; LZ77; embedded operating system; limited resources; matching record table; multiple index

作为资源受限的典型嵌入式设备, 星载计算机所使用的存储单元是关键资源, 在设计时对整个系统的性能和结构都有着决定性影响. 随着系统功能需求的不断增加[1], 其代码规模也在急速膨胀, 原本用来存放操作系统和应用软件的 PROM 无法提供足够的空间[2]. 而单纯增加 PROM 容量会导致成本、功耗、体积和重量的上升, 威胁到星载计算机的平稳运行[3], 大大降低了这种安全关键系统的可靠性. 针对以上问题, 亟需一种数据压缩方法来保证星载软件的正常存储与运行.

现如今的主流无损压缩技术是以 LZ77[4]衍生算法占据主导地位的, 如 LZMA[5]、Deflate[6]等. 由于单一模型 LZ77 算法代码规模小、执行时间快, 但压缩率不能满足大部分场景需求, 因此, 现代压缩技术将其与其他算法结合进行混合压缩; 这类混合压缩算法具有压缩率高、代码规模大、存储资源需求多、实现较为复杂等特点. 但是由于太空辐照、重量和成本等影响, 导致航天星载计算机的资源十分有限, 其 PROM 只有几百字节, SDRAM 只有数兆字节, 上述现代主流压缩算法无法在计算、存储受限以及可靠性要求高的条件下实现预先的压缩效果. 此外, 针对资源受限的压缩算法研究[7]集中于指令集分析上, 利用指令分割与指令重新编码技术优化 Huffman 算法, 但是实现较为复杂且不易于移植, 程序复用性较差. 文献[8]使用一种基于压缩后缀树技术将 LZ77 因式分解控制在线性时间. 文献[9]针对实时应用背景, 将 LZ77 压缩数据转换成一个生成文本的上下文无关语法, 只需使用语法的内存空间, 即可在线性时间中进行解压. 文献[10]利用 LZ77 与 Burrows-Wheeler 变换, 在线性空间内建立重复感知自索引, 其主要问题如下.

(1) 对于运行在复杂环境中的航天星载计算机来说, 压缩算法的实现需要体积较小的代码以及较高的抗干扰能力来保证系统的可靠性. 虽然单一模型算法实现体积小, 但压缩率难以满足需求. 而针对资源受限问题的研究旨在解决现代数据量爆炸对内存的负担和某些实时应用需求, 并非面向嵌入式轻量级压缩算法设计, 且未在压缩率上取得进一步提升;

(2) 为了实现更高的压缩比率, 桌面级压缩算法采用混合压缩思想, 在 LZ77 压缩的基础上再使用 Huffman 等熵编码进行混合压缩, 导致最后的代码规模与内存开销远高于单一压缩模型算法, 通常, 算法实现能达到近 100 KB 的规模, 超过了星载 PROM 的容量限制; 并且 RAM 开销是 LZ77 的数十倍, 会在系统运行时影响到其他任务的内存分配, 这些额外开销是星载计算机所不能提供的.

本文针对以上问题, 在单一压缩模型 LZ77 资源需求低的优势上进行改进. 面向资源极其受限的星载计算机, 提出一种基于匹配记录表的双重索引数据压缩算法, 在极低的 PROM 和 SDRAM 需求下, 对星载程序进行更高效的压缩处理, 具体创新点与贡献如下.

(1) 设计匹配记录表以弥补全局性缺陷. 本文设计了一种新的匹配记录表来记录所匹配到的具有高价值数据的索引, 其拥有相应的更新机制与存储规则, 与字典窗口结合形成双索引结构, 能够很好地实现单一压缩模型的局部性优势与高价值数据全局分布的互补, 弥补了字典压缩的全局性缺陷, 进一步减少了数据冗余;

(2) 优化了编码结构与编码方法. 本文在压缩过程中设计了更加紧凑的结构, 以 bit 为最小单位, 将匹配与未匹配情况分开处理, 并且设计了动态填充窗口机制与变长编码方法, 有效提升了编码效率;

(3) 极低的代码规模与内存消耗. 新算法在设计相关模型时, 以低存储需求和高压缩率并重, 以双索引结构来定位重复数据, 用高效的编码替换规则恢复原数据, 其代码规模与内存需求远低于混合压缩方法;

(4) 实验验证新算法的有效性. 本文首先在 PC 平台上验证 LZRC 算法的综合性能, 再将其移植到我国火星探测器计算机中以验证算法的有效性. 实验结果表明: LZRC 算法的平均压缩比较 LZ77 算法提高了 17%, 并且代码体积与内存需求仅为混合压缩的 1/6, 十分适合星载计算机的代码压缩需求.

1 研究背景

1.1 星载计算机

星载计算机是计算机技术在空间环境下的应用, 负责完成空间飞行器、星球探测器等设备的控制和数据处理任务. 由于空间环境的恶劣条件, 星载计算机需要耐受极端恶劣的温度、振动、各种高能粒子以及宇宙射线的辐射, 从而对其在性能、可靠性和成本上提出了巨大的挑战. 在高昂的研究与制造费用、有限的硬件资源下, 要确保系统的可靠运行, 是一项困难又关键的任务.

火星探测器(火星车)主要负责火星地貌探测、水冰和矿物岩石探查等任务, 是星载计算机的典型应用平台. 由于执行任务环境恶劣, 必须保证火星车设计的高可靠性. 另一方面, 在体积、重量、功耗和可靠性等限制下[11], 导致存储、计算资源极其受限. 处理器采用我国自主研发的单核 SOC2008 (25 MHz), PROM 空间 256 KB, SDRAM 空间 8 MB, 安装运行 SpaceOS[12]嵌入式实时操作系统. 火星车程序需要先固化在 PROM 中, 在系统上电后引导至 SDRAM 运行. 然而, 随着功能需求的不断增加, 火星车的操作系统和应用程序代码接近 7 万行, 编译完后的目标代码需要 400 KB 的空间, 已经超过了现有的存储设备(PROM)容量限制. 为此, 需要一种有效的代码压缩方法来缩小代码占用空间, 以完成火星车软件的正常存储和运行.

