代码缩减技术的研究*)

杨 群 杨献春 许满武

(南京大学软件新技术国家重点实验室,南京大学计算机科学与技术系 南京 210093)

摘 要 随着应用的深入,计算系统对性能的要求越来越高。另一方面,软件规模也越来越大,使得日益庞大的软件与有限的硬件资源之间的矛盾逐渐显现出来。在嵌入式系统、移动计算以及实时控制系统中,这个矛盾尤其突出。如何减少代码、提高代码的效率,成为近年来学术界和产业界关注的问题,许多组织和机构正围绕着此论题开展广泛而深入的研究。本文介绍代码缩减(code-size reduction)的研究背景,以及两种主要的代码缩减方法——代码压缩(code compression)和代码紧缩(code compaction);着重讨论代码紧缩技术,包括:代码紧缩的主要方法、各个方法的特点及其中的关键技术;分析代码紧缩技术尚存在的问题和面临的挑战,并对代码紧缩技术的未来发展趋势做了一些预测。

关键词 代码缩减,代码压缩,代码紧缩,编译器优化技术,代码因子提取,跨越跳转,过程抽取

Research on Code Reduction

YANG Qun YANG Xian-Chun XU Man-Wu

(State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210093)

Abstract Code-size reduction is now a hot and active research activity in such areas as networking, embedded system, and mobile computing. This is because the reduced-size code can improve the system performance without increasing the total system costs. In this paper, we firstly give an introduction of the motivation, concept, and two main strategies of code-size reduction. We then concentrate on code compaction technique, presenting various methods of code compaction, and summarizing their features and key techniques. Finally, we discuss the shortages of the existent compaction methods, with a prospect about the future work.

Keywords Embedded system, Code-size reduction, Code compression, Code compaction, Compiler optimization, Code factoring, Cross-jumping, Procedural abstraction

1 代码缩减研究的背景

代码缩减(code-size reduction)指的是对代码进行的不改变其本来功能的变换,变换的目的是为了减少代码。代码缩减最早可追溯到汇编语言时代。早期的机器价格昂贵、内存容量小,软件开发人员为了充分利用宝贵的硬件资源,使用代码缩减技术使代码以最少的代码完成最多的功能。之后,随着硬件技术的发展,CPU 速度大大提高、内存容量急剧增大,代码大小一度不再是制约软件开发的因素,开发人员更关注的是软件的功能是否强大、程序正确与否、代码的重用性好不好等问题,软件越来越复杂、越来越庞大。然而,近年来,随着网络技术的迅猛发展、嵌入式系统的渗透、无线网络和移动计算的普及以及实时控制系统的广泛应用,代码大小问题重新引起了人们的重视。如何减小代码大小、提高代码的效率,成为近年来学术界和产业界关注的问题[1]。

首先,在嵌入式系统、无线网络应用领域,由价格、重量、能源消耗等因素决定,硬件资源尤其是内存仍然宝贵。采用减小内存容量、缩减缓存大小等方法降低成本往往导致系统性能下降。如何设计占用内存较小的软件因而成为一个关键问题。开发人员选取恰当的算法、精心设计合适的数据结构等方法可以减小运行期软件对内存的占用量,但源代码却仍然可能占用很大内存。减小代码大小,一方面可以腾出有限的内存空间以满足其他应用对内存的需求,降低系统的成本;另一方面,减小代码大小、缩小内存空间意味着降低能耗,这也是嵌入式系统和移动计算需要解决的一个重要问题。

其次,在 Internet 环境下,大量代码的传输和执行使本来就有限的网络带宽更加拥挤,减小代码大小可以更有效地利用网络资源。事实上,已有一些压缩方法用于减小代码大小,如将 Java 程序的 class 文件打包成 jar 格式,或使用一些通用工具软件如 gzip 和 gunzip 等打包、解包程序。但这些方法对代码的缩减程度都有限[2],为进一步减轻网络负载,需要更有效的代码缩减技术。

自 20 世纪 60 年代软件工程概念的提出、80 年代面向对象软件开发技术兴起至今,工程化的软件开发思想深入人心。人们注重软件产品的开发周期以及软件的复杂度、可靠性、重用性等。在此思想指导下,各种有利于软件快速、有效开发的技术和工具不断涌现,如现在软件开发中广泛使用构件、模板以及由第三方提供代码库等技术。为了达到较高的代码复用性,这些构件或代码库要考虑各种可能出现的情况,并进行编码。而实际上,使用代码库或构件的程序,需要的可能只是其中一部分代码,代码库或构件给程序带来了大量的冗余代码,造成软件代码量过大。因此,在内存容量不足的嵌入式系统、移动系统等领域,软件开发技术面临新的挑战,代码缩减技术也变得越来越重要,成为当前一个活跃的研究领域。

*)本文受国家自然科学基金(60273035)、江苏省科技攻关项目(BE2003064)资助。杨 群 博士研究生,主要研究领域为软件方法学和新型程序设计;杨献春 教授,主要研究领域为软件方法学和自主计算;许满武 教授、博士生导师,主要研究领域为软件方法学和新型程序设计。

