

文章编号:1001-9081(2010)03-0596-04

针对共享内存 SPMD 程序的通信流依赖分析方法

王 骞, 刘晓洁, 梁 刚

(四川大学 计算机学院, 成都 610065)

(wqtn22@163.com)

摘 要:传统数据流依赖分析方法在处理共享内存单指令多数据(SPMD)程序时,不能识别共享数据访问语句所属的处理节点,也不能消除执行顺序不定的控制流对共享数据流依赖造成的影响,从而在分析共享数据依赖时产生精度较低的结果。根据共享内存 SPMD 程序的共享数据别名特性,提出了一种可扩展的共享数据通信流依赖分析方法,并将其应用于一个编译后端分析器原型中。实验表明,该方法较传统分析方法可发现更多精确的共享数据通信流依赖。

关键词:共享内存;单指令多数据;通信流依赖分析;别名分析;控制流分析

中图分类号: TP302 **文献标志码:** A

Method of communication flow dependence analysis for shared-memory SPMD program

WANG Qian, LIU Xiao-jie, LIANG Gang

(College of Computer Science, Sichuan University, Chengdu Sichuan 610065, China)

Abstract: The traditional analysis methods of data flow dependence cannot recognize the processing nodes to which the statements for accessing shared data should belong and also cannot avoid the impact caused by the control flow whose sequence is non-deterministic while dealing with shared memory Single Program Multiple Data (SPMD) program, thus generating the results with a lower accuracy when they analyze shared data dependence. A scalable analysis method of shared data communication-flow dependence was presented according to the alias feature of shared data in shared-memory SPMD program, which was applied in a prototype of back-end analyzer. The experimental results show that the new method can find more accurate shared data communication-flow dependence than the traditional method.

Key words: shared memory; Single Program Multiple Data (SPMD); communication flow dependence analysis; alias analysis; control flow analysis

0 引言

共享内存单指令多数据(Single Program Multiple Data, SPMD)程序使用典型的 fork-join 编程模式,不同处理节点通过临界区操作共用同一份共享数据,传统控制流图缺乏对程序多处理节点的表示能力,各处理节点可通过别名方式访问共享数据的抽象存储单元且执行顺序缺乏同步保证。传统数据流依赖分析方法虽可发现各处理节点对共享数据的访问,却不能区分访问所属的处理节点,也不能准确分析共享数据的依赖关系。

传统控制流图可被扩展以用于表示程序的多个处理节点,Bronevetsky^[1]将每阶段执行相同程序副本的处理节点收集在执行集合中。Strout 等人^[2]使用过程间控制流图表示不同处理节点上的控制流。这些方法主要用于消息传递程序,缺乏单独跟踪某个处理节点对共享数据访问的能力。Gu 等人^[3]将子过程和循环区块从程序中分离出来,对数组进行有针对性的数据流依赖分析,其应用范围较窄。本文将传统控制流图扩展为 SPMD 控制流图,实现了对单个处理节点访问共享数据的跟踪,同时将任意共享数据的特定结构块分离出

来,对其应用通信流依赖分析框架,可精确发现共享数据的通信流依赖。

1 SPMD 程序控制流分析

在进行通信流依赖分析前,引入以下定义。

定义 1 计算操作部署矩阵 D 。 m 维迭代空间 I 的索引向量表示为 $\vec{i} = (i_1, i_2, \dots, i_m)$ ($0 \leq i_k \leq b_k, 1 \leq k \leq m$), 则 I 的总迭代规模为 $g = \prod_{k=1}^m b_k$, 则计算操作部署矩阵 D :

$$D = \begin{bmatrix} d_{11} & \cdots & d_{1p} \\ \vdots & & \vdots \\ d_{s1} & \cdots & d_{sp} \end{bmatrix}; 0 < s \leq \prod_{k=1}^m b_k, 0 < p \leq n$$

其中 D 是 I 的所有计算操作在 n 维处理节点空间 P 上的时空可执行位置的矩阵表示。 D 的任意元素 d_{ij} 为 1 表示处理节点 i 上存在第 j 个被执行的计算操作,为 0 表示该处为空计算操作。