星载软件通常由 3 部分组成: 初始化引导模块 Loader、操作系统以及应用程序. 考虑到最大化发挥压缩带来的体积收益, 本文将操作系统与所有的应用程序一起压缩. 图 1 展示了星载计算机的代码压缩方案, 总体上分为离线阶段和系统启动阶段. 在离线阶段, 先将引导程序 Loader 放置在 PROM 首部, 在 Loader 之后存放操作系统与应用程序联合编译打包压缩的映像. 当系统启动后, 首先从 PROM 中的 Loader 开始运行, Loader 进行一些必要的初始化之后, 将整个压缩映像通过解压缩模块加载至 SDRAM 中, 然后跳转到 SDRAM 并将控制权交给操作系统. 由于压缩和解压缩模块是为了减少存储负担额外引入系统的, 因此需要较低的计算资源与存储需求, 以免影响到星载计算机上其他任务的正常运作. 本文正是针对上述流程中的压缩与解压缩方法来进行研究.

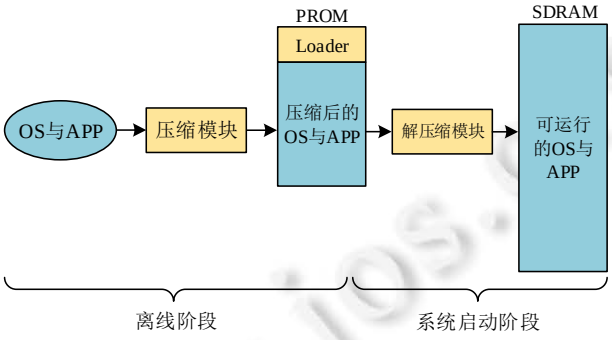

图 1 星载计算机软件压缩

1.2 无损数据压缩技术

基于字典的 LZ77 和 LZ78[13]算法的压缩可行性建立在数据的局部重复性上, 利用一个称作字典窗口的缓冲区存储最近已压缩的部分数据, 将等待压缩的数据在字典窗口内进行模式匹配, 若存在重复时, 则进行编码替换, 以达到压缩的目的.

由于上述单一压缩模型仅采用局部重复信息进行压缩, 很难在压缩率上有进一步的提升. 近年来, 研究重心逐渐落在基于字典与统计的混合压缩上. Gzip 的核心算法 Deflate 首先使用 LZ77 算法进行压缩, 对得到

的编码再使用 Huffman 编码进一步压缩. LZMA 也是先采用 LZ77 进行压缩, 再使用区间编码压缩, 额外引入了 Hash 表索引与马尔可夫链优化时间与内存开销. 混合压缩算法的结构如图 2 所示.

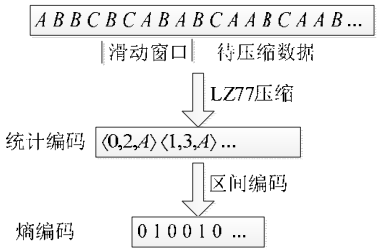

图 2 LZMA 混合压缩算法流程

以上两种算法是目前主流压缩技术的核心部分[14,15], 其压缩比率得到了大幅度的提升. 但是由于使用了两种算法的混合, 代码体积与内存需求也大幅上升, 星载计算机无法提供相应的资源需求.

本文在单一压缩模型 LZ77 算法的存储需求和时间消耗的优势上进行研究. 字典压缩算法使用两个缓冲区在待压缩数据上进行滑动编码, 其中, 一个称为字典窗口缓冲区, 用来存放部分已编码完成的数据, 并将窗口内的数据当作字典索引, 可以对以后重复出现的数据进行编码替换, 其大小固定, 因此只能存放一部分最近的数据; 另一个是前向缓冲区, 存放待压缩的数据. 这样, 两个缓冲区不断地在数据上滑动编码直至结束.

当滑动窗口增大时, 可以存放更多的字典索引内容, 这意味着对等待压缩的字符串有着更大的匹配可能, 压缩比率也更高. 然而, 随着滑动窗口的增大, 首先带来的是匹配过程中时间开销上的极度膨胀; 其次, 滑动窗口的增大直接导致了编码的增长, 用来对字符串进行替换的编码效率也有所降低, 最终的压缩比可能更差. 因此, 两种缓冲区大小的选择对于压缩效果有着十分显著的影响. 除此之外, 原算法在编码结构上存在着大量冗余, 当未能在字典匹配到字符串时, 需要输出若干无效字符, 这部分可以通过匹配标志变长编码来减少冗余. 三元组中的最后一个分量是用于进行定位的字符, 本来, 这个字符可以为下一次匹配提供信息, 被加入三元组后, 不仅影响到了下一次的字符串匹配, 同时也给编码总码长带来 1 字节的冗余, 可以舍弃. 其次, LZ77 算法对各个分量的编码均采用等长码, 由统计数据与数据局部性原理可知, 大部分的成功匹配发生在距离滑动窗口边界的附近, 即可以对偏移值采用变长编码进一步加以优化.

LZ77 算法虽然充分发挥了数据局部性的优势, 但却没有考虑到全局性, 并且当前匹配到的字符串可能在窗口外存在多个相同子串. 是否可以设计某种数据结构或者匹配方法, 将原有的局部优势与某些具有高价值数据的全局分布结合起来, 这将会给整体压缩效果带来进一步的提升.

2 LZRC 压缩算法

2.1 总体设计

本文针对星载计算机这种资源受限环境下嵌入式设备的代码压缩需求, 以高压缩率和低资源需求为设计目标, 在字典压缩的基础上提出了一种基于匹配记录表双重索引结构的 LZRC 算法.

压缩流程如图 3 所示, 首先进行字典窗口缓冲区、匹配记录表等初始化工作, 压缩过程就是将待压缩数据与字典窗口和匹配记录表中的数据进行匹配工作, 若存在一致, 则通过最优判断, 用相应的编码方法来代替原数据达到压缩的目的. 同时, 字典窗口缓冲区会随着压缩的进行更新内部数据. 匹配记录表内的数据按照一定规则进行填充, 容量固定, 拥有相应的替换规则以保证数据的高价值. 与字典窗口相结合, 可弥补字典压缩的全局性缺陷. 同时, 由于匹配记录表的加入, 算法需要对编码结构与编码方法进行相应的设计(动态填充、变长编码、最佳替换判断等)以达到高效率的编码替换. 在设计过程中, 为了减少算法对存储的需求, 匹配记录表和滑动窗口均记录与原数据相关的索引结构. 具体思路如下所述.

