一种快速的滑动标记缩并垃圾收集算法

中国科学技术大学计算机科学与技术系 安徽合肥
中国科学院计算技术研究所计算机系统结构重点实验室 北京

K 1 语言完全面向对象 因此对象局部性是衡量 虚拟机性能的重要指标 在 虚拟机中 由垃圾收集算法负责检测并且回收不再使用的对象 它直接影响着 程序的性能 保持对象分配序能够提供最佳的局部性 滑动标记缩并算法正是基于这一原则 但是传统上的设计使得算法的效率很低 本文提出一种快速的滑动标记缩并算法 它通过位图、活块池和块内偏移表来简化算法 大大的降低了开销 实验结果表明 快速的滑动标记缩并算法使得标准工业测试程序 的性能在 上有不同程度的提高 最高达到 同时程序的局部性也优于线性标记缩并算法 与深度遍历序相比 !" # $%& ! &&& $(")) # 与 级 *+ 失效率改善最多分别为 和 ,

1 o M 垃圾收集 标记缩并 位图 活块池 块内偏移表
Ï ms Ë| ÓDSM' - ÓcI | .. ,

A Fast Slide Mark Compact Algorithm

/01 2 &3 /4-56 !&37 3
(. Department of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui , China;
. Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing , China)

Abstract: - %&* & &#(# $ + 7 $)&# %&* %8 9 + *+ $%(# *& : * %&&# + $ 7 $(& " #(& %($3 (+ %&&# + $& #+ (&&%#3 ; :#& &$( ) $ $%(# *& : * %&&# + 9 + *+ #(* $ + & #+ (78 # 7 7% % 7%&* :&&% (&))$ 7% + #$ %$$+&9 + *+ $: & $ (: ($#8 $ (#(7 *+ # & + :#& (%& $$ 7 #$ (, #(* 9 + ! * *+ $$ 7 #$

Key words: 3 #7 3 *&%&* & # *& : * # 7 7% % 7%&* :&&%&))$ 7%

1 „ ý

是当今主流的程序开发语言之一 它具有易移植性、安全性、动态性等优点 在 的世界中 虚拟机起着重要的作用 是程序平台无关和拥有良好性能的关键 语言完全面向对象 因此对象的局部性是衡量 虚拟机性能的重要指标[= >] 语言允许程序员显式地申请对象空间 而由 虚拟机负责检测和回收不再使用的对象 该过程称为垃圾收集

经典的垃圾收集算法[>]有引用计数法、标记清除算法、节点复制算法和标记缩并算法 其中标记缩并算法在保证高效的同时带来最好的局部性 因此被许多虚拟机[= >]广泛采用 根据缩并过程中移动对象顺序的不同 它可以分成两类 线性型[' >]和滑动型[>] 线性型尽可能使原来的单元和它所指向的单元放在相邻的位置 包括深度遍历序 广度遍历序和层次遍历序 它们的时间复杂度相同 但是深度遍历序往往能够获得更好的空间局部性[' >] 滑动型按照单元分配的次序将存活单元滑动到堆的一端 这样可以提供最佳的局部性[>] 然而此类算法堆遍历次数较多 因此带来的开销较大

本文提出一种快速的滑动标记缩并算法 它在标记阶段记录位图和存活块池 在缩并阶段计算块内偏移表 将对堆的遍历转化为对块内偏移表的访问 大大地降低遍历堆所带来的开销 同时活块池的引入使得该算法很容易被应用在并行垃圾收集算法中 实验证明该算法使得标准工业测试程序 [>]的性能有不同程度提高 最高达到 同时程序的局部性也优于线性标记缩并算法 与深度遍历序相比 !" 失效率改

收稿日期 . 修回日期
基金项目 国家自然科学基金杰出青年基金 5&, . . 国家 重点基础研究发展规划 5& ." , 国家自然科学基金 5& ' ' ' ' ' 国家 , 高技术研究发展计划 5& ,-- 北京市自然科学基金 5&

善最多为 级 *+ 失效率改善最多为 ,

2 ß º

2 1 DRLVM

目前主流的虚拟机有 15 A< >、" - ?&* A< >、@A ? < > ? $ #*+ # % *+ 和 ? ! < > 8 *? ! 8# # % *+ 其中 15 A和" - ?&' A都是商业化的产品 性能优秀但是模块层次不清晰 针对性太强 因此可研究性较差 @A ? 和 ?! 都是致力于研究用途的虚拟机不同的是 @A ? 采用 语言编写而成 ?! 采用 BB 和少量汇编语言编写而成 并且相对于 @A ? 而言 它具有更好的结构层次和优秀的模块化等特征 因此本文选择在 ?! 上展开研究