定义 2 通信流依赖。对任意处理节点 i, j 及其执行语句 $stmt_i, stmt_j (i \neq j)$, 若 $stmt_i$ 的执行一定早于 $stmt_j$ 且 $stmt_i$ 与

收稿日期:2009-09-17;修回日期:2009-11-04。

基金项目:国家自然科学基金资助项目(60873246);国家 863 计划项目(2006A01Z435);教育部新世纪优秀人才计划项目(NCET-04-0870);四川省应用基础研究计划项目(05JY029-021-1);四川大学青年科学基金资助项目(校青 07001;校青 07002)。

作者简介:王骞(1986-),男,甘肃张掖人,硕士研究生,主要研究方向:信息安全、智能计算; 刘晓洁(1965-),女,四川成都人,副教授,主要研究方向:网络安全; 梁刚(1976-),男,四川成都人,讲师,博士研究生,主要研究方向:网络安全。

$stmt_i$ 之间存在共享数据依赖,则称存在一个一定通信流依赖,记为 $stmt_i \delta_{ac} stmt_j$;若 $stmt_i$ 的执行可能早于 $stmt_j$ 且 $stmt_i$ 与 $stmt_j$ 之间存在共享数据依赖,则称存在一个可能通信流依赖,记为 $stmt_i \delta_{uc} stmt_j$ 。

矩阵 D 的行元素是各处理节点同时执行的计算操作,列元素是一个处理计算节点的全部计算操作。SPMD 控制流分析建立在计算操作部署矩阵 D 上,为每个处理节点分离出任务基本块、执行组集合及同步点,以构建一个 SPMD 控制流图 SMC。

二元组 $n_{SMC} \langle pid, sid \rangle$ 指示 SPMD 控制流图基本块在矩阵 D 中的部署位置,其中: pid 为基本块节点所属的处理节点; sid 为基本块节点的顺序编号。处理节点数为 n ,迭代规模为 g 的 SPMD 程序, $C_p \langle N_{SMC}^p, E_{SMC}^p \rangle$ 为处理节点 p 的控制流图,则 SPMD 控制流图 $SMC = \bigcup_{p=1}^n C_p, N_{SMC}^p$ 为 C_p 的基本块节点集, $N_{SMC}^p = \bigcup_{s=1}^g n_{ps}, E_{SMC}^p$ 为 C_p 的控制流边集。

SPMD 程序有处理节点子集 $P \{p_i | 1 \leq i \leq t, t \leq g\}$ 。若 P 的全体元素 p_i 在基本块节点 $n_{p_i l}$ 执行相同任务,则集合 $G \{n_{p_i l} | 1 \leq l \leq t\}$ 为 SPMD 程序的一个执行组集合,执行组全集 $SMG = \bigcup_{i=1}^k G_i (k \leq g)$ 。若 P 的全体元素 p_i 在其基本块节点 $n_{p_i l}$ 之后进行控制流同步,在 $n_{p_i l}$ 后插入一个虚拟基本块节点 $n'_{p_i l}, n'_{p_i l}$ 不会出现在 SMC 中,则集合 $S \{n'_{p_i l} | 1 \leq l \leq t\}$ 为 SPMD 程序的一个控制流同步点。函数 $dge_of(set)$ 用于确定集合 set 包含元素的个数。函数 $stmt(n_{SMC})$ 返回执行语句为基本块节点 n_{SMC} 的执行语句。

图1为根据计算操作部署矩阵构建的 SPMD 控制流图,图中并未标出控制流同步点。

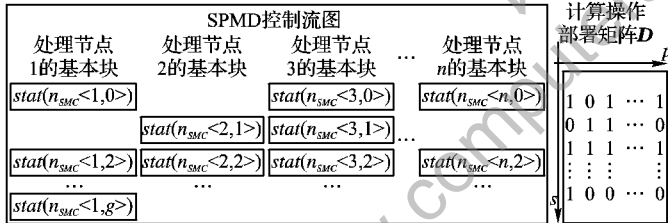


图1 根据计算操作部署矩阵构建的控制流

$alias(sh) = \bigcup_{p=1}^n sh_p \cup \bigcup_{p=1}^n alias(sh_p) (n = dge_of(G_{sh}))$, 函数 $im_of(shname)$ 返回 $shname$ 的共享始元。各类共享数据存在如下初始约束:

1) 静态共享变量 ssv 在其声明执行组 G_{ssv} 处的共享属性为 $\langle ssv, G_{ssv}, static-var, \{0\} \rangle$, 别名变量集合为 $\bigcup_{p=1}^n ssv_p (n = dge_of(G_{ssv}))$;

2) 静态共享指针 ssp 在其声明执行组 G_{ssp} 处的共享属性为 $\langle ssp, G_{ssp}, static-ptr, \{0\} \rangle$, 别名变量集合为 $\bigcup_{p=1}^n ssp_p (n = dge_of(G_{ssp}))$, 且 $\forall n_{ps} \in G_{ssp}$, 由 n_{ps} 开始的对 ssp_p 的任意多层引用可到达的抽象存储单元集合为 \emptyset ;

3) 动态共享变量 sdv 在其分配执行组 G_{sdv} 处的共享属性为 $\langle sdv, G_{sdv}, dynamic-var, \{0\} \rangle$, 别名变量集合为 $\bigcup_{p=1}^n sdv_p (n = dge_of(G_{sdv}))$, 在其释放执行组 G'_{sdv} 处的共享属性为 $\langle sdv, G'_{sdv}, null, \emptyset \rangle$, 别名变量集合为 \emptyset , 且 $\forall n_{ps} \in G'_{sdv}, sdv_p$ 所指向的抽象存储单元为 \emptyset ;

4) 动态共享指针 sdp 在其分配执行组 G_{sdp} 处的共享属性为 $\langle sdp, G_{sdp}, dynamic-ptr, \{0\} \rangle$, 别名变量集合为 $\bigcup_{p=1}^n sdp_p (n = dge_of(G_{sdp}))$, 在其释放执行组 G'_{sdp} 处的共享属性为 $\langle sdp, G'_{sdp}, null, \emptyset \rangle$, 别名变量集合为 \emptyset , 且 $\forall n_{ps} \in G'_{sdp}, sdp_p$ 所指向的抽象存储单元为 \emptyset , 由 n_{ps} 开始的对 sdp_p 的任意多层引用可到达的抽象存储单元集合为 \emptyset 。

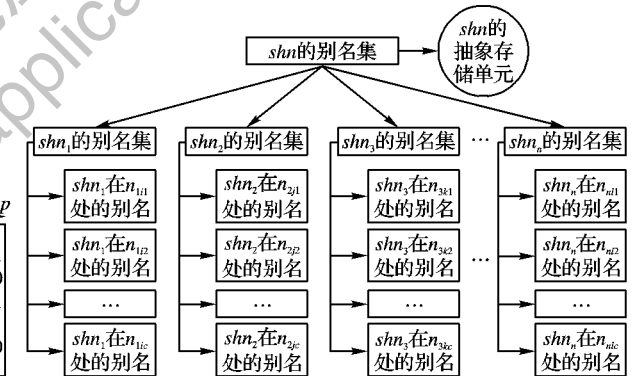


图2 共享始元在各处理节点上的别名

2 共享数据别名分析及通信流依赖识别

假设基本块节点对共享数据的访问都是原子的,忽略过程间调用的影响,共享数据别名分析可发现各处理节点上共享数据的别名,跨处理节点传播数据流状态并识别通信流依赖。

2.1 共享数据别名收集器

别名收集器将共享数据视为外部的、必然存在别名的数据。静态共享数据在声明时就已指定了共享范围;动态共享数据的共享范围为向其分配共享抽象存储单元的处理节点。扩展 BURS 方法^[4]可以用于识别动态共享数据。

四元组 $v_{shm} \langle name, scope, type, index \rangle$ 指示一个共享数据的共享属性; $name$ 是共享数据的名字; $scope$ 是表示共享数据的共享范围的执行组集合; $type$ 是共享数据的类型,其值为 $\{dynamic, static\} \times \{var, ptr\}$; $index$ 表示共享数据的共享范围发生变化的顺序,是一个顺序号集合。函数 $alias(shname)$ 返回共享数据 $shname$ 的别名集合。图2展示了共享始元与其各处理节点上的别名之间的关系。