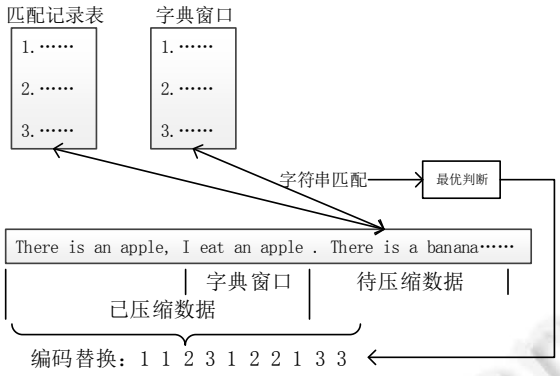

图 3 LZRC 压缩流程

2.2 优化数据缓冲区，允许匹配重叠

前向缓冲区的目的在于保存将要压缩的部分数据，将此缓冲区内的字符串与滑动窗口进行迭代模式匹配，匹配到字符串的最大长度总是会被前向缓冲区的大小所限制. 因此可以不再采用前向缓冲区，只保留字典窗口缓冲区. 在字符串的匹配过程中会出现类似图 4 的情况: 当待压缩字符串在字典窗口中进行模式匹配时，窗口内已经匹配到最后一个字符，但是仍然可以继续匹配，即待压缩字符串本身在某处产生了部分重复的现象. 此时继续匹配所得到的三元组，仍然可以唯一地恢复出原始字符串. 本文将这种现象称为匹配重叠(overlap). 在优化缓冲区和允许匹配重叠的情况下，短语数量[16]有一定程度的减少. 在这里，短语数量代表着三元组编码的个数，它是压缩效果的直接体现. 在相同的数据条件下，短语数量的减少表明编码效率的提高，平均每个三元组能够表示更长的字符串. 注意，此时短语是包括匹配成功与失败的总个数. 表 1 和表 2 分别展示了前向缓冲区与匹配重叠对短语数量的影响.

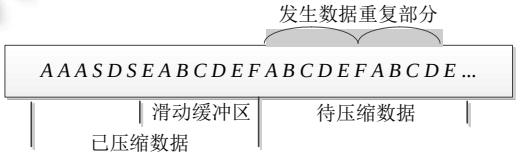

图 4 待压缩数据发生部分重复导致的匹配重叠现象

表 1 前向缓冲区对短语数量的影响

压缩对象	短语数量/个	
	含有前向缓冲区	舍弃前向缓冲区
a.bin	10 264	9 935

表 2 匹配重叠对短语数量的影响

压缩对象	短语数量/个	
	不允许 overlap	允许 overlap
a.bin	9 935	9 924

表 2 的实验前提是舍弃前向缓冲区，由于通常前向缓冲区较小，这会限制匹配过程中允许重叠的效果. 在这里，前向缓冲区大小为 16 字节，滑动窗口为 1 024 字节，压缩对象 a.bin 为部分嵌入式操作系统代码的二进制形式.

2.3 优化编码结构与编码方法

2.3.1 编码结构

在介绍编码结构前，需要先说明与以往按字节为单位进行编码的不同，LZRC 算法是逐字节地处理数据，

并按 bit 为单位进行编码, 这样使得各分量码长更加灵活. 之前提到原算法的编码结构存在大量冗余, 本文设计了一种更加合理的编码结构, 如图 5 所示.

图 5 编码结构

LZRC 算法使用了一位匹配标志位来判断是否匹配: 当未匹配时, 只需要写入 8 位的 ASCII 值; 匹配成功时, 写入偏移量与字符串长度, 这样能够去除由于未匹配带来的编码冗余. 其次, 原算法在匹配成功时仍然需要在末尾写入字符串的后一位字符, 这不仅会使得编码码长增加, 还会影响到下一次的模式匹配. 因此, 本文舍弃了后一个字符, 结果按照编码仍然可以唯一地恢复匹配字符串.

2.3.2 动态填充与变长编码

本文在对偏移量与字符串长度的编码中均使用变长编码, 前文提到过偏移量 *offset* 的分布特性, 若按照数据的局部性原理来说, 字符串匹配的位置总是靠近滑动窗口起始点的, 因此可以对 *offset* 采用按长度分类编码, 比如当滑动窗口容量 m=1024 B 时, 则 *offset* 总码长为 10 位; 当匹配偏移 *offset*<32 B 时, *o_len* 只需要 5 位码长, 剩下的情况使用 10 位码长, 但是需要额外增加 1 位标志位. 然而在实际应用中我们发现, 匹配位置更多的是分布在窗口中间, 因此额外的标志位反而带来更多的负担.

此外, 在刚开始进行压缩的过程中, 窗口是逐渐填满的, 因此可以在窗口未满时使用动态编码, 码长 *o_len*=*upper*_log(*window*). 其中, *window* 是指逐渐填充的窗口大小.

对于 *length* 的编码方法设计类似于计算机内存空间地址的分配, 默认 *length* 大于 1 时进行编码, 这是由于只匹配到一个字符时的编码码长反而多于不编码时码长. 编码举例如图 6 所示.

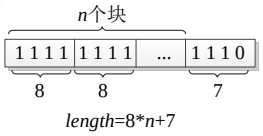

图 6 *length* 分段编码

最终得到的对 *length* 的编码实质上是将字符串长度进行分段存储的前缀编码, 当长度大于一定阈值时, 则由若干个连续的 1111 和最后的余数组成. 通过多次实验, 本文设计的编码以 8 为阈值, 这种前缀编码计算速度较快也有着较好的编码效率.

2.4 匹配记录表

LZ77 算法很好地照顾到了数据的局部性特点, 但是对于全局性则没能很好地利用; 其次, 当在滑动窗口中成功匹配到字符串时, 该字符串很可能会在窗口以外的地方多次出现. 为此, LZRC 算法设计了一种数据结构, 称为匹配记录表, 采用链表的形式将每次匹配到并且符合要求的字符串索引按照一定规则存储起来. 这样, 在每次压缩时, 就需要先在匹配记录表中进行字符串匹配, 再在滑动窗口中匹配, 每次选取两者中压缩效果最佳的索引进行编码替换. 匹配记录表大小固定, 拥有相应的更新机制和存储规则. 将匹配记录表与滑动窗口相结合, 能够进一步提高匹配几率和编码效率. 加入匹配记录表后的编码结构如图 7 所示.

