pT-树：高速缓存优化的主存数据库索引结构

杨朝辉 王立松
（南京航空航天大学信息科学与技术学院 南京 210016）

摘 要 随着主存速度和现代处理器速度之间的差距逐渐扩大，系统对主存的存取访问成为新的瓶颈，Cache 行为对主存数据库系统更加重要。索引技术是主存数据库系统设计的关键部分。在 CST-树的基础上应用预取技术提高查找操作的性能，提出了一种 Cache 优化的索引结构预取 T-树(pT-tree)。pT-树使用预取技术有效地创建比正常数据传输单元更大的索引结点，从而降低了 CST-树的高度，减少了从父亲结点遍历至孩子结点时的 Cache 缺失。实验结果表明，pT 树与 B+-树、T-树、CST-树、CSB+-树相比查找性能有所提高。

关键词 索引结构，pT-树，预取，主存数据库

Prefetching T-Tree: A Cache-optimized Main Memory Database Index Structure

YANG Zhao-hui WANG Li-song
(Information Science & Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Abstract As the speed gap between main memory and modern processors continues to widen, memory access has become the main bottleneck of processing, so the cache behavior becomes more important for main memory database systems (MMDBs). Indexing technique is a key component of MMDBs. We proposed a cache-optimized index—Prefetching T-tree (pT-tree) based on a novel CST-tree index, which applies prefetching to CST-tree to accelerate search operations. pT-tree uses prefetching to effectively create wider nodes which are larger than the natural data transfer size. These wider nodes reduce the height of the CST-Tree, thereby decreasing the number of expensive misses when going from parent to child. The experimental performance study shows that our pT-Trees can provide better search performance than B+-Trees, T-Trees, CST-Trees and Cache Sensitive B+-Trees.

Keywords Index structure, pT-tree, Prefetching, Main memory database

1 引言

随着内存价格不断降低，使得计算机配备超大容量的内存成为现实，可以将系统中经常被访问到的数据，甚至所有的数据放置在内存中。可以预言，在不久的将来，所有的数据都可以常驻在内存中。但是，随着处理器速度与动态随机访问存储器速度、磁盘速度的差距持续增大，系统对主存的访问形成新的瓶颈。在数据库管理系统中，高效使用高速缓存以获取高性能变得尤为重要。现代分级存储体系中，缓存存在于不同层次：典型的两层或更多层次的静态随机访问存储器，既可以作为存储在动态随机访问存储器中的主存内容的高速缓冲存储器，也可作为存储在磁盘中的内容的高速缓冲存储器[1]。历来数据库研究者都集中研究后一种形式的高速缓存(也称为“缓冲池”)。近来的研究表明，即使是在传统的面向磁盘的数据库中，大约有 50%或者更多的执行时间经常浪费在高速缓存缺失。显而易见，对于主存数据库而言，高速缓存的性能是至关重要的。因此，目前的研究再次关注于核心数据库算法和数据结构，注重减少高速缓存缺失，从而提高数据库系统的整体性能。

索引是影响主存数据库性能的关键因子之一，并且在单值查询、范围查询、嵌套循环联接等操作中非常有用[2]。这些高层次操作可以被分解为两种低层次访问类型：查找指定关键字和扫描索引的一部分[1]。对于单值查询与嵌套循环联接，查找时间是其关键因素。现有的索引结构中，T-树[3]被广泛地用来作为主存数据库的索引结构，设计重点只局限于降低索引的空间占用率，T-树的高速缓存性能低于 B+-树[4]。在 Cache 敏感性方面，尽管 B+-树优于 T-树，但是 B+-树的高速缓存性能仍然不尽人意，导致了高速缓存性能更好的 CSS-树[5]和 CSB+-树[6]的产生。

本文主要在一种新的 Cache 敏感树——CST-树[7]的基础上，使用预取技术有效地创建比正常数据传输单元更大的索引结点来提高查找操作的性能，提出了一种 Cache 优化的索引结构——预取 T-树。本文第 2 节讨论相关的研究工作；第 3 节详细介绍 pT-树的构造和相关分析；第 4 节展示实验结果；最后是结束语。