本文结构如下：第2部分首先介绍影响代码大小的因素，然后介绍代码缩减的两种主要方法和研究状况；第3部分着重讨论代码缩减当前研究的主流研究方向——代码紧缩技术，包括代码紧缩的各种方法及其中的关键技术；第4部分分析代码紧缩技术目前存在的问题和未来的研究方向；最后是结语。

2 代码缩减的方法及其研究现状

2.1 影响代码大小的因素[3]

影响程序代码大小的因素有3个：

1)指令体系。程序可执行代码由执行程序的指令数和指令编码的位数两者共同决定，而指令数和编码指令所需的位数则随指令体系的不同而不同。因此，不同的指令体系产生的可执行代码的大小是不同的。

指令体系与指令数之间、指令编码所需的位数之间的关系是：指令集包含的特殊指令越多，程序需要的指令数越少；指令集越大，编码指令所需要的位数越多。举个例子：一个C语言的程序，编译成ARM体系(Advanced RISC Machine)，生成指令3500条；编译成ARM-Thumb体系(Thumb是为了缩减代码而设计的16位ARM指令集)，则生成指令5000条。表面看上去采用ARM指令体系生成的代码数多于ARM-Thumb，但实际上，由于ARM包含的特殊指令多，指令体系大，编码指令所需要的位长为32bit，而ARM-Thumb的位长为16。因此，最终ARM生成代码为3500×32bit，而ARM-Thumb生成代码5000×16bit，ARM的代码量比ARM-Thumb还大30%。

2)指令编码。指令编码是指采用特定数量和顺序的位序列表示指令。其基本思想是：把给定的程序看作一个输入的字符序列，统计字符序列中的每个字符出现的频率，对出现频率高的字符用短的编码表示，对出现频率低的字符用相对较长的编码表示，最后输出缩减了的字符序列。指令编码的方法有多种，如Huffman编码[4]、LZW算法[5]等。对指令进行重编码即代码压缩，是数据压缩的一种。采用好的编码方法可以大大缩减程序代码大小。

3)程序编译。程序编译是把编写的程序作为输入，生成用目标语言编写的等价程序的过程。在编译过程中，编译器除了对代码进行从输入的源程序到机器语言的翻译外，还要在编译过程的不同阶段做各种优化，提高目标代码的执行效率、缩小目标代码等。例如，在中间代码生成过程中，编译器常使用常量传播、常量合并、冗余消除、长度缩减、归纳变量优化、过程调用去除等代码优化方法；在目标代码生成过程中，编译器要进行指令选取、指令调度和寄存器分配，这个阶段则进行一些提高代码性能的优化，如最优寄存器分配(optimal register allocation)[6]、基于程序梗概的缓冲区优化(program profile-based cache optimization)[7]、窥孔优化(peephole optimization)、代码联合定位(code co-location)等。编译器在各个阶段可选择的优化方法很多，它们的目标不同，有的为了消除影响后继分析的不良因素，有的为了减小代码大小，还有的则是为了提高代码执行效率，因此，要缩减代码大小，必须选取恰当的编译优化方法和编译过程。

三种影响程序大小的因素中，第一种和机器的硬件设计有关，本文对此不做介绍。第二种和第三种分别是代码压缩和代码紧缩技术，是本文下面要介绍的。

2.2 代码缩减的两种主要方法

如前所述，代码缩减是指用某种格式存储代码，使之占用更小的空间。代码缩减主要有两种方法：代码压缩(code compression)和代码紧缩(code compaction)。两种方法都是为减小代码大小而对代码进行变换的代码缩减技术，但采用的方法和得到的结果形式不同。

代码压缩方法通过对代码进行重编码，减小输出字符序列，可用于代码的机器码形式。常用的编码算法有：Huffman编码[4]、Arithmetic编码[8]、Dictionary-based编码[9]等。代码压缩是建立在输入字符序列的静态统计信息基础之上的，因而大多数代码压缩方法包含模型和编码器/解码器两个部分。代码压缩得到的结果是不能直接执行的压缩代码，执行前要进行解码，一般需要硬件支持。

代码紧缩方法是对代码进行各种变换，减小代码大小。代码紧缩得到的是可直接执行的代码，因而不需要执行前的解码，也就不需要额外的硬件支持。代码紧缩可用于源代码、中间代码(IR, Intermediate Representative)和机器代码等各种形式。可采用的方法有：1)传统的编译器优化方法，如公共子表达式(common sub-expression)提取，冗余代码、不可达代码和死代码的去除，表达式简化，常量合并(constant folding)，常量传播(constant propagation)等。2)代码因子提取(code factoring)[1]：把公共代码提取出来以减小代码大小的方法。又分为代码因子本地变换(local factoring)、跨越跳转(cross-jumping)、过程式抽取(procedural abstraction)三种。3)基于软件的更高抽象级别——软件体系结构的方法，下文将介绍此方法，此处不详述。

