文章编号:1001 - 9081(2006)05 - 1167 - 04

# 逆向工程中基于投机的反向优化技术研究

苏 铭,宋宗宇,赵荣彩
(信息工程大学 信息工程学院,河南 郑州 450002)
(mingsu@sohu.com)

**摘 要**:投机机制缩减从内存中装入的延迟,编译器优化为了充分利用这一架构特征,需要对程序的低级代码进行深度重构,这很大程度上增加了对代码的静态分析和再工程的复杂性。文中提出的反向优化技术,在安腾的二进制代码中消除投机指令并保证程序的语义,使得投机消除后的程序更容易理解,更易于应用逆向工程的技术进行代码再工程。

**关键词**:逆向工程;投机;反向优化;安腾;再工程
**中图分类号**: TP311 **文献标识码**:A

# Reverse optimize technique based on speculation in reverse engineering

SU Ming, SONG Zong-yu, ZHAO Rong-cai
(*College of Information and Engineering, Information Engineering University, Zhengzhou Henan* 450002, *China*)

**Abstract:** Speculative loads can reduce the latency associated with loads from memory. In order to make effective use of the capabilities of such architecture, compilers profoundly restructure the low level code of programs, however, making it potentially difficult to reconstruct the original program logic from an optimized executable. This complicates the task of software systems that statically analyze or modify executable programs. A technique for removing speculative instructions from itanium binary programs was described in a way that is guaranteed to preserve program semantics, thereby making the resulting programs easier to understand and more amenable to re-engineering using traditional reverse engineering techniques.

**Key words:** reverse engineering; speculation; reverse optimize technique; itanium; re-engineering

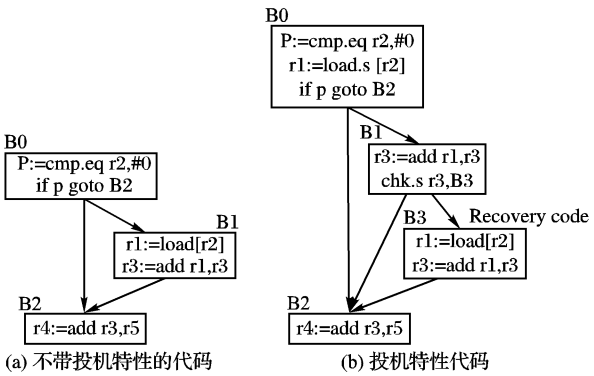
## 0 引言

CPU 和主存之间在速度上的差异越来越大,存储延迟已经成为影响微处理器性能提高的严重瓶颈。在先进的体系结构中应用指令调度技术来缓和这对矛盾,例如安腾提供了新的架构特征称为投机,它的思想是允许(长-延迟)指令提前执行——尽可能早地运行高代价计算,这样当需要时计算结果已经得到。明智的应用投机可以使得性能得到很大提高[1],但是投机改变了产生代码的结构,不能很好地反映源程序的逻辑,而且和没有优化的代码相比,指令的位置在很大程度上发生了改变。这样投机使得低级代码晦涩并且增加了程序的理解、分析和逆向工程的难度。因而,使得源码不可知的软件维护或理解工作复杂化。

## 1 投机带来的问题

为了产生高效的代码,优化的编译器试图通过指令调度来掩盖延迟。但是指令调度受指令间相关性的限制。特别,当指令 $I$ 与条件分支指令 $J$ 控制相关 —— 也就是,$J$ 决定 $I$ 是否执行 —— 则 $I$ 不能调度到分支指令 $J$ 之前,如图 1(a)。基本块 $B_0$ 测试 $r_2$ 是否包含非空值,基本块 $B_1$ 的装入指令与 $B_0$ 的分支指令控制相关,在以下情况下将装入指令提前到分支之前会导致错误:如果 $r_2$ 值为空,结果代码将产生错误。这样的控制相关限制了隐藏高代价延迟操作的能力,例如从内存中的装入操作。