2 相关研究

2.1 高速缓存 Cache

高速缓冲存储器 Cache[8]是位于中央处理器与主存之间

到稿日期：2010-11-29 返修日期：2011-02-26
杨朝辉(1984—)，男，硕士生，主要研究方向为数据库系统及应用，E-mail：yangzh0315@163.com；王立松(1969—)，男，博士，副教授，主要研究方向为数据库技术、安全数据库，E-mail：wangls@nuaa.edu.cn(通信作者)。

的容量小、交换速度快的静态随机访问存储器，可以将正在执行的指令地址附近的一部分指令或者数据从主存动态随机访问存储器调入高速缓冲存储器，供 CPU 在一段时间内使用。一个高速缓存是 3 个参数决定其性能的：高速缓存容量、缓存块大小、关联性，其中缓存块是高速缓存与主存之间的基本的数据传输单元。当所需数据或指令在高速缓存中时，它们很快就被访问到。否则，就会引起一次高速缓存缺失，必须从低一级高速缓存或者主存中装入相应的缓存块。因此，内存数据库索引结构的高速缓存优化设计目标就是在操作索引时减少高速缓存失配次数，在高速缓存容量有限的情况下，尽可能提高对高速缓存的利用率。

2.2 B+-树

B+-树[4]是一种性能良好、广泛应用于传统数据库管理系统的索引结构。但 B+-树的内部结点仅仅是用来比较的，真正指向实际记录的指针都保存在叶子结点内，B+-树的高速缓存利用率较低。文献[5]分析了主存数据库中 B+-树的搜索时间，结果表明 B+树的搜索性能低于 CSS-树。另一方面，B+-树增量性能较好，插入和删除效率相对较高。

2.3 T-树

T-树[3]是内存数据库中最主要的一种索引方式。它是 AVL 树的一个变种，吸收了 AVL 树和 B-树的优点。T-树中所有的结点都直接指向数据项，节省了中间结点所占用的空间。T-树的提出时间很早，在当时的环境下还没有意识到缓存对索引性能的影响，设计重点只局限于降低索引的空间占用率，而没有针对高速缓存做优化。因此，T-树的高速缓存性能反而还不如 B+-树。

2.4 CSB+-树

CSB+-树[6]在 B+-树的基础上，针对高速缓存做出了优化，提高了高速缓存性能。CSB+-树吸取了 CSS-树[5]的设计思想，在树的结点中不再保存指向子结点的指针，而只在结点头部保存一个指向下层孩子结点的指针。同时，兄弟结点之间建立组的概念并连续存储，通过指针和偏移量定位子结点。CSB+-树提高了对高速缓存的利用率，减少了查询过程中的缓存失配次数。

2.5 CST-树

文献[7]中提出的 CST-树由数据结点与结点组组成。数据结点包含关键字与相应的记录 ID(RID)，结点组包含数据结点的最大键值与指向一个控制块的指针。控制块包含指向相应数据结点的指针、父亲结点组的指针、高度和指向孩子结点组的指针。CST-树对于原来的 T-树中的结点仅保留最大键值来进行比较，这样构造了一个仅仅包含原 T-树结点最大键值的二叉查找树。CST-树将结点组大小设计成一个缓冲块的大小。

3 pT-树

现代微处理器提供以下机制来处理大量的 Cache 缺失时延。首先，现代微处理器为了在存储层次中提高并行化，允许多重独立的 Cache 缺失同时发生。例如，Compaq ES40[9]每个处理器支持 32 in-flight 装入、32 in-flight 存储和 8 个独立片外 Cache 缺失，并且它的交叉存储系统支持 24 个独立缺失。其次，为了使得应用充分利用这种并行化优势，同时提供预取指令，软件可以在需要数据前将数据装载进高速缓存。前期的研究表明，对于基于阵列与基于指针的程序代码，预取可以通过重叠成功地消隐 Cache 缺失带来的性能影响[10]。所以，影响现代计算机性能的不再是 Cache 缺失数，而是不能通过诸如预取等技术成功地消隐的显式缺失延迟。