执行组 G_{sh} 初始声明或分配共享数据时,使用同一个逻辑名字 sh 。处理节点 p 将 sh 重命名为 sh_p , 称 sh 为共享始元,

2.2 共享数据别名传播器

根据基本块节点对共享数据的访问,定义二类共享数据操作: $use(n_{SMC}) = n_{SMC}$ 读取的共享数据集合; $mod(n_{SMC}) = n_{SMC}$ 修改的共享数据集合。

别名传播通过数据流分析到达基本块节点 n_{SMC} 时,设定如下的附加传播规则:

1) \forall 共享数据 $shn \in im_of(shn)$, 若 $v_{shm}(shn).type = var$ && $shn \in use(n_{SMC}) \cup mod(n_{SMC})$, 为 $im_of(shn)$ 生成共享属性 $\langle im_of(shn), n_{SMC}, static-var, gen_shd_id(sh) \rangle$;

2) \forall 共享数据 $shn \in im_of(shn)$, 若 $v_{shm}(shn).type = ptr$ && $shn \in mod(n_{SMC})$ && $stmt(n_{SMC}).right = local_malloc()$, 为 $im_of(shn)$ 生成共享属性 $\langle im_of(shn), n_{SMC}, null, \emptyset \rangle$, 相当于 shn 释放了共享属性, $im_of(shn) = null$;

3) \forall 共享或非共享数据 dt , 若 $dt.type = ptr$ && $dt \in mod(n_{SMC})$ && $stmt(n_{SMC}).right = share_malloc(im)$, 为 im 生成共享属性 $\langle im, n_{SMC}, dynamic-ptr, \{0\} \rangle$, 相当于 dt 开启了共享属性, $im_of(dt) = im$ 。

$local_malloc()$ 是每个处理节点的局部内存分配器。 $share_malloc(im)$ 返回共享始元 im 的抽象存储单元的位置。

$gen_shd_id(shn)$ 向共享数据 shn 返回可用的 $index$ 集合。若同步间隔 $\langle S_k, S_{k+1} \rangle$ 内处理节点 $p_i (1 < i < w)$ 对共享数据 shn 的访问次数为 o_i , 第一个访问基本块为 $n_{j_{i_1}}$, 且 $n_{j_{i_1}} \in S_k$ && $n_{j_{i_1+O_i}} \in S_{k+1}$, 令 $thci$ 为共享始元 shn 当前最小可使用的顺序号, 则 $index$ 为 $\{thci, thci + 1, \dots, thci + e - 1\}$, 函数 $np_of(set)$ 返回执行组 set 包含的处理节点数, $w = np_of((n_{j_{i_1}} \in S_k) \cup (n_{j_{i_1+O_i}} \in S_{k+1}))$ 为同步间隔内访问 shn 的处理节点数, $e = \sum_{p=1}^w \sum_{q=1}^{j_{i_1+O_i}} (dge_of(shn \in use(n_{pq}) \cup (shn \in mod(n_{pq})))$ 为同步间隔内对 shn 进行访问操作的总基本块节点个数。

shn 的别名集为 $alias(sh) = \bigcup_{p=1}^n sh_p \cup \bigcup_{p=1}^n alias(sh_p) (n = dge_of(G_{sh}))$, 对 shn 的别名数据的操作也体现为对共享始元的操作, 通过别名分析, 可以将任何对共享数据的抽象存储单元的操作记录在共享属性集内。图 3 描述了基本块节点访问共享始元的时机。

2.3 识别通信流依赖

共享始元的共享属性集在别名传播完毕时确定, 通过共享属性集和控制流图, 可对基本块间的通信流依赖进行识别。SMC 中已加入了控制流同步点信息, 图 3 中虚线框中的基本块节点仅存在一个可能的访问顺序。