2 2 ¥ F ® ™T

?! 的垃圾收集算法将堆分成三个区域 50 5 #$ #8 07C* : * 0 # 07C* : * 和!0 ! #8 07C* : * 如图 所示 同时 堆又被分成 个分代 年轻的分代和年老的分代 50 为年轻分代 而 0 和!0 则构成年老的分代

50 和 0 两者以块 7%&' 为单位组织的 块大小可配置 缺省值为, A" A %&8 !0 则组织成自由链表的形式 采用标记清扫的方式进行收集 而且大型的对象将会直接被分配在!0 中 这主要是因为拷贝大型对象的开销很大 因此 一般的垃圾收集设计都是将大型的对象放在一特定区域里 并尽量避免对大型对象的拷贝 在算法设计中 允许对大型对象的最小大小进行配置

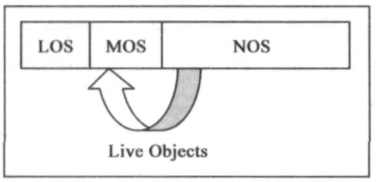

图 1 堆的布局

2 3 . d ¥ Á î S: ê i Ø E

传统的滑动标记缩并算法需要遍历堆 次 标记阶段 次 缩并阶段 次 并且在每个对象的头中保留一个指针域来存放迁移地址 如图 *a* 所示 在缩并过程中 第一次遍历堆时 计算每个存活对象的地址 并将其保存在它的对象头的)&9 #(3D ((#$$域中 新地址的计算方法很简单 就是到目前为止所遇到的存活对象的大小的总和 存放在)# 变量里 第二次遍历堆时 更新存活对象中指针域的值 使它们的值等于它们所指向对象的)&9 #(3D ((#$$域 最后 清除每个节点的)&9 #(3D ((#$$域 为下次垃圾收集做准备 同时将对象到他们的新地址 在这一阶段末尾 所有存活数据被缩并地存放在堆中较低的部分 而堆中第一个自由位置的索引存放在变量)# 中

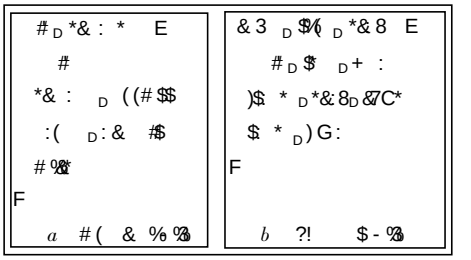

图 滑动标记缩并算法 传统 $?!

2 4 DRLVM Ï ¥ Á î S: ê i Ø E

传统的滑动标记缩并算法之所以需要对堆进行四次遍历 是因为存活对象的目的空间与源空间是公用的 更新指针域必须在移动存活对象之前进行 否则有可能活的对象会被覆盖掉 在 ?! 的设计中 将存活对象的目的空间与源空间分开 这样就不会存在原来的对象会被移动的对象覆盖的现象

算法如图 *b* 所示 简单描述如下

第一次遍历堆时 计算每个存活对象的地址 然后将其从50 中拷贝到 0 中去 并且将这个地址保存在50 中相应对象头的)&9 #(3D ((#$$域中 第二次遍历堆时 更新存活对象中指针域的值 使他们的值等于它们所指向对象 &9 #(3D ((#$$域 ?! 中的滑动标记缩并算法需要对堆遍历三次 包括标记阶段

3 y Î ¥ Á î S: ê i Ø E

3 1 ØE Ï P¨ ¥ M

如图 中所示 新的对象分配在50 空间中 一旦50 中剩余空间不足时 便会产生一次垃圾收集 存活的对象将从50 拷贝到 0 中去

在本文提出的滑动标记缩并算法中 我们引入三个数据结构 位图 活块池和块内偏移表

位图 " 该图中每一位对应50 中的 个字节 一个对象的起始位和终止位将在 " 对应的位上置 在 ?! 中 对象以 个字节对齐 如果50 大小为 那么 " 大小为 A

活块池 !" 池中存放的是每次垃圾收集时存活的基本块 在 中 每次垃圾收集发生时 基本块的存活率在 的情况下低于.