3.1 索引查找：预取技术创建加宽索引结点

CST-树的查找总是从根结点组开始逐层进行查找的，在结点组内的查找也是从对应的二叉查找树的根结点开始的。遍历了结点组内的二叉查找树没有找到绑定该关键字的数据结点，那么就需要到相应的下一层孩子结点组中进行查找。如图 1 所示，待查找的关键字为 287，首先从根结点组开始查找，因为查找的关键字 287 大于关键字 160、240、280，所以到下层的结点组中进行查找。在查找的第二个结点组中，因为关键字 287 小于关键字 300，所以标记关键字 300 对应的数据结点的位置，同时继续进行比较关键字 300 这个结点的左孩子结点关键字 290。关键字 287 小于关键字 290，此时将标记位改为关键字 290 对应的数据结点的位置。在叶子结点组进行了最后一次关键字比较后，在最后的标记位标记的数据结点中进行折半查找。如果找到关键字 287，那么查找成功，否则查找失败。

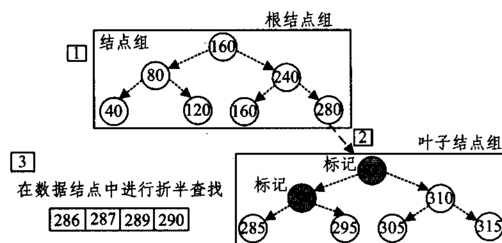

图 1 CST-树查找操作

CST-树结点组的大小为一个 Cache 块大小，所以在结点组内查找不会引起 Cache 缺失。每当需要到下一层相应的孩子结点组中进行查找时，就会引起一次 Cache 缺失，所以查找过程引起的 Cache 缺失数大概是与 CST-树的高度成比例的。因此，当结点组设计得越大时(多个 Cache 块大小)，树的高度会越低。在没有应用预取技术时(例如，Cache 缺失很昂贵并且不能被重叠)，索引结点大小大于正常数据传输大小(例如主存数据库的一个 Cache 块、磁盘驻留数据库的一个磁盘页)——实际上会影响性能。原因是将索引结点设计得更大后，虽然降低了树的高度，但同时会引起更多的附加 Cache 缺失。

例如，主存储器数据库中的 CST-树索引结构包含 270000 个关键字，数据结点包含 4 个关键字，Cache 块大小为 64 字节，关键字、指针、记录 ID 都是 4 个字节。如果结点组的大小为一个 Cache 块大小，树的高度至少是 4 层。图 2(a)说明在基于 Compaq ES40 的机型上，4 次 Cache 缺失花费 600 指令周期。如果结点组大小为两个 Cache 块大小，树的高度将降低至 3 层，如图 2(b)所示，将引起 6 次 Cache 缺失，执行时间增加 50%。

所以 Cache 缺失数与下式成比例：

$$w \times h \tag{1}$$

式中，w 为结点组包含的 Cache 块个数，h 为树的高度。

利用预取技术，可以预计地址的 Cache 缺失所引起的时延可能被消隐。图 2(c)表明，如果预取两个 Cache 块大小的结点组的第二 Cache 块，将其与第一个 Cache 块并行获取，与

图 2(a)中一个 Cache 块大小的结点组相比将获得更好的性能。Cache 缺失重叠装入的大小取决于存储层次的实现细节，总体趋势是支持更大的并行性。实际上，由于多重 Cache、内存条和交叉纵横连接器，因此可以通过重叠装入消隐多重 Cache 缺失。图 2(c)表明基于 Compaq ES40 的机型上仅在 10 个指令周期内才发生一次连续 Cache 缺失的时间，而这相对总共 150 个指令周期的缺失时延仅是一小部分。因此，通过设计比平常 CST-树的结点组更大的结点组，即使没有完善的缺失重叠技术，也会获得很大的性能提高。