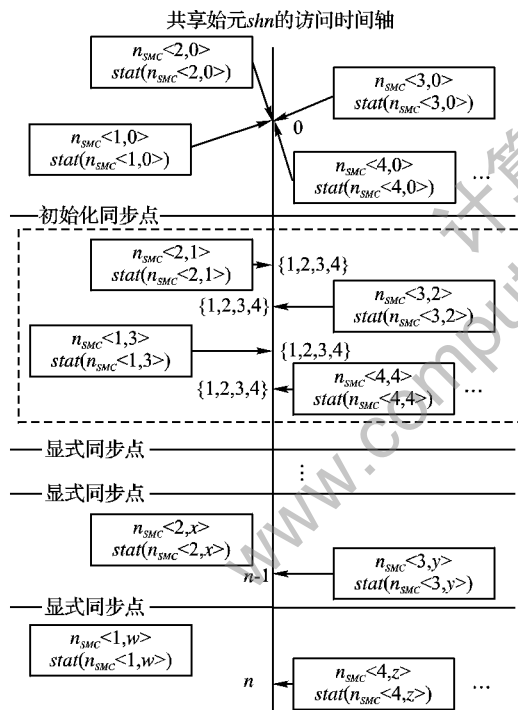


图3 共享数据 shn 的访问顺序和基本块节点的关系

通信流依赖分析函数 $cfd_analysis$ 用来识别某个共享数据的通信流依赖的信息。

算法 1 SPMD 通信流依赖分析框架。

输入: 包含 w 个处理节点的 SPMD 控制流图的结构块

$\bigcup_{i=1}^w SC_i$, 共享数据 shn 在 $\bigcup_{i=1}^w SC_i$ 内的共享属性集 $v_{shn}tset$, 顺序确定共享属性集 $v_{shn}dset$, 顺序不定共享属性集 $v_{shn}nset$ 。

输出: 结构块 $\bigcup_{i=1}^w SC_i$ 内的通信流依赖集 def_cfd 。

if $w > 1$ {

if $sync_between(v_{shn}tset, v_{shn}nset) = false$ {

$v_{shn}nset \cup = v_{shn}tset$

$v_{shn}rset = condition(shn, v_{shn}nset)$

if $v_{shn}rset \neq \emptyset$

{ $v_{shn}dset \cup = v_{shn}rset, v_{shn}nset -= v_{shn}rset$ }

else { $def_cfd = \top$, end }

else { $v_{shn}dset \cup = v_{shn}nset$ }

< $tdef_cfd, v_{shn}mset > = match(shn, v_{shn}tset \cup v_{shn}dset)$

if $v_{shn}mset \neq \emptyset$ {

$def_cfd = v_{shn}mset, v_{shn}dset -= v_{shn}mset$

$def_cfd = tdef_cfd$

else { $def_cfd = \perp$ }

else { $def_cfd = ncf_analysis(SC_1)$ }

函数 $sync_between$ 用于判断两个共享属性集之间是否存在显式的同步点。condition 函数对 shn 的共享属性集进行执行条件检查, 将满足执行条件的可释放的共享属性的 $index$ 域修改为其条件执行顺序, 并返回全部可释放共享属性。match 函数返回共享始元 shn 的共享属性集中匹配的通信流依赖集及其相应的共享属性集。condition 和 match 函数可根据实际应用进行设计和替换, Birch 等人^[5] 结合非线性可变整数测试与递归链分析的方法, 可用于 match 函数中以确定可能匹配的循环依赖。

若 def_cfd 为 \top , 则结构块不存在可以确定访问顺序的基本块, $any_def_cfd \cup \top = \top$ 。若 def_cfd 为 \perp , 则结构块存在确定访问顺序确定的基本块, 但不存在精确匹配的通信流依赖信息, $any_def_cfd \cup \perp = any_def_cfd$ 。若 def_cfd 为有意义的信息, 则分析框架找到了精确匹配的通信流依赖。

3 实验结果及分析

我们开发了并行编译后端分析原型 ChunI, 该原型实现了 SPMD 控制流图的构建功能, 包含根据 SPMD 控制流图进行各种数据流分析的框架。在 ChunI 中添加了通信流依赖分析方法, 使其能在 SPMD 程序的别名传递过程中附加对共享数据的访问跟踪, 并识别通信依赖。