块内偏移表 0 每次垃圾收集发生时 用来记录基本块中第一个活对象的目标地址在 0 中的偏移

位图可以使存放标记信息的内存数量最小化 除此之外 它还有两个优点 第一 如果位图相对较小 它可以保存在内存中 从而读写标记位不会导致缺页中断 第二 在标记阶段中 收集器不会改写任何堆对象 这就意味着堆内存对应的所有页面不会成为脏页 因此当它们被换出操作系统的虚拟存储页面时 操作系统无需将这一页面写入交换磁盘

算法的比较 分别

域和

的次数 传统的滑动标记缩并算法需要遍历

中的滑动标记缩并算法需要遍历堆

堆

次 代价就是需要额外的遍历 " 次 但是和堆相比 " 要小得多 另外本文提出的算法不再需要为每个对象保留)&9 #(3D ((#$$域 大大的节省了空间

V1 ØÕÁî S: êi ØE¥1

	遍历堆的次数	I&9 #(O	遍历 " 的次数
传统算法		P$	
?! 中		P$	
快速算法		5&	

4 ØE i › Ä Z ë ¥ ü %„ ‰Z

随着共享内存的多处理器被广泛使用 许多内存管理技术的研究者们将目光转移到多处理器系统上来 为了让快速的滑动标记缩并算法也适用于多处理器系统 我们将其进行进一步地拓展

在目前的算法中 每个6 6 #7 3 &9%* & 线程从池中取出不同的基本块并对其中的活对象进行缩并处理 由于不同的6 线程对同一个池进行操作 那么池的取操作就需要同步 为了避免同步的开销 我们采用交叉访问基本块的方式进行 如下图 所示 池中有若干个基本块 有两个6 线程 分别用6 D 和6 D 来表示 那么6 D 将处理的基本块号依次为 , 6 D 负责的基本块为 . 当6 线程个数为 T 时, 线程 i 与基本块的对应关系如下:

$$i\mathrm{B}\ T\mathrm{K}\ j, j\mathrm{J}\quad ,\ ,\ ,$$

上述做法很好地减少了同步开销 但同时引入一个新的问题 不同的6 线程进度可能不一致 在很多时候 会因为少数6 线程的进度慢而增加算法的时间复杂度 针对该现象 我们为每个6 线程维护一个基本块队列 7%&' Q 最开始每个Q 中的内容就是他负责的基本块集合 当某个7%&' Q 空闲了 它就尝试着检查其他线程的队列 当它发现一个非空的7%&' Q 它就把该Q 中一半的任务获取出来放入自己的7%&' Q 中 因为几个收集线程可能同时变为空闲 因此 从某个7%&' Q 中获取任务的操作必须进行同步 当所有的7%&' Q 变为空的时候 那么缩并过程就结束了

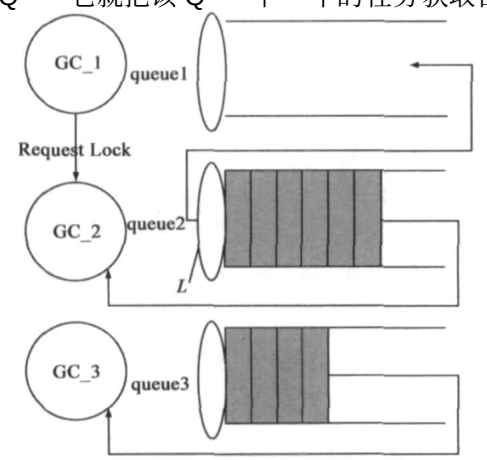

图4 动态负载平衡机制

如图 所示 有6 D 6 D 6 D 三个垃圾收集线程 各自有一个7%&' Q 分别为Q Q 和Q 当前时刻6 D 对应的Q 已经空闲 它会去检查Q 发现Q 中还有 , 个基本块没有处理完 6 D 便会向Q 发出请求 同时去获取Q 的锁 一旦成功 便会截获Q 中一半的任务

5 Ÿ ? s

本文将从复杂度和局部性两个方面对快速的滑动标记缩并算法进行评估 一方面将该算法的时间复杂度与线性标记缩并算法以及 ?! 中的滑动标记缩并算法进行比较 另一方面对比快速的滑动标记缩并算法和线性标记缩并算法对内存访问性能的影响

5 1 Ü û º

本文中采用 中的程序做为测试程序 并且在 上运行最大的数据集 因为运行较小的数据集 、 时 收集到的数据不具有代表性 运行环境是 64N @ 6" 内存 操作系统是; (&9H #&) $$& % 编译器使用的是 15 A .