投机机制[2]的基本特征是投机装入指令,操作码标记为“load.s”。投机装入的行为与正常装入的最大不同点是:如果指令产生一个异常,例如段或页错误,则不会立即处理;相反的,设置与装入目标寄存器相关的特殊位,当程序运行到后面需要使用装入的值,这时使用特殊的投机检验指令(chk.s)对目标寄存器进行检验。如果寄存器设置了 NAT(Not A Thing)位,则跳转到编译器提供的恢复代码处执行,否则,正常顺序执行。NAT 位可以在寄存器之间传播,也就是说,如果指令的源寄存器设置了 NAT 位,则该指令中的目标寄存器也将设置相应的位。在投机装入后面的一串相关的指令将在恢复代码中出现。




(a) 不带投机特性的代码 (b) 投机特性代码

图 1 特征代码

将图 1(a)中的装入指令提到前面的分支指令之前,将一般的装入指令变换成投机装入指令,结果代码如图 1(b)。与变换前相比:首先增加了更多的指令,更多的执行路径,更复杂的程序结构;其次投机装入远离它的应用,并插入了恢复代码。在大的程序中,投机装入指令越过多条分支指令被提前,

**收稿日期**:2005 - 11 - 30;**修订日期**:2006 - 03 - 02 **基金项目**:河南杰出人才创新基金资助项目(0521000200)

**作者简介**:苏铭(1974 - ),女,河南洛阳人,博士研究生,主要研究方向:计算机软件与理论; 宋宗宇(1975 - ),男,江苏句容人,讲师,硕士,主要研究方向:计算机软件与理论; 赵荣彩(1957 - ),男,河南洛阳人,教授,博士生导师,主要研究方向:计算机软件与理论.

同时恢复代码中含有其他的投机或检验指令,这将会造成控制流很大程度上复杂化。文中提到的投机消除算法是投机的逆向过程,使得代码更容易理解。

## 2 投机消除技术

投机消除是指将包含投机装入的代码变换成语义上等价的"正常"程序,消除部分或所有的投机指令。将投机装入指令移动到代码的某一个或多个点,在该处可以被不带投机特性的装入操作所代替,称其为装入下沉;另外投机消除还需要确认消除检验和相应的恢复代码是安全的,这样投机装入才能被消除投机后的装入所替代。通常这两个步骤执行起来要复杂得多,因为装入和检验之间往往不是一一对应的关系。下面先讨论指令相关性和投机域这两个概念。

### 2.1 指令相关性和投机域

#### 2.1.1 指令间关系

如果投机装入 $I$ 下沉,则其他与 $I$ 相关的指令也必须下沉。为了精确相关性这个概念,定义两条指令 $I,J$ 为直接相关($I \Longleftrightarrow J$),如果:

1) $I$ 写入的寄存器或内存单元都可被 $J$ 读取;

2) $I$ 可以从 $J$ 写入的任何寄存器或内存中读取;

3) $I$ 和 $J$ 可以写同一寄存器或内存单元。

$\Longleftrightarrow *$ 表示 $\Longleftrightarrow$ 的反传递闭包,如果 $I \Longleftrightarrow * J$ 成立,则 $I$ 和 $J$ 相关。

通常,确定检验指令是否与给定投机装入指令相关,需要确定检验指令的源寄存器是否设置了 NAT 位。给定指令 $I \equiv$ 'r := load. s…' 定义了寄存器 $r$ 和检验指令 $J \equiv$ 'chk. sr,…',称 $J$ 检验 $I$,如果下面条件成立:

1) $r' \equiv r$,$r$ 的定义 $I$ 到达 $J$;

2) 应用 $r$ 的指令 $I'$,传播 NAT 位从源操作数到目的操作数,于是有(a)$r$ 的定义 $I$ 到达 $I'$;(b)$J$ 检验 $I'$。

与投机装入指令 $I$ 相关的投机检验指令集合 $Chk(I)$,定义如下:

$Chk(I) \equiv \{J \mid J$ 是投机检验,$J$ 检验 $I\}$

与投机装入 $I$ 的 $Chk(I)$ 集合类似,定义与检验指令 $J$ 相关的投机装入指令集合 $Ld(J)$:

$Ld(J) \equiv \{I \mid I$ 是投机装入,$J \in Chk(I)\}$

该集合可以从 $Chk$ 集合中导出。

#### 2.1.2 投机域

为了将装入指令下沉到投机检验指令 $J$,初始化时下沉到 $J$ 的指令集必须很好定义。也就是,对所有的投机装入指令,$I \in Ld(J)$ 是相同。

为了能够描述的更精确,定义投机域如下:

**定义 1** 投机装入 $I$ 的投机域是一对 $(L,C)$,$L$ 是投机装入指令集合,$C$ 是投机检验指令集合,$L$ 和 $C$ 是满足下面条件的最小集:(a) $I \in L$;(b) 如果 $x \in L$ 并且 $y \in Chk(x)$,则 $y \in C$;(c) 如果 $x \in C$ 并且 $y \in Ld(x)$ 则 $y \in L$。

投机域在投机消除中作为一个独立的单元。这意味着对每一个域,装入下沉成功则域中所有的投机代码被消除,或者失败,没有指令被消除。从投机装入 $L$ 到检验 $C \in Chk(L)$ 的执行路径 $\pi$,定义 $DEP_L(\pi)$ 表示沿着 $\pi$ 与 $L$ 相关的指令集合,这样可以精确定义装入下沉的条件。

**定义 2** 投机装入的投机域 $(L,C)$ 是路径无关的,是指对任意投机装入对 $I_1, I_2 \in L$ 和检验 $J \in C$,在任何 $I_1$ 和 $J$ 之间的路径 $\pi_1$ 和 $I_2$ 和 $J$ 之间的路径 $\pi_2$,存在 $Dep_{L_1}(\pi_1) = Dep_{L_2}(\pi_2)$。

### 2.2 算法概要

投机代码有两个特性使得装入下沉变得复杂。首先,可能根据投机装入的结果做多种操作(例如算术运算)。如果投机失败,则计算结果的 NAT 位将通过这种操作传播。其次,投机装入和投机检验不一定一一对应:特殊的投机可能有几条相关的检验指令,一条投机检验指令可能与多条不同的投机装入指令对应。第一个特性意味着进行装入下沉时,不仅要移动投机指令,而且还要移动与投机指令相关的指令。第二个特性意味着当投机检验指令与多条投机装入指令相关时,必须确认下沉的指令集对每一条相关的投机装入指令是相同的。

算法开始将投机装入指令和投机检验指令组成投机域,初始化时,投机域为投机装入指令和相关检验指令之间存在的检验者和被检验者关系闭包。

对每个投机域,需要确定无论对在该域中任何一个投机装入和投机检验,必须下沉的指令集 $I$ 是否相同。如果条件满足,则在投机域中每一个投机装入处删除指令序列 $I$,在每一个投机检验开始处插入指令序列 $I$ 实现装入下沉。

装入下沉将投机装入指令经过任何相关的条件分支移动到输出被检验的基本块中。一般检验结果有两种可能。经过正常程序代码的"pass"路径,和经过投机恢复代码的"fail"路径。为了保证消除投机不影响程序的语义,需要确定这两条路径产生等价的程序状态。在算法实现中验证使用通过 Omega 算子约束解决技术[3]。

完成以上步骤后,投机消除算法最后完成:

1) 应用不带投机特性的装入指令替换投机域 $R$ 内的投机装入指令。

2) 删除投机域 $R$ 内的每一个投机检验指令。

其中删除投机检验也要把指向恢复代码的控制流边删除,这将使得相应的恢复代码变为不可达,而这些不可达的代码在后面的程序分析中会被检测和消除。

### 2.3 具体算法

算法可以分为 6 步:

第 1 步:将程序 $P$ 中的投机装入指令和投机检验指令分组,放入到各自的投机域中。

第 2 步:对 $P$ 中每一个投机域 $R$,需要验证路径无关性,见定义 2。在投机装入和相关检验之间存在的多条不同的路径,路径无关性要满足沿着任何一条路径与投机装入指令相关指令是相同的。

应用内存二义性消除技术识别内存访问的相关性。所谓内存二义性的消除是指根据给定的程序点上寄存器内容判断程序在执行中是否存在包含地址重叠的两个寄存器。由于在机器代码级缺乏指令的语义,所以判断比较困难。在算法中的实现可以概括为指令检查[4]技术。

应用简单的迭代数据流分析算法,结合程序中的每一个内存引用,以及对子集的访问。分析的基本点来自可执行文件的不同段的产生方式。一般来说,从源模块产生一个目标模块,编译器无法得到其他目标模块的信息,例如数量,大小或链接的顺序,因此无法假设这些域在可执行文件中的最终位置。

分析的基础是下面定义的域集合:

$D = \{stack, global, GOT, num\}$

其中:$stack$ 指栈地址,$global$ 指全局,$GOT$ 指向全局偏移表,$num$ 表示数字常数。全局偏移表为只读域包含 64 位代码或全局数据地址。分析域是该集合的幂集,$P(D)$ 根据包含的子集排序;$(P(D), \subseteq)$ 形成一个完全格,包含最小元素 $\vartheta$(标记不可达的引用)和最大元素 $D$(标记不知的值),以及与操作 $\cup$。

基本块中的指令"$r \mapsto s$" 表示寄存器 $r$ 指向域 $S$ 的集合:

集合 ∪ 操作和与操作符功能一致,在基本块中传播信息。值的传播迭代进行直到遇到固定点,也就是直到任何寄存器的计算集合不再变化。

经过分析以后,通过 $r_1$ 和 $r_2$ 的间接内存访问 $r_1 \mapsto s_1$ 和 $r_2 \mapsto s_2$,如果 $s_1 \cap s_2 = 0$,则可以推出为不相关。

第 3 步:实施装入下沉。如果确定投机域$(L,C)$ 是路径无关的,投机下沉就比较直接:

(1) $\pi$ 是$L$中装入指令到$C$中检验指令之间的任意路径,$S = DEP_L(\pi)$ 表示 $\pi$ 上与 $L$ 相关的指令。

(2) 对每一个投机装入 $I \in L$,删除 $I$ 和 $C$ 中任意检验之间的指令 $S$。

(3) 对每一个检验 $J \in C$,将指令 $S$ 拷贝到 $J$ 的基本块顶部。另外,如果在 $S$ 中有任何非投机指令 $S'$ 且计算值在路径上是活跃并且不经过投机检验指令,则拷贝 $S'$ 到该路径上。完成投机下沉的代码结构如图 2 所示。

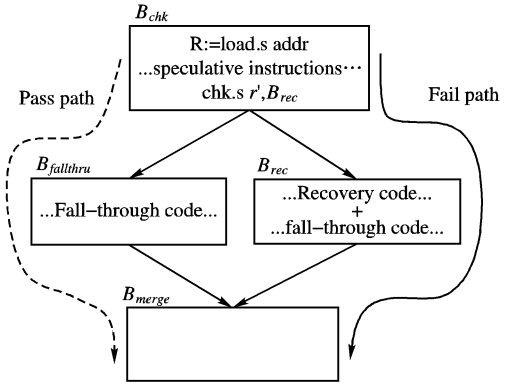



图 2　装入下沉后代码结构

第 4 步:验证装入下沉产生的代码的路径等价以及装入等价性。如图3 所示,基本块 $B_{chk}$ 的投机检验有两种可能输出,如果投机装入完全成功,而不设任何的 NAT 位,则执行路径为 $\pi_{pass} \equiv B_{chk} \rightarrow B_{fallthru} \rightarrow B_{merge}$, 如果投机装入失败,则设置 NAT 位,执行沿着失败路径经过恢复代码 $\pi_{fail} \equiv B_{chk} \rightarrow B_{rec} \rightarrow B_{merge}$。