当未成功匹配时, 仍然按照之前的规则写入标志位 0 和字符的 ASCII 码, 主要变化在于匹配成功时. LZRC 对匹配成功时的编码结构进一步划分, 匹配过程首先在匹配记录表中进行查找, 再在滑动窗口中进行模式匹配; 若仅在前者匹配成功, 则写入匹配成功标志 1 再写入目标字符串在匹配记录表中的索引; 若仅在后者匹配成功, 则依次写入匹配标志 0、偏移量和字符串长度; 紧接着, 还需要将该字符串存放进匹配记录表. 还有一种情况是两者都匹配成功时, 就需要进行最优判断: 以编码效率较高者进行选择. 在这里需要注意的

是：若不进行有效判断，则会导致匹配记录表中充斥着大量碎片和低价值字符串，这会导致同一个字符串需要更多的编码单元表示，同时也降低了匹配记录表的命中率．加入匹配记录表后的数据结构如图 8 所示．

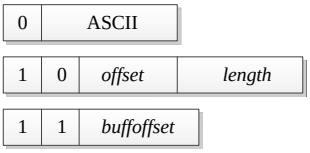

图 7　LZRC 编码结构

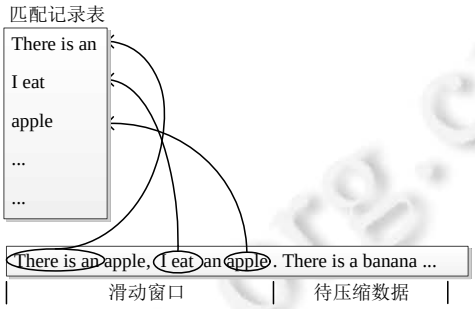

图 8　LZRC 匹配数据结构

当压缩过程成功匹配到字符串时的编码算法如下所示．

ALGORITHM 1. Encoding process upon successful matching during compression.

Input: offset: the offset of the matched string in the window;
length: the length of the matched string in the window;
buffLength: the length of the matched string in the record table;
buffOffset: the offset of the matched string in the record table;
buffSum: the number of elements in the matching record table;

Output: encoding result *str*.

```
1   if (match successfully) then
2   |   Write1toCompression(str);                    //成功匹配到字符串时写入标志位
3   |   if (length>buffLength) then
4   |   |   Write0toCompression(str);                //判断最长匹配字符串的有效位置
5   |   |   offsetCoding(offset);                    //对滑动窗口内的匹配字符串偏移编码
6   |   |   lengthCoding(length);                    //对滑动窗口内的匹配字符串长度编码
7   |   |   if (buffSum<maxnum) then
8   |   |   |   buff_insert(offset,length);         //更新匹配记录表
9   |   |   else
10  |   |   |   buff_delete, buff_insert(offset,length);   //保持记录表内元素数量恒定
11  |   |   end
12  |   else Write1toCompression, buffOffsetCoding;   //
13  |   end
14  |   return str
15  end
```

匹配记录表按链表组织，以字符串的长度降序存储，实质上是以在滑动窗口中匹配到的字符串长度为价

值标准, 字符串越长, 代表每次命中时单位编码能够表示更多的数据量. 当匹配记录表的内容填满时, 再次插入字符串需要将链表的最后一个元素删除, 以保证记录表元素数量与顺序的恒定. 对于表内的组织规则, 本文也尝试过以访问频率进行排序, 但是效果较差. 此外, 在实现过程中, 匹配记录表并不存储字符串实际内容, 而是通过使用一个指向原始待压缩数据的字符指针与一个整型 *length* 变量来确定唯一的字符串, 这样就进一步减少了时间和内存消耗. 影响上述编码算法时间因素的关键在于将匹配字符串的索引信息插入匹配缓冲表, 每次需要遍历表来保证表中字符串长度的有序, 但是由于表内元素数量控制在 1 024 以内, 因此时间复杂度为 O(1).

原始压缩的字符串使用匹配记录表, 可以有效地进一步减少冗余, 然而, 因为每次匹配需要在匹配记录表和滑动缓冲区中进行查找, 这必定会带来时间复杂度的上升, 而增加优化措施又会带来代码规模与内存消耗的增加, 所以需要对窗口与记录表的大小进行合适的选择. 本文在多次实验后, 选择 1 KB 的匹配记录表与 1 KB 的滑动缓冲区, 压缩算法如下所示.

ALGORITHM 2. LZRC compression process.

Input: data waiting to be compressed;

Output: compression result compre.

```
1   Initialize window, buff;                          //初始化滑动窗口与匹配记录表
2   foreach characterInstring do                      //逐字符处理
3   |   buffOffset, buffLength=FindinBuff(character,buff);   //匹配记录表内查找
4   |   length, offset=FindinWindow(character,window);       //滑动窗口内查找
5   |   if (length>1||bufflength>1) then
6   |   |   StringCoding(offset,length,buffLength,buffOffset);
7   |   |   WriteCodetoCompression;
8   |   else
9   |   |   Write0toCompression(compre);
10  |   |   WriteCtoCompression(compre);                //保持记录表内元素数量恒定
11  |   end
12  |   moveCharactertoWindow(·);                       //
13  |   update Character;
14  end
15  return compre
```

整个压缩过程是在 *while*(·)中遍历完所有字符, 由于加入了匹配记录表的额外处理开销, 整个压缩算法时间复杂度为 O(*m*×*n*), 其中, *m* 代表了编码效率因子, 由压缩过程的匹配次数决定, 每次成功匹配均需调用滑动缓冲区与匹配记录表的相关操作, 虽然两者的容量均控制在 1 KB 以内, 但是随着匹配次数的增加, 会对整体时间产生较大影响. 此外, 由于新算法仅引入了匹配记录表的 1 KB 常数空间, 在空间复杂度上, LZRC 算法仍然为 O(*n*).

LZRC 算法的解压缩过程如下所示.

ALGORITHM 3. LZRC decompression process.