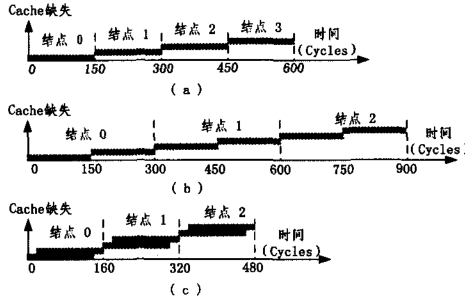

图 2　Cache 缺失时延

3.2　pT-树对 CST-树的修改

本文考虑一个标准的 CST-树结构：CST-树由数据结点与结点组组成，数据结点包含关键字与相应的记录 ID(RID)，结点组包含数据结点的最大键值与指向一个控制块的指针。控制块包含指向相应数据结点的指针、父亲结点组的指针、高度和指向孩子结点组的指针。pT-树修改了 CST-树结点组的大小，结点组大小被设计为一个 Cache 块大小的数倍，使得在结点组内的二叉查找树的查找范围更大。在当前结点组中查找不成功时，进入相应孩子结点组进行查找。查找成功时在对应的数据结点中进行折半查找。pT-树的基本操作算法是 CST-树基本操作算法的简单扩展。在此仅就修改的关键部分进行简单描述。

查找：结点组中进行查找前，预取组成结点组的所有 Cache 块。

插入：因为首先通过查找确定插入位置，所以在插入前查找路径上的索引结点都已经在高速缓存中。仅当由于插入而导致数据结点上溢出，需要重新分配关键字时，才会引起额外的 Cache 缺失。在重新分配关键字前预取相应结点。

删除：因为首先通过查找确定插入位置，所以在删除前查找路径上的索引结点都已经在高速缓存中。仅当由于删除而导致数据结点下溢出，需要重新分配关键字时，才会引起额外的 Cache 缺失。在重新分配关键字前预取相应结点。

3.3　pT-树性能分析

正如本节开始所讨论的，通过减少 pT-树的高度，同时不增加访问每层的结点的代价，可以提高查找操作的性能，减少查找的时间。更新操作总是以查找开始的，而当查找代价降低时，更新操作的代价也会随之降低。仅当插入时出现上溢出或者删除时出现下溢出时，昂贵的更新代价才会产生。尽管在多个 Cache 块大小的结点组中处理结点的上溢出以及下溢出相对需要较多时间，但是发生这些事件的相对频率可以减少。因此可以预期，索引结点包含多个 Cache 块的 pT-树比索引结点包含单个 Cache 块的 pT-树的更新操作性能更高。

将结点组大小设计为多个 Cache 块大小，减少了 pT-树操作的时间开销。T-树的关键字个数不变时，相应的 pT-树的结点组越大，pT-树的高度就越低。假设一个结点包含关键字个数相同，皆为 M，并且左右子树的高度相差为零的满 T-树，如果 T-树包含的关键字个数为 N，那么树中的结点个数为 N_1/M，T-树的高度为：

$$\log_2(\frac{N}{M}+1) \tag{2}$$

假设与该 T-树相应的 pT-树的结点组大小为 w 个 Cache 块大小，每个 Cache 块包含 m 个关键字，则每个结点组包含的关键字数目为 $d=w\times m$，那么