投机消除的作用是两方面的,首先,消除投机检验指令和失败路径 $\pi_{fail}$,其次 $B_{spec}$ 中的投机指令被转换成没有投机特性的指令,意味着投机代码延迟的异常经过投机消除后不再被延迟,为了保证正确性,必须满足下面两个条件:

(1) [路径等价] 执行路径 $\pi_{pass}$ 和路径 $\pi_{fail}$ 是等价的,对每一个寄存器和内存地址 $x$,$x$ 在 $B_{merge}$ 的入口的值经过 $\pi_{pass}$ 还是 $\pi_{fail}$ 是相同的。

(2) [装入等价] 对在 $B_{chkc}$ 中的投机装入每个内存地址 $y$,在 $B_{rec}$ 中有相应的从 $y$ 的无投机特性的装入指令。

路径等价最简单的情况是除了投机装入被投机消除装入替换,恢复代码与投机代码相同。然而,通常寄存器的值在经过从寄存器 $r$ 投机装入和检验后可能会变化。为了在装入失败后恢复,在重新发布装入指令之前,需要重新计算正确地址。因此,恢复代码需要额外的指令来修正程序的状态。

验证路径等价涉及到沿着成功和失败路径的寄存器和内存内容。在这个过程中,当前对内存的处理方式比较保守:如果成功或失败路径上依赖投机装入的指令中包含任何存储到内存的指令,则认为路径不等价,放弃对该投机域的投机消除工作。这样证明路径等价归根结底是推出在成功和失败路径上寄存器的内容。首先指定逻辑公式 $\Phi$,断言存在与路径不等价的相应程序状态,也就是对某些寄存器 $r$,$r$ 在成功路径上的值与失败路径上的值不同。然后使用强制解决方法试图表示 $\Phi$ 为不可满足的[3]。如果可以做到这样,则得出结论不存在引起路径等价无效的程序状态,因此证明路径等价保持。

装入的等价性验证与路径等价性验证方法非常类似。方法是将投机装入指令与恢复代码中的非投机装入指令结成对,然后应用与上面类似的基于约束验证两种装入中使用的寄存器值是否一致。

第 5 步: 对 $P$ 中每一个投机域 $R$,应用投机消除装入替换 $R$ 中的每一个投机装入。

第 6 步:删除 $R$ 中的每一个投机检验。

### 2.4　投机消除实例

下面是应用投机消除算法对安腾一段二进制代码进行投机消除的实例,投机消除前后的程序在语义上是完全等价的。

实现投机消除的代码实例:

投机代码

```
sub r8 = r8, r2; ;
ld4. s r19 = [r3]
cmp4. lt. unc p0, p15 = r27, r30
(p15) br. cond. dpnt. many 40000000000098b0; ;
adds r18 = 1, r19
shl r17 = r19, 8; ;
shl r16 = r18, 8
sxt4 r15 = r17
sxt4 r14 = r16
shladd r11 = r15, 2, r28; ;
ld4 r10 = [r11]
shladd r9 = r14, 2, r28
chk. s. i r19,  4000000000009a40
ld4 r26 = [r9]; ;
…
4000000000009a40: (  recovery code)
ld4 r19 = [r3]; ;
adds r18 = 1, r19
shl r17 = r19, 8; ;
shl r16 = r18, 8
sxt4 r15 = r17; ;
shladd r9 = r14, 2, r28
ld4 r10 = [r11]
br. many4000000000009a40; ;
```