Input: data waiting to be decompressed;

Output: source data string.

```
1   Initialize window, buff, bit;          //初始化滑动窗口、匹配记录表与解压缩标志位
2   where bit<compre.size do
3   |   if (compre[bit++]=1) then          //有匹配到字符串
4   |   |   if (compre[bit++]=1) then      //在匹配记录表中为有效恢复数据
```

```
5  |  |  |  str=readBuff(bit);
6  |  |  |  WriteStrtoString(str);                    //从匹配记录表中解压缩数据
7  |  |  else
8  |  |  |  str=readWindowCode(bit);                  //从滑动窗口中恢复数据
9  |  |  |  WriteStrtoString(str);
10 |  |  end                                          //保持记录表内元素数量恒定
11 |  else
12 |  |  str=readCharacter(bit);
13 |  |  WriteCtoCompression(compre);                 //没有发生匹配，恢复单个字符
14 |  end
15 |  bit+=codelength;                                //更新解压缩数据位置
16 end
17 return string
```

解压算法是压缩的逆过程，首先读入标志位，并根据标志位来判断有无匹配字符串，若无匹配，则直接读取 1 字节的 ASCII 码并写入解压缩文件；否则，继续读取匹配位置标志位，由匹配位置来决定字符串的恢复来源. 此时，滑动窗口缓冲区与匹配记录表的构建在解压过程中随着字符串的恢复来完成. LZRC 解压缩算法充分继承了 LZ77 的优势，即不需要像压缩过程中那样以 O(m×n)的时间代价进行搜索，而是直接得到字符串的偏移与长度，这样可以很快地将字符串唯一地恢复，而大部分的时间消耗在构建缓冲区的过程中，其时间复杂度和空间复杂度均为 O(n).

3 实验测评

3.1 实验配置与算法实现

为了更好地对 LZRC 算法进行综合测试，本文将实验分为两部分，其中，

- 上半部分在 PC 平台上将新算法与单一压缩模型 LZ77、现代混合压缩 LZMA 进行对比. 考虑到性能差异问题，三者均在 GCC 编译器环境下实现；
- 实验的下半部分则是将其移植到火星车计算机进行实际验证. 此时，由于混合压缩算法的代码规模和运行内存需求大，火星车计算机中的存储设备无法提供，因此仅对比了新算法与 LZ77 的性能.

火星车计算机是典型的资源受限嵌入式系统，使用基于 SPARC 架构自主研发的 32 位抗辐照单核处理器 SoC 2008，其上运行自主研发的 SpaceOS 实时操作系统，具有较高的可靠性，已经广泛应用于我国多个重大航天工程中. 实验平台参数见表 3，火星车计算机验证系统关系图如图 9 所示.

表 3 系统实验环境

实验系统参数	
平台: PC	平台: 火星车计算机
CPU: Intel i5-3470	CPU: SoC 2008 单核
主频: 3.2 GHz	主频: 25 MHz
内存: 8 GB	内存: PROM 256 KB, SDRAM 8 MB
操作系统: Win 7	操作系统: SpaceOS 2.0

在系统验证平台中，中心计算机为火星车计算机的核心部分，其上运行空间操作系统，具备状态数据获取能力，接收来自人机接口仿真计算机注入的指令，完成数据分析、计算与控制. 人机交互及显示系统运行集成开发环境，具备监控调试中心计算机的能力. 态势感知模拟系统负责模拟火星车实际运行环境，并将红外、电磁等数据通过态势感知计算机提供给中心计算机. 敏感器模拟系统通过硬件线路与软件仿真计算，模拟星上敏感器和卫星动力学特性，提供中心计算机所需各部件输出数据，通过输入输出接口，使地星联试设备构

成闭环回路.

在算法移植过程中，将压缩与解压缩模块源码在人机接口仿真计算机的编译环境下重构，在离线阶段，将操作系统与应用软件联合编译代码通过压缩模块进行压缩得到映像，并与解压缩模块通过人机接口仿真计算机注入至中心计算机，在系统启动时，由解压缩模块对映像进行解压缩释放，最终将控制权交给操作系统. 为了更准确地测量算法的各项指标，验证实验采用软件插桩的方式获取算法的运行时间等数据.

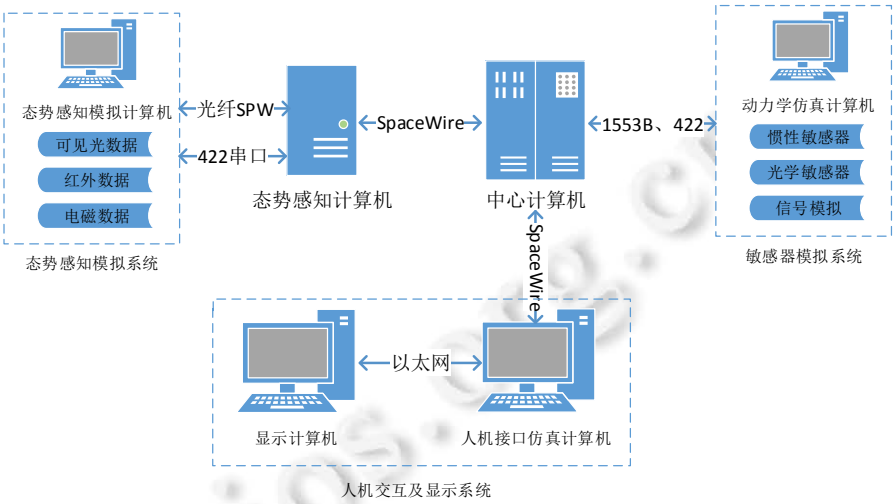

图 9 火星车计算机验证系统关系图

3.2 详细指标分析

实验压缩对象选择 Calgary Corpus 语料库[17]，其内容是比较常用的文本和二进制数据的集合，多用于无损压缩算法的性能测试，另外再加入 SpaceOS 与应用程序部分源码作为额外的测试对象(命名为 a). 测试指标分为两大部分：一是衡量压缩效果的压缩比与压缩时间，另一部分是算法运行时的内存消耗与代码体积. 作为应用在星载计算机等资源受限且可靠性需求高的嵌入式平台上的数据压缩算法，首先需要考虑的是代码体积与内存消耗的影响，这是保证系统能否稳定运行的关键.

$$压缩比=\frac{压缩后文件大小}{压缩前文件大小}.$$

首先在 PC 平台上对 3 种算法进行综合测试，各算法对基准文件的压缩前后大小与压缩比结果如图 10 和表 4 所示，算法的压缩与解压缩运行时间对比如图 11 和图 12 所示，图中横坐标标号代表 14 个压缩对象.