实验内容主要为使用传统方法 Banerjee-Test^[6-7] 和 ChunI 分析开发的共享内存流水线式入侵防御系统。设计了对应于该系统的辅助函数 condition 和 match, 这两个辅助函数即为算法 1 中使用的可替换辅助函数。condition 函数主要影响 ChunI 对基本块访问共享数据顺序的确定; match 函数主要影响分析结果的精度。进行实验时, 首先使用传统数据流分析方法获取该入侵防御系统源码中的共享数据依赖数, 以此作为分析结果的下界; 然后分别应用开启或关闭两个辅助函数的 ChunI 分析该入侵防御系统, 从而测得在四种情况下共享内存通信流依赖分析方法的实际分析能力。

实验结果如表 1 所示。Banerjee Test 方法虽发现入侵防御系统的 146 条通信流依赖, 但仅能精确匹配其中 17 对, 分析结果精度并不高。ChunI 在不开启 condition 函数时, 仅能发现 22 条通信流依赖, 这是由于入侵防御系统未使用显式同步点, 各处理节点执行顺序不定, 使得程序中原本可能发生依赖的基本块, 并没有被算法 1 加入到变量的顺序确定共享属性集中, 从而减少了可供分析的范围。match 函数的分析是建立在顺序确定共享属性集上的, 它仅能从该集合中选出相互匹配的通信流对, 单独开启 match 函数, 并不能扩充顺序确定共享属性集, 此条件下的 ChunI 也不能发现更多的通信流依赖, 但比不开启辅助函数的 ChunI 可识别稍多的匹配通信流对。开启 condition 函数后, 顺序确定共享属性集得到极大扩充, 发行的通信流依赖数大大增加, 即使没有 match 函数的辅助, 依然有 35 对匹配通信流对被识别出来, 单独开启 match 函数时也仅识别出 7 对。开启全部辅助函数的通信流依赖分析方法

的能力大大增强了,在发现了 378 条通信依赖的同时,还识别出了 136 对匹配的通信流。

表 1 两种分析方法所发现的通信流依赖数

分析方法	通信流依赖数	匹配的通信流对数
Banerjee-Test	146	17
关闭辅助函数	22	4
开启 <i>condition</i> 函数	378	35
开启 <i>match</i> 函数	22	7
开启全部辅助函数	378	136

通过对表 1 的分析可发现,处理节点的执行顺序极大地影响了 ChunI 发现通信流依赖的能力。若能较好地定义 *condition* 函数,则 ChunI 可发挥较强的通信流依赖分析能力。*match* 函数针对已存在的通信流依赖进行分析,定义良好的 *match* 函数,使 ChunI 可发现更多精确的匹配通信流对。

根据 ChunI 的分析结果,对该入侵防御系统进行并行化优化,优化前后分别测试不同模块处理 5000 余个数据包的性能,执行性能对比如表 2 所示。访存密集型的模块,如数据包缓存/重组模块和数据包解码模块,优化后性能提升明显,分别达到 37.37% 和 49.09%。这是由于互斥操作降低了共享内存 SPMD 程序在访问临界区时的效率。计算密集型的模块,如模式匹配模块,其计算过程针对局部数据而非共享数据,因而受临界区互斥操作影响较小,优化后性能仅提升 6.27%。统计模块是一个访存频率高、访存局部性小的模块,并行化优化对其影响也较小,性能仅提升了 3.45%。

表 2 并行化优化前后各模块的性能对比

模块名	并行化优化前	并行化优化后	性能提升/%
数据包缓存/重组模块	0.098 2	0.061 5	37.37
数据包解码模块	0.027 5	0.014 0	49.09
模式匹配模块	0.148 4	0.139 1	6.27
统计模块	0.031 9	0.030 8	3.45

传统数据流依赖分析方法无法识别更多的通信流依赖^[8],导致编译器无法对一些隐式可并行化模块,如上述访存密集型模块,部署更多的并行化优化措施,ChunI 则可将这些通信流依赖识别出来,根据它的分析结果对这些模块进行