整个 pT-树的结点组个数为：

$$n=\frac{N}{Md}$$

结点组内的折半查找树的高度为：

$$h=\log_2(d+1) \tag{3}$$

每个结点组内折半查找树最底层结点个数为：

$$2^{\log_2(d+1)-1}=(d+1)/2 \tag{4}$$

每个结点组的孩子结点组个数为：

$$t=d+1 \tag{5}$$

整个 pT-树的高度为：

$$H=\frac{\log_2(\frac{N}{M}+1)}{\log_2(d+1)}=\log_2(\frac{N}{M}-d) \tag{6}$$

所以结点组包含的关键字数目 d 越大，树的高度就越低。因此必须考虑理想的 pT-树中结点组的大小，不是结点组包含的 Cache 块越多越好，实际上有两个系统参数影响结点组大小。第一参数是存储器子系统可以重叠的多重 Cache 缺失数，具体等于一个高速缓存缺失全部时延(T_1)除以额外的管道式高速缓存缺失增加时延(T_{next})。可以称这个比例(T_1/T_{next})为标准带宽(B)。B 的值越大，表示系统重叠并发访问的能力越强，因此从包含多个 Cache 块的结点组获益的可能性就越大。总的来说，并不期望结点组包含的最优的 Cache 块数目($w_{optimal}$)超过 B。第二个会潜在限制结点组大小的系统参数是高速缓存的大小，但是实际给定了理想的 B 值后这就不再是一个限制了。

接下来进一步定量分析理想的索引结点的大小。前面已经讨论过，一个包含 N 关键字、结点组大小为 w 个 Cache 块大小的满 pT-树的高度为 $\log_2(\frac{N}{M}-d)$。所以 pT-树结点组的平均查找长度为：

$$\begin{aligned}
ASL &=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\sum_{j=1}^{H}j\cdot t^{j-1}\\
&=\frac{1}{n}(\sum_{i=0}^{H-1}t^i+t\sum_{i=0}^{H-2}t^i\cdots+t^{H-1}\sum_{i=0}^{0}t^i)\\
&=\frac{1}{n}[H\cdot t^H-(t^0+t^1+\cdots+t^{H-1})]\\
&=\frac{1}{n}[(H-1)t^H+1]\\
&=\frac{1}{n}\{[\log_2(\frac{N}{M}-d)-1]\cdot t^{\log_2(\frac{N}{M}-d)}+1\}\\
&=\frac{1}{n}\{[\log_2(\frac{N}{M}-d)-1]\cdot(d+1)^{\log_2(\frac{N}{M}-d)}+1\}\\
&=\frac{1}{\frac{N}{Md}}\{[\log_2(\frac{N}{M}-wm)-1]\cdot(wm+1)^{\log_2(\frac{N}{M}-wm)}+\\
&\quad 1\}
\end{aligned}$$

$$= \frac{Md}{N}\left[\left(\log_2\left(\frac{N}{M}-wm\right)-1\right)\cdot(wm+1)^{\log_2\left(\frac{N}{M}-wm\right)}+1\right]$$

$$= \frac{Mwm}{N}\left[\left(\log_2\left(\frac{N}{M}-wm\right)-1\right)\cdot(wm+1)^{\log_2\left(\frac{N}{M}-wm\right)}+1\right] \quad (7)$$

所以平均查找时间为：

$$T = ASL\times[T_1+(w-1)\times T_{next}]$$
$$= T_{next}\times ASL\times(B+w-1) \quad (8)$$

式中，使得 T 值最小的 W 的取值就是 $w_{optimal}$。

表 1 列出变量名称与定义。

表 1　变量名称与定义

变量	定义
w	结点组包含的 Cache 行个数
m	一个 Cache 块包含的关键字个数
N	pT-树包含的关键字个数
d	结点组包含的关键字个数
T_1	一个 Cache 缺失全部时延
T_{next}	额外的管道式高速缓存缺失增加时延
B	标准存储带宽
M	数据结点包含的关键字个数

4　性能测试与评价

4.1　测试环境

本文通过多功能可执行程序在最新机器上运行的仿真，来测试 pT-树的性能。对于新一代机型，处理器速度与存储器速度之间的差距急剧增大，所以关注最新的机器性能特征相当重要。仿真的存储器体系层次是基于当前最新机型之一——Compaq ES40，但是稍微做了修改，包括动态调度、与运行时钟频率为 1GHz 的 MIPS R10000 类似的超大规模处理器。表 2、表 3 列举了仿真器的关键参数。使用版本为 2.95.2的 GCC 编译 C 源代码为 MIPS 可执行程序。在源代码中填加预取指令，使用 GCC 的 ASM 宏将这些预取指令译为有效的 MIPS 预取指令。