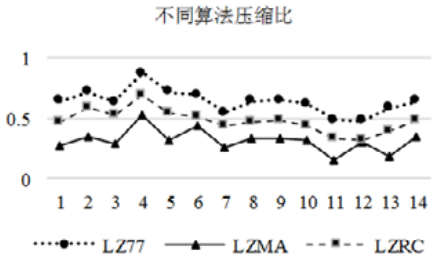

图 10 PC 平台中不同算法的压缩比

表 4 不同算法的文件压缩大小(单位: B)

压缩对象	原始大小	LZ77	LZMA	LZRC	压缩对象	原始大小	LZ77	LZMA	LZRC
bib	111 261	71 172	30 623	51 716	paper1	53 161	33 796	17 223	24 324
book1	768 771	547 332	260 967	452 100	paper2	82 199	53 252	27 211	39 428
book2	610 856	385 284	169 797	318 724	progc	39 611	24 324	12 509	16 900
geo	102 400	88 836	53 074	69 892	progl	71 646	34 564	10 332	23 556
news	377 109	268 804	119 304	204 804	progp	49 379	23 556	14 964	15 620
obj1	21 504	14 852	9 378	10 756	trans	93 695	54 020	16 799	36 100
obj2	246 814	133 636	61 663	106 500	a	28 216	17 924	9 835	13 572

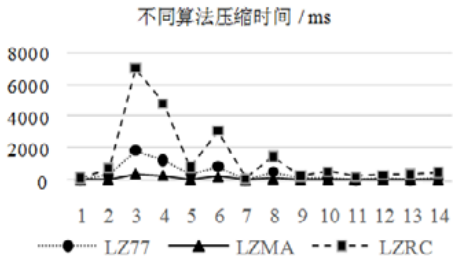

图 11 PC 平台中不同算法的压缩时间

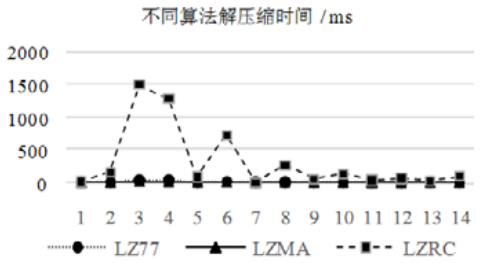

图 12 PC 平台中不同算法的解压缩时间

通过压缩效果的测试可以得出: 在基准压缩对象中, LZRC 算法的平均压缩比较 LZ77 算法提升了 13%. 而在运行时间上, LZRC 算法对一些数据量较大的纯文本类型数据进行压缩时效果较差, 但对程序源码等二进制文件进行压缩的时间开销在系统允许的范围内, 这也符合我们针对星载计算机进行代码压缩的目标. 由于前两者都属于单一模式压缩, 无法达到 LZMA 混合压缩算法的压缩比率, 并且 LZMA 采用多种数据结构的优化, 利用空间换时间, 在运行时间上也大大缩短; 匹配记录表的使用, 给 LZRC 算法带来时间上的额外开销, 若使用类似 LZMA 的数据结构优化方法, 则无法保证 LZRC 算法在代码体积与内存消耗上的优势, 这直接关系到航天星载计算机在复杂环境运行时的可靠性和抗干扰能力. 实验进一步测试了 LZRC 压缩算法在运行时的内存消耗与代码体积, 所有算法的核心代码体积均以在同一编译器编译后的.o 文件的大小来衡量, 结果如图 13 和图 14 所示.

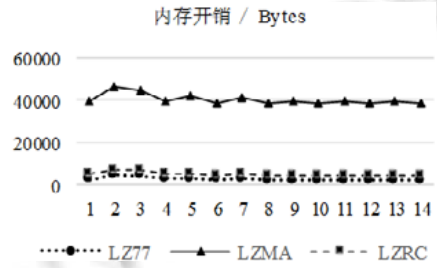

图 13 PC 平台中不同算法的内存开销

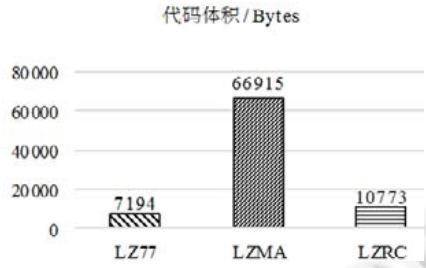

图 14 PC 平台中不同算法的代码体积

通过各算法的内存消耗和代码体积结果可以得出: LZRC 算法的平均内存消耗为 3 898 B, 仅比 LZ77 算法多出 1.4 KB, 并且在代码体积上也只多出 3.6 KB; 而混合压缩算法 LZMA 的平均内存消耗已经超过了 40 KB, 在代码体积上也达到了 67 KB, 这无疑给资源受限嵌入式系统带来了沉重的负担, 无法保证系统在复杂环境中运行的稳定性, 而 LZRC 算法在资源消耗上控制得较好.

将 LZRC 移植到我国火星探测器计算机上进行实际验证, 由于混合压缩 LZMA 算法的代码体积与内存消耗过高, 无法很好地移植到火星车计算机上, 实验仅对比了 LZ77 与 LZRC. 为了符合 LZRC 算法的设计目标, 即缩小嵌入式系统中的程序规模, 压缩对象选取 Calgary Corpus 语料库中的二进制源代码. 平均压缩效果与资源需求如图 15–图 18 所示.