5 2 ØE¥ ¯ s

图 . 中给出的是不同的标记缩并算法所耗费的时间 6 其中浅色条表示快速的滑动标记缩并算法的6 深色条表示 ?! 中的滑动标记缩并算法的6 白色条是线性标记缩并算法的 6 前二者都是基于保持对象分配序原则的 从图中可以看出两点 第一 就保持分配序和深度遍历序而言 深度遍历序所耗费的时间较短 因为它只需要遍历堆一次 而保持分配序的时间较长 在 ?! 的标记缩并算法中需要遍历堆三次 而在快速的滑动标记缩并算法中 需要遍历堆一次和 " 两次 第二 与 ?! 中的滑动标记缩并算法相比 本文算法6 明显降低 对于大部分测试程序 它的时间与深度遍历序的时间接近 这充分说明本文的滑动标记缩并算法优于 ?! 中的滑动标记缩并算法

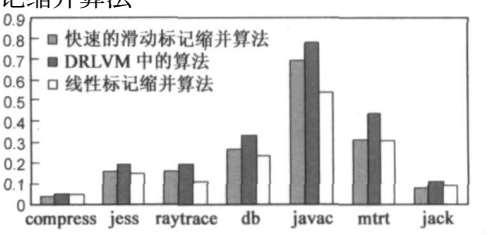

图5 各种标记缩并算法的 GC 时间比较

位图、活块池和块内偏移表需要引入内存开销 位图的大小取决于50 空间 块内偏移表的大小取决于 0 空间和基本块 活块池的大小取决于每次垃圾收集时活块的数目 而事实表明基本块存活率低于 . 这意味着活块池大小仅为50 空间的 . 在我们的实验环境中 基本块大小为 , A" 位图、活块池和块内偏移表占用的内存空间为

$$\frac{50}{}\mathrm{B}\ \frac{50}{\mathrm{K}\ ,\ \mathrm{K}K}\mathrm{B}\ \frac{50}{,\mathrm{K}\,K}\ KJ$$

该公式表明它们引入的内存开销可以忽略不计

5 3　ØE　† Ÿ ¥ • Y

程序运行的总时间为垃圾收集时间和用户程序时间 &# 之和 垃圾收集的时间开销反映了垃圾收集算法自身的效率 而对于仅有垃圾收集算法不同的运行时系统来说决定用户程序时间的主要因素包括分配函数和写拦截函数的实现以及垃圾收集算法对用户程序的对象局部性的影响 在本文实验中提到的三个垃圾收集算法 均采用相同的分配函数实现 并且这三个算法都是标记缩并算法 不存在写拦截的开销 因此 影响 &# 的主要因素应该是垃圾收集算法对对象局部性的影响

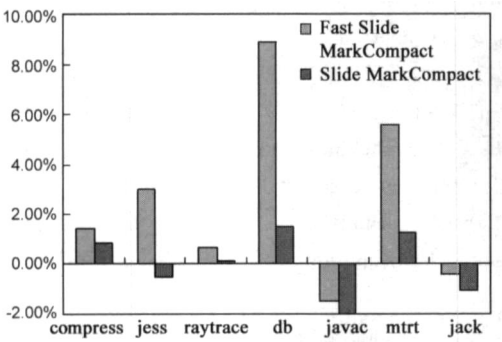

图 6　快速的滑动标记缩并算法的加速比

图 , 给出的是滑动标记缩并算法所带来的加速比 基准是深度遍历序的标记缩并算法 浅色条是本文算法所带来的加速比 而深色条是 ?! 中的滑动标记缩并算法所带来的加速比 从图 , 中可以看出对于 测试程序来说 ?! 的缩并算法性能普遍不如深度遍历序的标记缩并算法 而本文的缩并算法普遍优于深度遍历序的标记缩并算法 上面已经介绍 & % 是6 和 &# 之和 对于 ?! 的标记缩并算法和本文的标记缩并算法 二者的 &# 应该是一致的 因他们都是基于保持对象分配序这一原则的 那么影响 & % 的应该就是 6 事实上 结合图 . 和图 , 发现 6 对于程序运行时间的改善是非常重要的 其中 # 有 的性能提高 而 6 的降低正是性能改善的主要原因