代码压缩和代码紧缩最重要的区别是能否直接执行，执行时是否需要解码。从节省空间的角度考虑，一般来说，使用代码压缩技术代码的减少量大于代码紧缩技术，但执行时其解压缩需要的内存也更多；从执行时间来看，使用代码压缩技术需要在代码执行前进行解压缩，带来额外的时间耗费；使用代码紧缩技术不需要在执行前进行代码解压缩，但执行时过程调用(在过程式抽取和跨越跳转两种方法中存在)会延长执行时间(当然这可通过一些其他方法进行消除)。在实际应用中，两种方法兼用能达到更好的代码缩减效果。比如，先采用代码紧缩技术去除代码中的冗余代码，然后进行代码压缩以进一步减小代码大小。

2.3 代码缩减两种方法的研究状况

代码可看作是一种数据，但又不同于一般的数据。代码的这种特点决定代码缩减两种主要技术的不同发展状况。

一方面，代码是一种数据，因而代码压缩可看作数据压缩的一种，很多数据压缩技术可直接应用。另一方面，代码又不同于一般的数据，代码是具有一定的语法和语义的数据，表现在程序中就是具有一定的结构和含义，例如，程序代码具有的类型信息、调用关系的层次结构等。代码的这种特性可以用来帮助分析数据(代码)之间的关系，获取更多的信息以缩减代码。近年来发展起来的代码紧缩技术也正是基于此，通过对程序本身进行分析，去除不必要的代码。

数据压缩的研究历史悠久，几乎和信息技术同步，技术比

[1] 本文中代码因子提取的含义与文[12]相同，包括代码因子局部变换、跨越跳转、过程提取三种方法。有些文献中代码因子提取仅指代码因子局部变换。

较成熟，相关的成果和文献资料也很多，目前不是代码缩减研究的主流，因而本文不打算对此再做讨论。文[10]中详细介绍并对比了10种代码压缩方法，有关的介绍和评述可参考之。代码紧缩技术正处于积极研究之中，涉及面广，有关的方法和算法复杂繁多，但相关的讨论和文献在国内还很欠缺，因而下面就对该技术的研究状况和当前的进展进行比较全面的介绍，并在此基础上进行归纳和分析，清晰地展现出有关的研究思路和方法。

3 代码紧缩

如前所述，代码紧缩是为减小代码大小而对代码进行的、结果仍然是可执行代码的变换。其主要方法包括：1)传统的编译器优化方法；2)代码因子提取(code factoring)。本文中代码因子提取泛指各种识别和抽取程序中重复代码的技术，又分为代码因子局部变换(local factoring transformation)、跨越跳转(cross-jumping)和过程式提取(procedural abstraction)3种；3)从软件体系结构出发的代码紧缩方法。

3.1 编译器优化(compiler optimization)

编译器优化的研究由来已久，最早可追溯到1952年[11]。多年来人们研究出大量编译器优化技术，归纳起来可分为两类：一类是为了提高代码的执行效率，一类是为了缩减代码大小。两类编译器优化方法目的不同，带来的效果不同，适用的场合也不同。例如，过程内嵌(procedure inling)采用过程体代替过程调用，可以提高代码的性能，但它会大大增加代码量，因此不适用于代码紧缩。

对代码紧缩有效的编译器优化方法包括：冗余消除、不可达代码消除、死代码消除和强度降低(strength reduction)。冗余消除是指消除代码中重复出现、执行结果相同的代码。不可达代码消除则是消除那些控制流上没有路径可达的节点，如分支语句中条件不成立时不执行的语句。如果某段代码执行出来的结果在程序的其他任何地方都用不到，这样的代码就是死代码。程序中不可达代码的消除有时会使一些代码从"部分"死代码变成"完全"死代码。强度降低是用代价比较小、执行速度快的指令代替代价高的指令，比如乘法指令可以用偏移和加指令来代替。强度降低并不总意味着代码长度减小，但对于代码紧缩来说，强度降低同时也应该是代码大小的减小。要执行编译器的上述优化，控制流分析、过程间常量传播、过程间寄存器生命期分析这几项技术是至关重要的[12]。

对于采用多种优化技术的编译过程来说，各种方法应用的顺序不同，得到的结果有很大差异。Cooper[13]提出采用遗传算法搜索得到程序的最佳优化顺序，他提供的测试数据表明，使用经遗传算法得到的优化顺序进行代码缩减，压缩率可以提高40%，运行速度加快26%。

3.2 代码因子提取(code factoring)

程序中重复出现的代码(指令)序列是代码紧缩的一个重要来源。把重复出现的代码序列抽取出来，称为代码因子提取。代码因子提取的一般步骤是：1)查找程序中重复出现的指令序列；2)构造一个代理指令，它取代基本代码块[2]中重复出现的指令序列；3)程序执行时，执行该代理指令。执行该代理指令应保证经代理指令变换的代码和原代码等价。