消除后代码

```
sub r8 = r8, r2; ;
cmp4. lt. unc p0, p15 = r27, r30
(p15) br. cond. dpnt. many 40000000000098b0; ;
ld4 r19 = [r3]
adds r18 = 1, r19
shl r17 = r19, 8; ;
shl r16 = r18, 8
sxt4 r15 = r17; ;
sxt4 r14 = r16
shladd r11 = r15, 2, r28; ;
ld4 r10 = [r11]
shladd r9 = r14, 2, r28
ld4 r26 = [r9]; ;
```

## 3　实验结果

本文的试验基础是二进制代码翻译系统 IATS,该系统是基于 UQBT[5] 的二进制翻译框架开发的,完成对 IA-64 二进制代码静态翻译,生成低级 C 代码。IATS 首先读入二进制文件,经过指令解码和语义映射生成第一级中间表示,这一级中间表示与汇编代码基本等价,为了消除代码机器相关性,需要进行代码提升和优化,生成与机器无关的带有控制结构的高级中间表示,为下一步提升到低级 C 代码做准备。优化代码

消除模块的作用是去掉代码中与源机器相关的谓词执行、投机执行等特性,恢复优化前的逻辑结构,进而简化后面代码转换的工作。

测试程序应用 SPECint-2000 基准测试集中的 7 个例子:bzip2、gzip、mcf、parser、twolf、vortex,以及 vpr,应用 Intel's icc 编译器 8.0,优化级别为 -O2,产生的二进制文件包含大量的控制投机代码。

投机算法的效率可以应用两种方式衡量:数量和质量。比较应用该算法对每一个程序消除投机装入和投机检验的代码比例。经过测试,该算法平均减少了 75% 的投机装入和投机检验代码。比较投机消除前后的 CFG 图的复杂性:计算指令的条数,基本块个数,以及程序中基本块的边数。结果显示经过投机消除以后,指令条数平均减少了 6%,基本块的个数减少 13%,减少的边数 12%。

对投机消除的性能的测试,测量了投机 7 个基准测试程序投机前后的执行时间。程序运行环境是 733 MHz Itanium2 处理器,运行 Intel's icc 8.0,1GB 的主内存和 2GB 的交换空间。执行时间的获得方式如下:每一个二进制运行 5 次,应用 time 命令计时,最大和最小运行时间被放弃。计算剩余三个执行时间的算术平均作为二进制文件的运行时间,额外的标志指示链接程序保持重定位信息。结果显示投机消除二进制程序性能损失平均为 5%。

## 4 结语

文章提出了在保证程序语义条件下将投机代码转换成正常的投机消除代码技术,降低了低级代码的复杂性,在逆向工程中增强了对此类代码的理解,使得更易于应用高效的传统逆向工程技术对代码重建和再工程。实验结果显示在测试程序中可以减少大约 75% 的投机装入和投机检验。

算法还存在可以进一步改进的地方。对于投机指令在软件流水的循环体内部的情况,文中提出的算法就不适用了,因为在这种情况下需要对软件流水循环展开,再进行投机代码的消除。这类问题需要对多种优化结构进行均衡分析,再结合投机恢复代码给予解决。

**参考文献:**


[1] LIAO SS, WANG PH. Post-pass binary adaptation for software-based speculative precomputation[A]. ACM SIGPLAN'02 Conference on Programming Language Design and Implementation(PLDI)[C], 2002.

[2] Intel IA-64 Architecture Software Developer's Manual[Z]. Intel Corp, 2000.

[3] PUGH W. The Omega test: a fast and practical integer programming algorithm for dependence analysis[J]. Communications of the ACM, 1992,35(8):102 - 114.

[4] DEBRAY SK, MUTH R, WEIPPERT M. Alias analysis of executable code[A]. 25th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL-98)[C], 1998.12 - 24.

[5] CIFUENTES C, VAN EMMERIK M, RAMSEY N. The Design of a Resourceable and Retargetable Binary Translator[A]. Proceedings of the Working Conference on Reverse Engineering[C]. IEEE CS Press. Atlanta, USA, 1999.280 - 291.