在主存环境下，对 T-树、CST-树、CSB+-树、B+-树的索引结构的操作性能进行测试与比较，CST-树的源代码、CSB+-树的源代码、T 树的源代码从其提出者的网站上下载，所有的索引结构的实现都支持查找、插入、删除。因为 CSB+-树使用"lazy"删除方式，所以在实现中采用这种使用最多的方式。

假设关键字为 4 字节的整数，每个指针大小为 4 个字节，Cache 块大小为 64 字节。根据表 2 给定的参数，标准带宽 $B=T1/T_{next}=150/50=50$，$m=15$，$M=8$，选取 $N=10^3,\cdots,10^9$，由 3.3 节式(6)通过计算与比较可得 $w_{optimal}=2$。如果标准带宽 $B=50$，那么 $w_{optimal}$ 增加到 13。

表 2　管道参数

Pipeline Parameters	
Clock Rate	1 GHz
Issue Width	4 insts/cycle
Functional Units	2 Integer, 2 FP, 2 Memory, 1 Branch
Reorder Buffer Size	64 insts
Integer Multiply/Divide	12/76 cycles
All Other Integer	1 cycle
FP Divide/Square Root	15/20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	gshare [15]

表 3　存储参数

Memory Parameters	
Line Size	64 bytes
Primary Data Cache	64kB, 2-way set-assoc.
Primary Instruction Cache	64kB, 2-way set-assoc.
Miss Handlers	32 for data, 2 for inst.
Unified Secondary Cache	2 MB, direct-mapped
Primary-to-Secondary Miss Latency	15 cycles (plus any delays Miss Latency due to contention)
Primary-to-Memory Miss Latency	150 cycles (plus any delays Miss Latency due to contention)
Main Memory Bandwidth	1 access per 10 cycles

4.2　测试结果

4.2.1　查找

第一个实验中比较了各种索引结构的查找性能。将叶子结点中的关键字数目取不同的值，进行 200000 次查找，测试查找时间。每个查找关键字都是随机选取的。图 3 是测试结果，可以看出 pT-树的查找速度最快，接下来是 CST-树、CSB+-树、B+-树和 T-树。pT-树平均比 CST-树、CSB+-树、B+-树和 T-树约快 16%，30%，41%，70%。

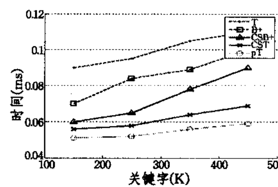

图 3　查找操作

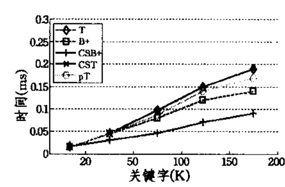

图 4　插入操作

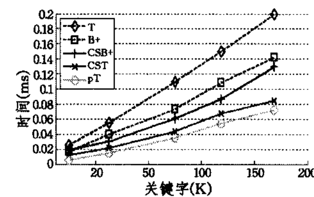

图 5　删除操作

4.2.2　更新

第二个实验中测试了各种索引结构的插入与删除操作的性能。在测试前，首先给索引结构块装入 100 万关键字，然后执行插入与删除操作并且测试性能。图 4 是插入操作性能测试结果，可以看出完全 CSB+-树删除操作性能最好，pT-树、B+-树、CSB+树、T-树删除操作性能较完全 CSB+树差。

图 5 是删除操作性能测试结果，总体趋势与图 3 查找操作类似。前面已经提到，使用"lazy"删除策略。删除操作的大多数时间花费在精确定位叶子结点中的关键字记录 ID。