代码因子提取有3种方法，其中代码因子局部变换方法适用于局部过程，跨越跳转方法适用于过程间，过程式提取方法适用于全局过程。

3.2.1 代码因子局部变换(local factoring transformation)

代码因子的局部变换是将两个或多个基本代码块中重复出现的相同代码序列移到这些代码块的前控制节点(predominator)或后控制节点(post-dominator)中的一种代码变换方法。例如，在N控制的两个基本代码块中，I1和I2指令相同，则I1和I2可以移到N中合并为一条指令，其结果是减少了一条指令。一个具体的例子如图1所示[14]。

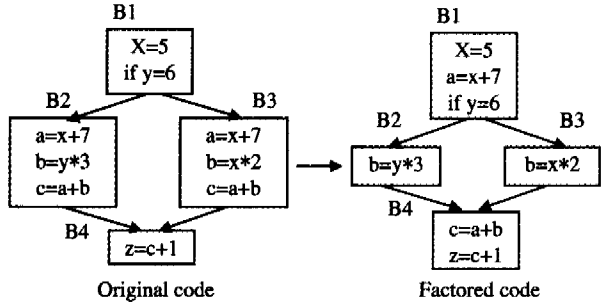

图1 代码因子提取例

指令序列必须满足如下条件才能被移动：1)被移动的指令序列使用相同的寄存器；2)被移动指令序列与其前后指令序列之间没有任何依赖关系。事实上，许多因素都会影响代码因子局部变换结果，包括寄存器分配，在代码因子局部变换之前进行的其他代码优化措施，如公共子表达式消除、强度减小等。

代码因子的局部变换是面向过程的，指令移动限于控制流图CFG(control flow-graph)中同一过程的基本块，或者说是流图中同一节点的并行的几个分支节点。下面介绍的跨越跳转方法对此进行了扩展，使得指令的移动可以在过程间进行。

3.2.2 跨越跳转(cross-jumping)

跨越跳转方法是把基本代码块的公共后缀抽取出来的代码变换方法。具体来说就是，如果有多个过程，其尾部指令序列相同，则只保留其中一个过程中的指令序列，而把其他过程中相同的指令序列部分代之为一个跨越跳转指令。跨越跳转限定只有过程后缀才可以合并，所以又称为尾合并(tail merging)。例如，有过程P1、P2、P3，它们的后缀部分代码相同，则可保留其中一个过程——假设为P1的代码后缀，而将其他两个过程P2、P3的相同后缀部分去掉，代之以一个跳转指令。

3.2.3 过程式抽取(procedural abstraction)

过程式抽取是针对整个程序进行的代码紧缩。可以把它看做是内联(inline)函数的逆操作，其基本思想是：将程序(包括函数库)中多次出现的相同代码序列抽取出来，用一个过程表示，原代码块中的指令序列则代之为对此过程的调用。过程式抽取要解决两个问题：1)如何查找重复出现的代码序列，后面将会看到，重复出现的代码序列可分为完全相同(操作符和操作数都相同)和相似(操作符相同，操作数不同)两类；2)如何评价查找到的代码序列是否该抽取为一个过程。

Marks[15]1980年首次提出过程式抽取方法。Marks把

[2] 一个最大的线性代码序列称为基本代码块。线性代码序列指没有跳出跳进语句的代码序列。

程序中指令的操作符和操作数都看作字符串，整个程序看作一个长字符串，使用字符串匹配方法识别和查找重复的代码序列。他的重复代码的查找和匹配方法是基于表结构的。具体来说就是：首先建立一个哈希表，对每个操作符建立一个链表记录操作符在程序中的位置，然后对哈希表中的操作符链上的操作符以配对迭代(iterative paring)的方式进行匹配检测，得到所有匹配的指令序列在程序中的位置和长度，并把这些匹配的指令序列加入到表中、成组，组中每个指令小序列作为抽取过程的候选。匹配检测完成后，对每组中的候选指令序列根据其长度和在程序中出现的次数决定是否抽取为过程。被选中为抽取过的指令序列插入到程序的尾部，原程序中的指令序列则插入一个伪指令代替抽取过程，在执行伪指令时调用抽取过程。Marks 得到的效果很不错，运用在机器码形式上，代码压缩率达到 50%；但压缩代码的执行速度比压缩前下降了 15%，这是由于执行时需要解析伪指令造成的。

Fraser 等[16] 1984 年提出一个改进的算法。与 Marks 基于表结构的方法不同，Fraser 等借鉴文本压缩使用的后缀树(suffix tree)来进行重复字符串的查找和匹配：通过构造后缀树，得到后缀树中出现多次的子串，这些子串就是子程序的候选。采用后缀树法减少了字符串子串搜索和匹配所需要的时间，因此自 Fraser 之后，后缀树就被广泛应用于过程式抽取中。图 2 是后缀树的一个具体例子。