为了进一步验证滑动标记缩并算法能够很好的改善程序的局部性 我们在图 给出了滑动标记缩并算法和线性标记缩并算法对 !" 以及! *+ 失效次数的影响 我们采用的性能分析器是 @ % #j&# * - %&N#·> #$& 我们发现除了 C * 其他程序的 !" 失效次数分别有 到 的提高 而对于! *+ 失效次数 有三个程序增加了 . 到 . . 它们分别是 #8#* C * 和 C*' 虽然 #8#* 和 C*' 的 !" 失效次数分别有 和 的降低 但它们的! *+ 失效次数却有不同程度的增加 二者综合的结果使得 #8#* 性能提高仅为 , C*' 性能下降

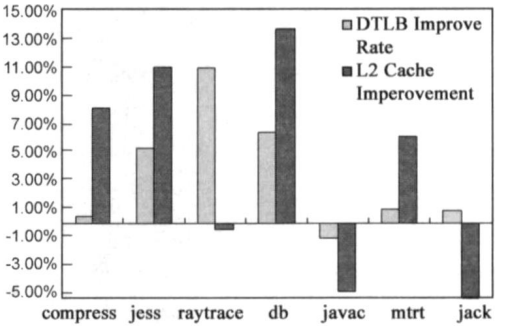

图 7　快速的滑动标记缩并算法的 DTLB 和 L2 Cache 失效率的改善

在大部分情况下 相对于深度遍历序 保持对象分配序能够更好地维持对象局部性 但这不是绝对的 * 是一个反例 当采用保持分配序的垃圾收集算法时 它的 !" 和! *+ 失效次数都有所增加 而事实上 它在编译 C 程序的过程中 常常需要对语法树进行深度遍历 采用深度遍历序的垃圾收集算法更符合 C * 的行为特征 因此能取得更好的性能

5 4　ñ GC L ñ { Ÿ ¥ F Î 1

图 中给出的标记缩并算法中缩并和标记各自消耗时间的最大比例 从图中可以看出 *& : #$$ C$$ 和 C*' 这三个程序的缩并阶段所消耗的时间都比较短 主要原因在于它们的总垃圾收集时间就很短并且每次对象的存活率都很低 另外四个程序的 6 都比较大 在最大情况下 缩并阶段消耗时间的比重超过了标记阶段 因此在多个垃圾收集线程同时工作时 对缩并阶段的并行考虑是有意义的

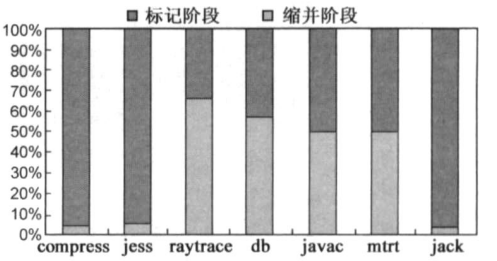

图 8　标记阶段时间 VS. 缩并阶段时间的最大值

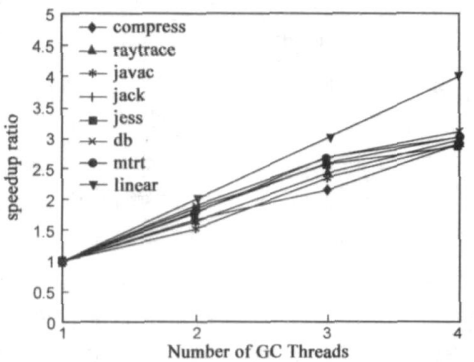

图 9 快速的滑动标记缩并算法 GC Time的可缩放性比较

图 是缩并过程所消耗时间随 6 线程个数的一个可缩放性比较 在该实验中 使用的硬件平台是一台 1 8 ##$ # # . $# $ 具有 , 个@ % H& 处理器 每个处理器的主频是 64/ 并有 " 的* *+ 操作系统平台是I (&# &# # %$ 我们的实验仅对缩并阶段的行为进行评估 从整体上看 当 6 线程个数为 的时候加速比为 随着 6 线程个数的增多 可缩放性逐渐变差 但总的来说 所有测试程序的加速比都分布在线性图附近 在 6 线程个数为 的时候 最高的加速比达到了 最差的有 **6** ² ,