并行化优化,可有效提升模块的执行性能。

4 结语

识别数据流依赖是循环向量化和并行化的重要前提,而识别通信流依赖也为共享内存 SPMD 程序的共享数据循环向量化合并行化提供了帮助。利用 SPMD 控制流图和共享数据别名分析方法,通信流依赖分析方法可使用不同的辅助函数,有效发现共享数据的别名,将更多的对共享数据的访问加入到可分析的范围,并提高了分析的准确性。

参考文献:

- [1] BRONEVETSKY G. Communication sensitive static dataflow for parallel message passing applications [C]// Proceedings of the 2009 International Symposium on Code Generation and Optimization. Washington, DC: IEEE Computer Society, 2009: 1-12.
- [2] STROUT M M, KREASECK B, HOVLAND P D. Data-flow analysis for MPI programs [C]// 2006 International Conference on Parallel Processing. Washington, DC: IEEE Computer Society, 2006: 175-184.
- [3] GU JUN-JIE, LI ZHI-YUAN. Efficient interprocedural array data-flow analysis for automatic program parallelization [J]. IEEE Transactions on Software Engineering, 2000, 26(3): 244-261.
- [4] 姜伟华,梅超,郭一,等.一种针对多媒体扩展指令集和实际多媒体程序的自动向量化方法[J].计算机学报,2005,28(8):1255-1266.
- [5] BIRCH J, PSARRIS K. Discovering maximum parallelization using advanced data dependence analysis [C]// Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications. Washington, DC: IEEE Computer Society, 2008: 103-112.
- [6] PSARRIS K, KYRIAKOPOULOS K. An experimental evaluation of data dependence analysis techniques [J]. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(3): 196-213.
- [7] 曾利永,杨灿群,黄春. GCC4.1 数据依赖分析器的分析与改进 [J]. 计算机工程与科学, 2006, 28(10): 104-106, 116.
- [8] 胡定磊,陈书明,赵常智,等. SIMD 指令自动向量化编译框架 [EB/OL]. [2009-06-10]. <http://www.ilib2.com/A-%E4%BC%9A%E8%AE%E8%AE%B0%E5%BD%95ID~6336875.html>.

(上接第 595 页)

实施了缓存后与实施前相比,除初次请求的响应时间较长外(由于缓存未命中),后续的请求的响应时间明显缩短,证明了该缓存机制的有效性。

4 结语

缓存作为解决性能问题的有效技术,已被成功应用于许多解决方案中。本文针对企业应用中的非实时类服务,提出一种便于实施的 SOAP 缓存机制,解决了因重复请求造成的不必要的服务性能损失,有效缩短了服务响应时间,对当前企业 SOA 建设中的服务性能优化有一定参考的借鉴作用。在今后的研究中,我们计划根据服务的访问频率、负载和响应速度等实时数据建立动态缓存机制,以优化缓存使用。

参考文献:

- [1] 陈隋和,钟勇.一种增强 Web services 的服务质量和性能的方法 [J]. 计算机应用, 2006, 26(2): 472-475.

- [2] 马晓轩,林学. Web 服务性能优化的研究 [J]. 计算机工程与应用, 2005, 41(8): 19-22.
- [3] NOTTINGHAM M. SOAP optimization modules: Response caching [EB/OL]. [2009-09-15]. <http://lists.w3.org/Archives/Public/www-ws/2001Aug/att-0000/01-ResponseCache.html>.
- [4] HE H, HAAS H, ORCHARD D. Web services architecture usage scenarios [EB/OL]. [2009-09-15]. <http://www.w3.org/TR/ws-arch-scenarios/>.
- [5] YAN F, RU F, ZHONG T, et al. Cache mediation pattern specification: An overview [EB/OL]. [2009-08-20]. <http://www.ibm.com/developerworks/webservices/library/ws-soa-cache-med/>.
- [6] TATEMURA J, HSIUNG W, LI W. Acceleration of Web service workflow execution through edge computing [EB/OL]. [2009-08-20]. <http://www2003.org/cdrom/papers/alternate/P172/p172-tatemura/p172-tatemura.html>.
- [7] 王新房,朱养鹏,邓亚玲.使用 Soap 扩展的 XML Web 服务 [J]. 计算机工程, 2005, 31(7): 138-140.