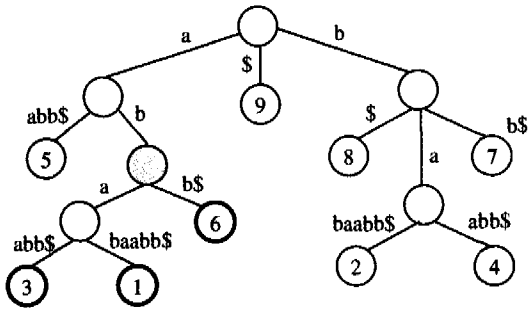

图 2　后缀树一例

上述的过程式抽取方法都要求代码绝对相等。实际上，程序中出现更多的是相似代码，要求代码完全相同，限制了过程式抽取发挥其作用。文[17]的实验甚至表明，现代编译器生成的代码不适合要求代码段完全相同的过程式抽取方法。针对这个问题，Barker[18]、Zastre[17]、Cooper[19]等分别提出了不同的解决办法。Barker 提出了参数化后缀树法，Zastre 提出了参数化过程式抽取(parameterized procedural abstraction)法。Cooper 等提出的改进方法是另一种思路，他们在 Fraser 字符匹配方法的基础上，引入寄存器改名，实现代码块的相似匹配，以允许指令的操作数不同。在文[19]中，Cooper 等人介绍，寄存器地址以使用或定义过的同一个寄存器的相对地址来表示，寄存器值不相同的代码序列被识别为是相同的(语义相同)；在分析寄存器的生存期后，对用寄存器改名，使相似代码变成相同代码。Saumya 等[20]也是采用寄存器改名的方法，但不是采用传统的把代码看作线性序列的方法，而是用分化区的方法来区分代码是否相似，即从可执行代码中产生控制流图，用图匹配方法查找相似区域，然后对相似代码应用寄存器改名法使之变成相同代码。Johan Runeson[21]则将此方法应用到中间代码级上，因为过程式抽取在寄存器分配之前完成，所以避免了寄存器改名问题。

3.2.4　代码因子提取的关键技术

上述的代码因子提取方法都建立在重复代码序列的查找和匹配的基础之上。公共代码的查找和匹配技术是影响代码因子提取压缩和执行效率的关键。有关公共子表达式的计算和模式匹配的算法很多，下面将对此进行简单介绍，更详细的讨论可参见文[22,23]。

1)公共子串的查找和选取

如前所述，查找重复代码序列即计算出代码中出现多次的子串，这些子串又称为公共子串。查找子串算法的时间、空间消耗以及子串的选取方法都直接影响到代码紧缩最后得到的压缩率和代码紧缩过程中的时空消耗。目前查找和选取子串用得最多的是后缀树法。后缀树法通过构造字符串的后缀树，得到出现多次的子串和与非叶子节点相连的边的权值。出现多次的子串即为公共子串，权值越大的边代表的公共子串节省的代码越多。后缀树的标准算法由 Weiner[24] 给出，McCreight[25] 对此进行了改进，Ukkonen[26] 进一步细化成在线的(即不需要事先建立数据的索引)构造算法，三者的空间和时间耗费均为线性 $O(n)$。还有一种与后缀树类似的方法，称为后缀数组法(suffix array)，它采用数组作为存储结构，其时空耗费也是线性的，但空间耗费仅为后缀树法的一半。因为采用数组作为存储结构，因此此法适用于简单的子串操作，对复杂的大字符串的操作时间消耗要大于后缀树法。另外还有一种称为 Suffix Cactus[27] 的方法，性能介于后缀树和后缀数组之间：对复杂子串的操作比后缀数组要快，比后缀树法节省空间。

2)子串匹配

传统的代码因子提取建立在代码完全相同的基础上，要求指令序列的各指令操作符和操作数都必须相同，因此要求子串相同匹配。为扩大代码因子提取的适用范围，近年来的一些研究都集中于代码相似匹配，只要求指令序列的各指令操作符相同，操作数可以不同，也就是要求子串相似匹配(approximate string matching)。

最早的字符串匹配方法可追溯到 Knuth-Morris-Pratt 的 KMP 算法[28] 和 Boyer-Moore 的 BM 算法[29]。KMP 是第一个线性时间和空间耗费的字符串匹配算法，BM 算法在对长字符串的操作上性能比 KMP 算法更优，通常被认为是最有效的字符串匹配算法。这些算法都是通过字符的比对来进行字符匹配的。字符匹配最重要的应用之一就是最长公共子串(longest common sub-expression)的计算。后缀树是一种支持快速搜索和匹配的数据结构，提供所有子串的直接访问和子串在串中出现的位置信息，可起到对字符串索引的作用。通过在线构建后缀树，Ukkonen[26] 的最长公共子串的计算算法在性能和复杂度方面都得到了改善。Fraser 进一步把后缀树应用到代码紧缩中，打开了代码紧缩应用后缀树的思路。此后，基于后缀树的公共子串搜索和匹配就在代码紧缩中得到了广泛的应用。