结束语　在主存数据库应用日益广泛的今天，许多应用环境下也增加了对数据库实时性的要求，而有效的索引结构是实现数据库实时性要求的有力保证。由于 CPU 速度和主存速度之间的差距逐渐扩大，系统对主存的存取访问成为新的瓶颈。针对这种情况，本文结合现代处理器结构的新特性——Cache 装入的并发性，在 CST-树的基础上提出了一种高速缓存优化的索引结构——预取 T-树(pT-tree)。pT-树使用预取技术，能有效地创建比正常数据传输单元更大的索引结点，从而降低 CST-树的高度，进而减少进行查找时的 Cache 缺失时延，在查询密集型应用中能够更快地执行任务。在不远的将来，随着 CPU 速度和主存速度之间的差距继续扩大，CPU 对主存访问时间的瓶颈作用加剧，pT-树的性能优势

也将更加明显。

参考文献

[1] Chen S,Gibbons P B,Mowry T C. Improving index performance through prefetching[C]//Proc. ACM SIGMOD. Santa Barbara, USA,May 2001:235-246

[2] Luan H, Du X Y, Wang S. Prefetching J+-tree: A cache-optimized main memory database index structure[J]. Journal of Computer Science and Technolgoy,2009,24(4):687-707

[3] Lehman T J,Carey M J. A study of index structures for main memory database management systems[C]// Proc. VLDB Conference. Kyoto,Japan,Aug. 1986:294-303

[4] Comer D. The ubiquitous B-Tree[J]. ACM Computing Surveys, 1979,11(2):121-137

[5] Rao J,Ross K A. Cache conscious indexing for decision-support in main memory[C]//Proc. VLDB Conference. Edinburgh, UK, Sept. 1999:78-89

[6] Rao J,Ross K A. Making B+-trees cache conscious in main memory[C]//Proc. ACM SIGMOD. Dallas, USA, May 2000: 475-486

[7] Lee I-H,Shim J,Lee S-G,et al. CST-Trees:Cache Sensitive T-Trees[C]//Proc. of the 12th International Conference on Database Systems for Advanced Applications (DASFAA 2007). 2007:398-409

[8] Hennessy J L,Patterson D A. Computer Architecture: A Quantitative Approach[M]. Morgan KaufmannPublishers Inc. ,2002

[9] Cvetanovic Z,Kessler R E. Performance Analysis of the Alpha 21264-based Compaq ES40 System[C]//Proceedings of the 27th International Symposium on Computer Architecture (ISCA). June 2000:192-202

[10] Luk C-K,Mowry T C. Compiler-based Prefetching for Recursive Data Structures[C]//Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS). October 1996:222-233

(上接第 122 页)

为了计算 $F_s(x) \pmod p$,只需进行 $k_1+k_2+\cdots k_i$ 次迭代即可。s 的取值的因子越多,其效率就越高[5]。确切地讲,在 2048bit 精度下,s 和 r 的上界是 2^{970}。

4.4 算法效率和复杂性分析

上述算法由于需要类似 RSA、ElGamal 等算法选择一个大素数,由文献[12]有实数域离散多项式的迭代算法的时间复杂度为 $O(\log_2 n)$,其空间复杂度为 $O(\log_2 n)$,因此基于实数域离散多项式的公钥算法的效率与 RSA、ElGamal 相同。

4.5 选择迭代初值需要注意的两类值

(1) 几个不能用来加密的特殊 x 值

由式(4)可知:$x=0$ 时,$F_n(0)$的值是 1、0、-1、0 的循环;当 $x=1$ 时,$F_n(1)=1$;当 $x=p-1$ 时,$F_n(p-1)$的值是 1 和 $p-1$ 的循环,这些值都是模 p 计算后的结果[9]。

由于在 $x=0,1,(p-1)$时,取 $F_n(x)$值的特殊性使得它容易被破解,因此在加密过程中不选择这 3 个点作为密钥。

(2) 迭代特性对 x 的影响

由文献[7]的迭代特性可以扩展到实数域上,得到

$$F_n\left(\frac{1}{2}(a+a^{-1})\right) \pmod p = \frac{1}{2}(a^n+a^{-n}) \pmod p \quad (11)$$