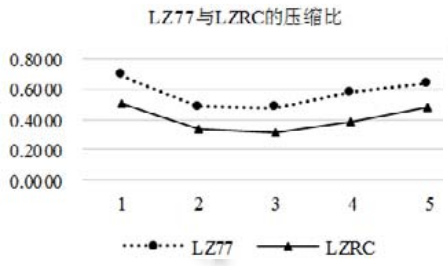

图 15 火星车平台中不同算法的压缩比

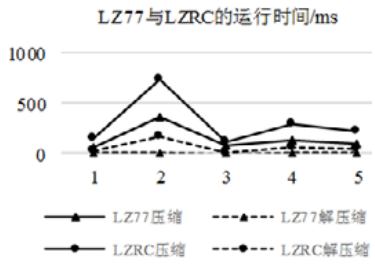

图 16 火星车平台中不同算法的运行时间

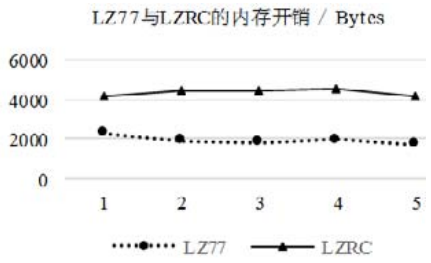

图 17 火星车平台中不同算法的内存开销

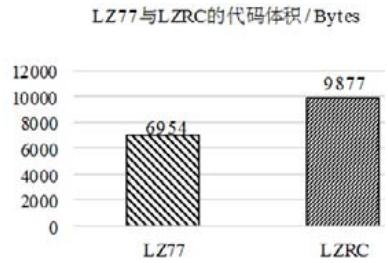

图 18 火星车平台中不同算法的代码体积

在火星探测车计算机的测试中, 压缩对象选择的是基准语料库中的二进制程序代码, 这更符合实际使用场景. 此时, LZRC 的平均压缩比较 LZ77 提高了 17%. 而由于星载计算机的计算能力受限, 对于新算法中增加的匹配记录表处理时间较长, 在总体运行时间上表现较差. 但是作为非实时任务, LZRC 算法的时间开销也在允许范围内. 内存消耗与代码体积指标对于系统的抗干扰能力和可靠性则更加重要, 其在火星探测车平台中的表现与在 PC 中基本一致.

3.3 性能分析

通过以上对 3 种算法在压缩效果与资源消耗这两方面的综合对比可以得出: 桌面级压缩算法代表的 LZMA 虽然在压缩比与运行时间上具有较强的优势, 但其内存消耗和代码体积过高, 仅核心算法实现就达到近 70 KB, 不适合在航天星载计算机上应用; 而新算法 LZRC 虽然时间开销大, 但是作为非实时任务, 在这种资源受限且运行环境复杂、可靠性要求高的背景下具有较好的压缩效果, 对于软件二进制代码的平均压缩比较 LZ77 算法提升了 17%, 并且对于内存消耗与代码体积也控制在较低水平, 仅为 LZMA 的 1/6. 同时, 本文将 LZRC 算法成功移植到我国火星探测车计算机上, 并验证了其压缩的有效性.

4 相关工作

大数据时代的来临, 导致数据的体量及增速都达到了前所未有的高度. 在传输网络和存储能力有限的情况下, 数据压缩技术发挥了越来越重要的作用. 在机器学习经历爆炸式的发展背景下, 数据压缩相关研究人员开始将重心放在深度学习与数据压缩的结合上, 寻求更加高效的压缩方法. Huang 等人[18]首次提出了基于卷积神经网络的块预测模型用于无损视频编码, 极大地提高了 HEVC 标准的性能. Li 等人[19]将栅格编码量化器整合到基于深度学习的图像压缩框架中, 在训练过程中采用软到硬策略进行反向传播, 该模型在低比特率的压缩对象中具有更好的性能. Alexander 等人[20]考虑到基于深度学习的图像压缩过程中量化器不可微问题, 提出了量化的可微近似方法, 并且利用解压块的空间依赖性辅助解压. 上述研究虽然可以很好地利用深度学习处理图像或者视频, 仅存储权重系数以达到数据压缩的目的, 但是神经网络的使用会导致过高的存储和计算消耗, 在一些资源受限的嵌入式环境中, 即使使用剪枝、模型压缩等优化手段, 仍然无法实现应有的效果.

在某些实时应用或存储资源受限等背景下, 需要压缩算法在运行时间与内存上进一步优化. 文献[7–10]正是针对上述问题展开研究. 此外, 文献[21]提出一种应用于高速数据流的实时压缩算法, 基于熵编码方法, 在压缩过程中为待压缩数据分配最少的位, 同时进行性能评估以保证自适应性. 文献[22]针对内存受限问题, 在上下文自适应二进制算术编码的基础上提出一种多参数概率估计方法, 利用基于指数加权移动平均值的两个假设概率估计器来辅助编码, 在不影响压缩率的条件下减少了运行时内存消耗. 以上研究在压缩算法的实时性和内存优化上有着重要贡献, 但未提升压缩率, 且原有算法的压缩效果并不能满足某些场景需求.

星载计算机作为安全关键系统, 需要在外太空的恶劣环境下持续稳定运行数十年, 高可靠性是其设计的首要目标. 文献[23,24]对星载计算机操作系统的设计与验证引入形式化方法, 分别针对 SpaceOS 中的内存管理模块和任务管理需求层进行建模及验证, 有效证明了设计层与需求层的正确性. 文献[25]针对我国首个深空探测再入返回任务设计双平台和多目标飞行器方案, 为嫦娥五号任务的研制奠定了基础. 系统的高可靠性

与低成本往往互相矛盾, 星载计算机为各部件分配的指标极其严格. 然而, 随着系统功能需求的增加, 就需要提升相应的存储能力, 从而导致整个系统成本、重量、功耗等相应的增加, 同时, 可靠性也会受到影响. 本文在火星车引导 Loader 模块中设计了软件代码的压缩方案, 在低存储需求的前提下, 极大地缩小了程序规模. 上述形式化方法的引入, 为星载计算机操作系统的设计与验证提供了有效支撑. 而本文旨在解决星载软件规模增长与存储单元有限的矛盾, 以保证程序的正常存储与运行.

5 总 结

随着我国航天事业的发展, 软件功能日益复杂, 其程序规模也显著扩大, 在资源极其受限的环境下, 需要通过有效的无损数据压缩技术来减轻存储设备的负担, 以保证星载计算机的稳定运行. 单一压缩模型算法的压缩效果不够好, 同时, 主流的混合压缩算法对于内存消耗和代码体积有着过高要求, 无法在航天星载计算机这种资源受限且可靠性、抗干扰能力要求高的环境下应用. 本文为控制资源开销选取单一压缩模型, 设计了一种新的匹配记录表双重索引, 弥补了字典压缩的全局性缺陷, 实现了原算法的局部性优势与高价值数据全局分布的互补, 并且在此基础上进一步设计了有效的编码格式与方法, 进一步减少了数据冗余并占用极低的存储空间, 最终成功移植到我国火星探测车计算机中. 实验结果表明: 在保障系统功能和可靠性的前提下, LZRC 算法相较于 LZ77 和 LZMA 算法能够进行更加有效的数据压缩, 大幅度缩小了程序规模, 保证了星载计算机的可靠运行.