随着代码紧缩技术研究的深入，采用字符串的近似匹配成为必然。Baker 首先在代码紧缩中提出并使用参数化后缀树法进行字符串的近似匹配。Baker 的方法中符号分为通用符号和参数符号两种，字符串由 $(\Sigma \cup \Pi)^*$ 构建，称为参数化字符串 p-串(p-strings)，其中，Σ 是通用字符集，Π 是参数符号集。Baker 定义基于 p-串的后缀树为参数化后缀树，他还定义同一个整数取代参数化符号后相同的 p-串为 p-匹配。举个例子：假设 $\Sigma=\{a,b,c\}$，$\Pi=\{x,y\}$，S=axbxcxc，T = ay-

bycyc，则 So = a0b2c2c，To = a0b2c2c，显然，So= To。通过建立参数化后缀树，子串计算和匹配都在参数化的字符串上进行，参数化后缀树很适合相似匹配。Baker 使用参数化近似匹配得到的代码紧缩效率很不错，代码减少量比用传统的过程式抽取大大提高，而且时间耗费也是线性的。以后，Baker 又改进了 BM 算法[30]，使之适用于参数化后缀树。

3.3 基于软件体系结构的代码紧缩方法

上述的代码紧缩方法都是建立在程序分析的基础之上的，在源代码可以获得的情况下是可行的。但在当前流行的软件开发中，为了提高代码重用性，开发人员常常采用构件技术装配程序，或在程序中直接调用第三方开发的库函数。这些构件或库要么不易更改，要么因为无法获得源代码而无法更改，导致代码紧缩无法执行。而且第三方提供的库或构件出于通用性的考虑，往往代码量非常大，导致矛盾更加突出。可以说，这是代码紧缩技术在现代软件开发方式下面临的新挑战。针对这种情况，Chris Lüer 等[31]提出了基于软件体系结构的代码紧缩。

在文[31]中，Chris Lüer 等提出了基于软件体系结构的代码紧缩方法的一些思路，他们也正在研究开发一个应用于 Java 程序的工具，提供给应用开发人员使用。这样的工具可以对代码库或构件进行重新配置，包括：1）去除不使用的代码和其他的冗余信息；2）对代码库或构件进行性能检测，并对其中占用内存多的实现代码用效率更高的实现进行代换。由于是针对目前普遍存在的问题，Chris Lüer 提出的这些设想非常实际，他们实现的将是更高层次的代码紧缩方法。可以说，基于软件体系结构的代码紧缩相对于传统的代码紧缩技术来说，将更实用，是代码紧缩技术未来很重要的一个研究方向。

4 代码紧缩技术存在的问题和未来的发展

前面回顾总结了代码紧缩技术的三种方法和其中涉及到的问题和关键技术。代码紧缩技术是个需要考虑多种因素的问题，包括：代码级别是源代码、中间代码还是机器代码？在寄存器分配前还是寄存器分配之后？最终代码是机器码（例如 C 语言）还是字节码（例如 Java 语言）？在不同场合下需要采用不同的方法，有时可能还要运用多种方法。这就造成各种方法使用起来性能不稳定，在某些情况下性能非常好的方法换到另一个环境可能完全是另外一回事。如 Marks 的过程式抽取方法在机器码级上应用，代码缩减率达到 50%，而在源代码级上仅为 1%～2%，使得本来就纷繁杂乱的代码压缩技术变得更加难以选择。

另外，有些代码压缩技术在实际程序运行时会造成一定程度的性能下降，如过程式抽取方法中，过程调用前后寄存器切换的耗费、过程调用和返回的耗费都会影响执行效率。有些因素引起的性能下降可以采取一定措施使之得到缓解，但有的就很难完全消除。因此，是否采用代码紧缩技术也要根据实际情况在多种性能指标中权衡、选取。这正像 ARPAD[10] 所说：没有一个任何情况下都“最好”的代码紧缩方法。

最后，仅仅依靠代码紧缩技术减小代码量可能还不足以适应当前应用对软件功能需求越来越高的趋势，如何从根本上改变目前软件发展滞后于实际需要的问题，我们认为，这不是代码缩减技术本身能解决的，未来还需要多个领域的合作研究才能很好地解决。我们预测可从如下方面进行研究：

1）从软件的高层次分析着手，将代码紧缩技术的三种方法结合，根据应用的特点进行分析，灵活地采用恰当的方法进行更有效的代码缩减；

2）传统的软件开发方式已经不适应所谓的“小内存”系统的软件开发，因此在这些领域可能出现一些与传统软件开发迥异的思想和方法；

3）改变目前软件开发受硬件资源限制这种状况，使“小内存”系统开发出来的软件功能可以更加复杂和强大。

结语 代码缩减因为目前计算机技术的进步和应用的深入而成为当前研究的一个热点问题，它也是一个涉及面颇广的问题，包括数据压缩、指令体系设计、编译器设计、算法设计、程序分析等领域，已经研究出的成果很多，但距离理想的实用效果还有很大距离。为达到更好的效果，未来还需做更多工作。本文介绍了代码缩减的有关问题，包括代码缩减的概念、主要的两种缩减方法——代码压缩和代码紧缩；重点介绍了代码紧缩技术：代码紧缩涉及的关键技术、目前的进展。最后，我们对代码紧缩技术的未来发展趋势做了一些预测，希望能对国内未来在该领域的研究起到一定的推动作用。