(上接第 1160 页)

监测主和数据库服务器位于不同的专用机器上时性能会更好,也会有更好的可测量性;体系结构的改善,如收集进程和控制进程在多个监测主上分级构建时,工作元的服务会明显增强。

### 4.2 本地代价

为研究监控系统的工作元代价,对 CPU、内存以及守护进程的网络使用率和性能日志中相同资源的总使用率进行对比。测量显示,守护进程对 CPU 和内存的使用几乎可以忽略,当每隔一秒采样指标并发送时间片的更新时,网络的利用率在 0.1% 以下。

### 4.3 系统特性

*轻量性* 在监控系统中的多线程、同步构建和 socket 通信都设计成轻量型,这些部分不仅能运行在 Windows 加 Entropia DCGrid 上,也能运行在 Linux/UNIX 加 Xtrem Web 上。在监测主端所有的监测进程都是轻量型的;在工作元端,待开发的一种守护进程可插入本监控系统运行在 Windows 或 Linux 的机器上携有工作元的 Xtrem Web 平台上。一些扩展一旦完成,监控系统将可用在由 XtremWeb 工作元和 Entropia 工作元组成的大型混合桌面网格系统中。

*协同工作性* 数据库的不同配置能适应不同监测指标的收集和特殊事件的过滤观察的需要。监控系统对一些现存服务有互操作性,如资源定位服务和预测服务等,为了在运行时能定位网络资源,监控系统和它的相关数据库也能被用在一些后端网格信息服务中。

## 5 结语

准确连续的系统监测与呈现需要在线性能调节与调度优化的支持。本文描述了一个能收集基于沙箱技术的桌面网格数据的监测工具,已实现和评估了基于 DCGrid 平台的一个监测系统,表现出了很好的兼容性。最重要的设计有两点:一是监测工作元使用资源最少;二是系统改变时才明显增加更新数据量。

测试显示,系统能完全准确地实时监测多达 11 000 个工作元,并将性能数据写入文件。当监测主的性能数据写入本地数据库并高精度地收集数据时,系统可从 1 100 个工作元收集性能数据包而不丢失。在低精度(每个工作元间隔几秒甚至几分钟)情况下大量工作元可被实时监测,标志着监测系统有较好的可监测性。监控系统的设计与 Entropia DCGrid 系统有相当的相容性,能很好地融合到现有各种信息服务系统中提供性能数据,是创建复杂桌面网格系统的性能调节和调度优化的基础。最后,该监控系统也将采用加密机制(公钥私钥)来保障数据的机密性,这将是继续研究和扩展的方向。

**参考文献:**


[1] TAUFER M. Inverting Middleware: Performance Analysis of Layered Application Codes in Hight Performance Distributed Computing[D]. Laboratory for Computer Systems, Department of Computer Science, Swiss Federal Institute of Technology (ETH) Zurich, 2002.

[2] 查礼,徐志伟,林国璋,等. 基于 LDAP 的网格监控系统[J]. 计算机研究与发展,2002,39(8):930 - 936.

[3] 刘黎明,熊齐邦. 网格环境中的资源监测[J]. 计算机工程,2003,29(19):124 - 126.

[4] CHANG F, ITZKOVITZ A, KARAMCHETI V. User - level Resource-constrained Sandboxing[A]. 4th USENIX Windows Systems Symposium(WSS 2000)[C]. Seattle, 2000.

[5] CALDER B, CHIEN A, WANG J, *et al.* The Entropia Virtual Machine for Desktop Grids[R]. San Diego, CSE Technical Report CS2003-0773, 2003.

[6] 黄飞赟,方涛,田明杰. 空间信息网格下监测系统体系的研究[J]. 计算机应用,2004,24(8):110 - 116.

[7] CRISTIAN F, FETZER C. The Timed Asynchronous Distributed Systems Model[EB/OL]. http://www.caip.rutgers.edu/~virajb/readinglist/labmeetingcritinas.pdf.