References:

[1] Butler R, Pennotti M. The evolution of software and its impact on complex system design in robotic spacecraft embedded systems. Procedia Computer Science, 2013, 16(3): 747−756.

[2] Ryabtsev VG, Volobuev SV, Shubovich AA. Fault-tolerant architecture of storage device for on-board spacecraft control systems. Russian Aeronautics, 2019, 62(1): 106−112.

[3] Dodd PE, Massengill LW. Basic mechanisms and modeling of single-event upset in digital microelectronics. IEEE Trans. on Nuclear Science, 2003, 50(3): 583−602.

[4] Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Trans. on Information Theory, 2003, 23(3): 337−343.

[5] Salomon D. Data Compression: The Complete Reference. 4th ed., London: Springer, 2007. 241−246.

[6] Rigler S, Bishop W, Kennings A. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In: Proc. of the Canadian Conf. on Electrical and Computer Engineering. Vancouver, 2007. 1235−1238.

[7] Bonny T, Henkel J. Huffman-based code compression techniques for embedded processors. ACM Trans. on Design Automation of Electronic Systems, 2010, 15(4): 1−37.

[8] Köppl D, Sadakane K. Lempel-Ziv computation in compressed space (LZ-CICS). In: Proc. of the 2016 Data Compression Conf. (DCC). Snowbird, 2016. 3−12.

[9] Bille P, Ettienne MB, Gagie T, Gørtz IL, Prezza N. Decompressing Lempel-Ziv compressed text. In: Proc. of the 2020 Data Compression Conf. (DCC). Snowbird, 2020. 143−152.

[10] Policriti A, Prezza N. Computing LZ77 in run-compressed space. In: Proc. of the 2016 Data Compression Conf. (DCC). Snowbird, 2016. 23−32.

[11] Zou J, Zhang Z, Xu H. Design of heartbeat invalidation detecting mechanism in triple module redundant multi-machine system. In: Proc. of the 2009 IEEE Circuits and Systems Int'l Conf. on Testing and Diagnosis. Chengdu, 2009. 1−4.

[12] Qiao L, Yang MF, Gu B. An embedded operating system design for the Lunar exploration rover. In: Proc. of the 5th IEEE Conf. on SSIRI. 2011. 160−165.

[13] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. IEEE Trans. on Information Theory, 1978, 24(5): 530−536.

[14] Kosolobov D, Shur AM. Comparison of LZ77-type parsings. Information Processing Letters, 2019, 141(JAN.): 25−29.

[15] Kodituwakku SR, Amarasinghe US. Compression of lossless data compression algrithom for text data. Indian Journal of Computer Sicence and Engineering, 2010, 1(4): 416−425.

[16] Langiu A. On parsing optimality for dictionary-based text compression—the Zip case. Journal of Discrete Algorithms, 2013(1): 65−70.

[17] Arnold R, Bell T. A corpus for the evaluation of lossless compression algorithms. In: Proc. of the ’97 Data Compression Conf. (DCC). Snowbird, 1997. 201−210.

[18] Huang H, Schiopu I, Munteanu A. Deep learning based angular intra-prediction for lossless HEVC video coding. In: Proc. of the 2019 Data Compression Conf. (DCC). Snowbird, 2019. 579−579.

[19] Li B, Akbari M, Liang I, Wang Y. Deep learning-based image compression with Trellis coded quantization. In: Proc. of the 2020 Data Compression Conf. (DCC). Snowbird, 2020. 13−22.

[20] Ororbia AG, Mali A, Wu J. Learned neural iterative decoding for lossy image compression systems. In: Proc. of the 2019 Data Compression Conf. (DCC). Snowbird, 2019. 3−12.

[21] Yamagiwa S, Hayakawa E, Marumo K. Adaptive entropy coding method for stream-based lossless data compression. In: Proc. of the Computing Frontiers Conf. (CF 2020). New York, 2020. 265−268.

[22] Haase P, Matlage S, Kirchhoffer H, *et al*. State-based multi-parameter probability estimation for context-based adaptive binary arithmetic coding. In: Proc. of the 2020 Data Compression Conf. (DCC). Snowbird, 2020. 163−172.

[23] Shaofeng L, Qiao L, Yang MF. Verification of design of memory management module for embedded system based on Coq. In: Proc. of the 11th IEEE Int’l Conf. on Software Engineering and Service Science (ICSESS). Beijing, 2020. 44−47.

[24] Jiang JJ, Qiao L, Yang MF, Yang H, Liu B. Operating system task management requirement layer modeling and verification based on Coq. Ruan Jian Xue Bao/Journal of Software, 2020, 31(8): 2375−2387 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5961.htm [doi: 10.13328/j.cnki.jos.005961]

[25] Yang MF, Zhang G, Zhang W, *et al*. Technique design and realization of the circumlunar return and reentry spacecraft of 3rd phase of Chinese Lunar exploration program. SCIENTIA SINICA Technologica, 2015, 45(2): 111−123 (in Chinese with English abstract).

附中文参考文献:

[24] 姜菁菁, 乔磊, 杨孟飞, 杨桦, 刘波. 基于 Coq 的操作系统任务管理需求层建模及验证. 软件学报, 2020, 31(8): 2375−2387. http://www.jos.org.cn/1000-9825/5961.htm [doi: 10.13328/j.cnki.jos.005961]

[25] 杨孟飞, 张高, 张伍, 等. 探月三期月地高速再入返回飞行器技术设计与实现. 中国科学: 技术科学, 2015, 45(2): 111−123.

邓岸华(1997－), 男, 硕士, 主要研究领域为嵌入式操作系统, 数据压缩.

杨孟飞(1962－), 男, 博士, 研究员, 中国科学院院士, 博士生导师, CCF 高级会员, 主要研究领域为空间飞行器嵌入式系统, 控制系统, 总体技术.

乔磊(1982－), 男, 博士, 研究员, CCF 高级会员, 主要研究领域为操作系统模型设计, 存储管理, 文件系统.