参 考 文 献

1 De Sutter B, De Bosschere K. Software Techniques for Program Compaction. Communications of the ACM, 2003, 46(8): 33～34
2 Clausen L, Schultz U, Consel C, et al. Muller. Java Bytecode Compression for Low-end Embedded Systems. ACM TOPLAS, 2000, 22(3): 471～489
3 van de Wiel R, et al. Code Compaction: Reducing Memory Cost of Embedded Software. Philips white paper, 2001. http://www.extra.research.philips.com
4 Nelson M, Gailly J-L. The Data Compression Book. 2nd ed. M&T Books, 1995
5 Welch T A. A technique for high-performance data compression. IEEE Computer, 1984, 17(6): 8～19
6 Fox A, et al. A Survey of General and Architecture-Specific Compiler Optimization Techniques
7 Bacon D F, et al. Compiler Transformation for High-Perfomance Computing
8 Bell T C, Witten I H, Cleary J G. Text Compression. Advanced Reference Series. Englewood Cliffs: Prentice Hall, 1990
9 Ernst J, Evans W, Fraser C W, et al. Code compression. In: PLDI '97. ACM, New York, 1997. 358～365
10 Beszedes A, et al. Survey of Code-size Reduction Methods. ACM Computing Surveys, 2003, 35(3): 223～267
11 Huffman D A. A method for construction of minimum redundancy codes. In: Proc. of the IEEE, 1952, 40: 1098～1101
12 Debray S K, Evans W, Muth R, et al. Compiler techniques for code compacion. ACM Trans Prog Lang Syst, 2000, 22(2): 378～415
13 Cooper K D, Scheilke P J, Subramanian D. Optimizing for reduced code space using genetic algorithms. In: ACM SIGPLAN Proc Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99), 1999, 34: 1～9
14 Johnson N, Mycroft A. Pattern and Approximate-Pattern Matching for Program Compaction, 2003, 7
15 Marks B. Compilation to compact code. IBM Journal of Research and Development 1980, 22(6): 684～691
16 Fraser C W, Myers E W, Wendt A L. Analyzing and compressing assembly code. In: Proc. ACM SIGPLAN Symp. Compiler Construction, 1984, 19: 117～121
17 Zastre M. Compacting Object Code via Parameterized Procedural Abstraction: [Master's thesis]. University of Victoria, 1995
18 Baker B S. A theory of parameterized pattern matching: Algorithms and applications. In: Proc ACM Symposium on Theory of Computing, New York: ACM Press, 1993. 71～80
19 Cooper K, McIntosh N. Enhanced code compression for embedded RISC processors. In: Proc. PLDI, 1999. 139～149
20 Debray S K, et al. Compiler Techniques for Code Compaction. ACM TOPLAS, 2000
21 Runeson J. Code Compression through Procedural Abstraction before Register Allocation. Uppsala Master's Theses in Computing Science 165, 2000-03-08
22 Crochemore M, Lecroq T. Pattern matching and text compression algorithms

23 Navarro G. A Guided Tour to Approximate String Matching. ACM Computing Surveys, 2001, 33(1): 31～88
24 Weiner P. Linear pattern matching algorithms. In: Proc 14th IEEE Annual Symp on Switching and Automata Theory 1973. 1～11
25 McCreight E. A space-economical suffix tree construction algorithm. ACM, 1976, 23(2): 262～272
26 Ukkonen E. Constructing suffix trees on-line in linear time. van Leeuwen J, Ed. Algorithms, Software, Architecture. Elsevier, 1992, 1: 484～492
27 Karkkainen J. Suffix cactus: A cross between suffix tree and suffix array. In: Proc Annual Symp on Combinatorial Pattern Matching (CPM'95) (1995), vol. 937 of Lecture Notes in Computer Science, Springer-Verlag, 1995. 191～204
28 Knuth D, Morris J, Pratt V. Fast pattern matching in strings. SIAM Jour Computing, 1977(6): 323～350
29 Boyer R S, Moore J S. A fast string searching algorithm. Comm. ACM, 1997, 20(10): 762～772
30 Baker B S. Parameterized pattern matching by Boyer-Moore-type algorithms. In: Proc. of the 6th ACM-SIAM Annual Symp on Discrete Algorithms, San Francisco, California, 1995. 541～550
31 Lüer C, van der Hoek A. Architecture-Based Program Compaction
32 Noble J, Weir C. Small Memory Software. Pearson Education, Harlow, 2001

(上接第 236 页)

稀疏地表示自然图像，因而随机噪声与自然图像的 contourlet 系数也具有更好的可分离特性。只需选取一个合适的阈值，简单的 contourlet 阈值去噪方法就可以获得比复杂的小波去噪算法更好的去噪效果。此外，M. N. Do 等人还将传统的隐性 Markov 树模型(Hidden Markov Tree, HMT)与 contourlet 结合用于图像去噪，也收到了很好的效果。