标记缩并算法可以分为两类 线性型和滑动型 它们在不同程度上确保对象在堆中的空间次序反映它们的分配次序 而滑动型倾向于提供更优的局部性 但传统设计使得滑动的缩并算法的时间复杂度很高 本文提出一种快速的滑动标记缩并算法 它通过位图、活块池和块内偏移表来简化算法 大大降低了开销 实验结果表明 快速滑动标记缩并算法使得标准工业测试程序 的性能有了不同程度的提高 最高达 同时再次证明保持分配序能提供最佳的局部性 相对于深度遍历序而言 的 !" 失效率改善最多为 级 *+ 失效率改善最多则为 ,

• I Ó D

< >; A + "+ $% % #&) % 3 ((:#& * 3 #7 3 *&%%* &)&# %&* %8 & N & <- > #&* (3$ &) + , - @ ! - 5 &) # * & #&3# 3 ! 3 3 $3 (@: % & < > 5 9 P&# - #$$, D

< > +% 7 " ($& % *+ *& $* & $$# * # ( ) &<- > #&* (3$ &) + - @ ! - 5 &) # * & #&3# 3 ! 3 3 $3 (@: % & < > 5 9 P&# - #$$ D

< > "%*' 7 # + 3 % 8+$ (? % $ + : #)&# * : * &) 3 #7 3 *&%%* & <- > @ #&* (3$ &) + & @ # & % &) # * & $# (&(%3 &) & : # 8$ $< > 5 9 P&# - #$$. D ,

< > "%*' 7 # + 3 %0 % (9 #O 4 3+ : #)&# * 3 #7 3 *&%%* & C 9 + '<- > @ #&* (3$ &) + , + @ # & % &) # * & &) 9 # 3 # 3< > 5 9 P&# - #$$ D ,

<, > ? & $? ! $ 6 #7 3 &%%* & - %&# + $) &# - & * 8 * &#8 3 < > - #* &+ ; %8 M & $,

<, > H 4 3 "%*' 7 # % + 3 #7 3 *&%%* & (3 :#& 3 :#&3# %&* %8 <- > @ #&* (3$ &) + + - %- @ ! - 5 &) # * & 07C* &# (#& 3# 3 8$ $! 3 3 $ (- : : % & $< > 5 9 P&# - #$$, D

< > 39 # 4 #N %@: #& 3 %&* %8 9 + : # %%+ ## *+ * %*& 8 3 6 <- > @ #&* (3$ &) @ # & % 8 : & $ & &#8 3 < > 5 9 P&# - #$$, . D ,

< > ; %# $ # $ &) (8 * %$ $# * # $! @ < > @ # $ * & $ & &) 9 # 3 # 3 . . D .

< > (#(#)&# * % & &# &# & *% C <0! > + : RR9 9 9 $ * &#3R&3R C

< > 4 &$ & # % *+ <0! > + : RR C $ *& RC $ R *+ &%&3 $R+ &$ & R

< > " - ?&*' . <0! > + : RR9 9 9 7 *& R) # 9&# G$ O 5 J (G+ M J R*& R #&(* $RG&*' R

< > ' $? $ #*+ # % *+ ? <0! > + : RR9 9 9 7 *& R(%& #9 &# $R&$$RC $#

< > 8 * ? ! 8 # # % *+ #% <0! > + : RR+ # & 8 : *+ &#3R$ 7* & : & $R(#% R (G + %

< > @ % &# &# & @ %? - #*+ * # &) 9 # %& # $ % &% @$ # * & ?) # * %0 ! > + : RR(&9 %& (%*& R($3 R #*+R %$

< . > @ % &# &# & #)&# * - %&N #<0 ! > + : RR 999 %*& R*(R&) 9 #R #&(* $R $ & R 3R R +

T € e º

; j 女 年 . 月出生于江西南昌 年毕业于中国科学技术大学计算机科学技术系 同年进入该校进行硕博连读 从事系统结构与动态编译等方面的有关研究

%N& Q& 3S * * *

c H a 男 年出生于安徽芜湖 博士 博士后 副研究员 硕士生导师 主要研究方向为高性能计算机系统结构、并行处理等

%%7N+ 3S * * *