则已知公钥(x,y),$x,y\in R$,$y=F_n(x)$后,破解 n 的过程为

① 令$\frac{1}{2}(a+a^{-1})=x$,则 $a=x+\sqrt{x^2-1}$;

② 由式(7)得

$F_n(x)\equiv F_n\left(\frac{1}{2}(a+a^{-1})\right) \pmod p = \frac{1}{2}(a^n+a^{-n}) \pmod p \equiv y$;

③ 得到 $a^n=y\pm\sqrt{y^2-1}$;

④ 已知 a,a^n,可通过求对数得到 n。

从以上破解过程可知,在利用迭代特性将求实数域 Chebyshev 多项式 $F_n(x)$中的问题转化为求离散对数的问题时,须满足(x^2-1)是模 p 的平方剩余,当 $gcd((x^2-1),r)=1$ 时,$m^2\equiv(x^2-1) \pmod p$有解[13]。否则,就不能求出 a,破解也就不可行。因此,在选取密钥(x,y)时,只要选择 x 使得(x^2-1)不是模 p 的平方剩余,就可以避免破解者利用迭代特性将求解 n 的复杂度降低。

结束语 将 Chebyshev 多项式与模运算相结合,对其定义在实数域上进行了扩展,结合 RSA 算法中密钥产生结构和 ElGamal 加密方案,利用 Chebyshev 多项式的半群特性,提出一种基于实数域的 Chebyshev 多项式的公开密钥算法;其安全性与 RSA 相似,都基于大数因式分解的难度,或者与 ElGamal 的离散对数难度相当,并易于软件实现。下一阶段,将通过研究提出基于有限域 Chebyshev 多项式的密钥协商、公钥加密和数字签名算法;并通过实验,分析其作为公钥加密体系的基础相对于 RSA 和 ElGamal 系统在计算效率上的优势。

参考文献

[1] Diffie W,Hellman M E. New Directions in Cryptography[J]. IEEE Transactions on Information Theory,1976,IT-22:644-654

[2] Kocarev L. Chaos-based cryptography: A Brief Overview [J]. IEEE Circuits Mag. ,2001,1(3):6-21

[3] Dachselt F, Schwarz W. Chaos and Cryptography [J]. IEEE Trans. Circuits Sys. 1:Fudam. Theory Appl. ,2001,48(12):1498-1509

[4] Schmitz R. Use of Chaotic Dynamical Systems in Cryptography [J]. J Franklin Inst. ,2001,338:429-441

[5] Kocarev L. Public-key Encryption Based on Chevyshev Maps[C]// Proc IEEE Symp. Circuits Syst. Vol 3,2003:28-31

[6] Bergamo P,D'Arco P,Santis A D,et al. Security of Pulbic-key Cryptosystems Based on Chebyshev Polynomials [J]. IEEE Tran. on Circuits and System-1: Regular Papers,2005,52(7): 1382-1392

[7] G'erard M. Algebraic Method for Constructing on Way Trapdoor Function[D]. Notre Dram:University of Notre Dame,2003

[8] Yoshimur T, Kohda T. Jacobian Elliptic Chebyshey Rational Map[J]. Physical D,2004,148(3/4):242-254

[9] 刘亮,刘云,宁红宙. 公钥体系中 Chebyshev 多项式的改进[J]. 北京交通大学学报,2005,29(5):56-60

[10] 王大虎,魏学业,李庆九,等. 基于 Chebyshev 多项式的公钥加密和密钥交换方案的改进[J]. 铁道学报,2006,28(5):95-98

[11] Kohda, Tohru, Hirohi F. Jacobian elliptic. Chebyshev rational maps[J]. Physical D,2001,148(3):242-254

[12] 王大虎. 非线性理论在保密通信中的应用研究[D]. 北京:北京交通大学,2006

[13] 卢铁成. 信息加密技术[M]. 成都:四川科学技术出版社,1989