3.4 图像检索

利用 contourlet 的特征提取能力可以很容易地实现各种类型的图像/视频检索方案。一般来说，基于 contourlet 的检索方案可以表述为如图 5 的形式。首先将待检索的查询图像变换至 contourlet 域中，随后求取 contourlet 系数的特征参数，再按照匹配准则对图像特征库中的图像特征逐次进行比对。若满足阈值条件，则将该幅图像作为检索结果输出。

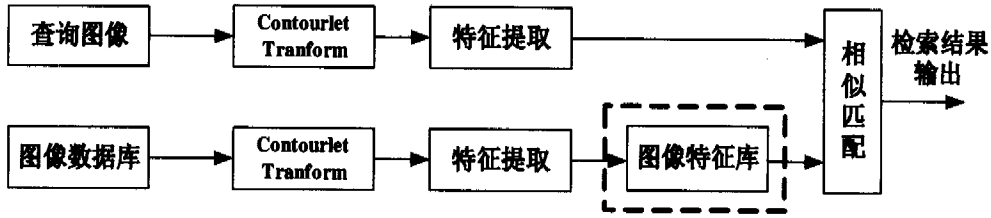

图 5 contourlet 图像检索系统原理框图

在基于 contourlet 的图像数据库检索中，如何设定有效的 contourlet 域特征参数是检索系统性能优劣的关键之处。Duncan Po 和 M. N. Do 使用 HMT 模型求取 contourlet 域中的特征参数，使用 Monte-Carlo 算法估计两个 HMT 模型之间的 K-L 距离作为查询图像与图像数据库图像之间相似度的度量，其结果令人满意。

4 研究展望

从多尺度几何分析理论的提出至今不过短短几年时间，其理论和应用研究便得到了广泛重视，积累了一定的研究成果，而 contourlet 理论作为最新的 MGA 工具更是目前该领域研究的焦点之一。contourlet 能够更为稀疏地表示自然场景图像，比小波变换具有更好的非线性逼近性能；相对于 ridgelet、curvelet 等其它多尺度几何分析工具，contourlet 变换具有更少的冗余度和更好的逼近性能，因此 contourlet 在去噪、压缩、特征提取等传统的图像处理领域具有天然的优势。然而对于 contourlet 变换，其理论框架和实际应用还有待进一步探索和完善。经过分析，我们认为目前还有如下方面的问题值得做进一步研究：1) contourlet 的构造方法研究；2) 高维 contourlet 变换的构造；3) 快速 contourlet 算法的研究；4) 自适应 contourlet 基的构建；5) 多 contourlet 理论框架的建立；6) 在传统图像处理方面的应用；7) 在图像编码和图像/视频检索等的研究；8) 与神经网络和模式识别相结合的研究。

目前国内外在 contourlet 方面的研究才刚刚起步，见诸文字的相关研究还较少。我们相信随着对 contourlet 理论和应用研究的充分展开，contourlet 变换将具有更加光明的应用前景。

参 考 文 献

1 Candes E. Ridgelets: Theory and Applications: [Ph. D. Thesis]. Department of Statistics, Stanford University, 1998
2 Donoho D L, Duncan M R. Digital curvelet transform: Strategy, implementation [C]. In: Proceeding of SPIE, 2000, 4056: 12～29
3 Starck J L, Candes E J, Donoho D L. The curvelet transform for image denosing [J]. IEEE Trans on Image Processing, 2002, 21 (11): 131～141
4 Pennec E L, Mallat S. Image compression with geometrical wavelets [C]. In: Proc. of ICIP '2000, Vancouver, Canada, 2000. 661～664
5 Do M N, Vetterli M. Contourlets: a new directional multiresolution image representation [J]. Signal, Systems and Computers, 2002, 1: 497～501
6 Burt P J, Adelson E H. The laplacian pyramid as a compact image code [J]. IEEE Trans on Communication, 1983, 31(4): 532～540
7 Bamberger R, Smith M J. A filter bank for the directional decomposition of image: Theory and design [J]. IEEE Trans. Signal Processing, 1992, 40(4): 882～893
8 焦李成，谭山. 图像的多尺度几何分析：回顾和展望[J]. 电子学报，2003, 31(12A): 1975～1981
9 Po D D, Do M M. Directional multiscale modeling of images using the contourlet transform [C]. In: IEEE workshop on Statistical Signal Processing, 2003. 262～265
10 Do M N, Vetterli M. Contourlets, in Beyond Wavelets. New York: Academic Press, 2002
11 Aiazzi B, Alparone L. Pyramid-based multiresolution adaptive filters for additive multiplicative image noise [J]. IEEE Trans on circuits and systems II, 2001, 45(8): 1092～1097
12 Eslami R, Radha H. On low bit-rate coding using the contourlet transform [J]. Signal, Systems and Computers, 2003, 2: 1524～1528