ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.9, September 2008, pp.2228−2242
DOI: 10.3724/SP.J.1001.2008.02228

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

利用循环分割和循环展开避免 Cache 代价*

刘 利+, 陈 彧, 乔 林, 汤志忠

(清华大学 计算机科学与技术系,北京 100084)

Optimization to Prevent Cache Penalty by Loop Partition and Loop Unrolling

LIU Li+, CHEN Yu, QIAO Lin, TANG Zhi-Zhong

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: E-mail: liuli03@mails.tsinghua.edu.cn

Liu L, Chen Y, Qiao L, Tang ZZ. Optimization to prevent cache penalty by loop partition and loop unrolling. *Journal of Software*, 2008,19(9):2228−2242. http://www.jos.org.cn/1000-9825/19/2228.htm

Abstract: Due to the increasing speed gap between memory system and processor, cache hierarchies have been implemented into memory system, but additional latency (cache penalty) is introduced. This paper presents an algorithm named as prevent cache penalty by loop partition-unrolling (PCPLPU), which can prevent cache penalty in loops by the combination of loop partition and unrolling. Experimental results show that PCPLPU can prevent cache penalty and improve the performance of programs.

Key words: loop partition; loop unrolling; cache penalty; bank conflict

摘 要: 存储系统与处理器之间的速度差距逐渐变大,为此,cache 使用了分级机制,但这也带来了额外的存储延迟(cache 代价).提出一种利用循环分割和循环展开相结合避免 cache 代价的 PCPLPU(prevent cache penalty by loop partition-unrolling)算法.实验结果表明,PCPLPU 算法能够有效避免循环代价,提高程序性能.

关键词: 循环分割;循环展开;cache 代价;bank 冲突

中图法分类号: TP311 **文献标识码**: A

由于存储系统与处理机之间的速度差距变大,分级机制应用到 cache 设计中,但却带来了额外存储访问延迟[1,2].执行程序时,如果存取访问过程中出现了 Bank 冲突,或者存储端口不够用,或者低级 cache 无法快速处理一些特殊情况,额外延迟就产生了,我们称这种额外延迟为 cache 代价(cache penalty,简称代价).循环的执行时间在整个程序的执行时间中占有较大比例,对提高程序的性能来说,避免循环中的代价会有较大潜力.本文提出了一种通过循环分割[3]和循环展开[4,5]的结合来避免代价的 PCPLPU(prevent cache penalty by loop partition-unrolling)算法.本文以 Itanium 2[1]的存储系统为模型,分析了如何结合循环分割和循环展开来避免循环中的代价,Itanium 2 是 IPF(Itanium processor family)[6,7]的第 2 代处理机.

本文第 1 节介绍相关工作.第 2 节介绍本文的相关知识.第 3 节举例说明循环分割和循环展开的结合避免代价的可能性.第 4 节提出 PCPLPU 算法.第 5 节是实验结果和分析.最后是全文的总结.

* Supported by the National Natural Science Foundation of China under Grant No.60573100 (国家自然科学基金)
Received 2005-10-08; Accepted 2006-07-10

1 相关工作

为了提高存储系统的性能,将 cache 的分级机制应用到了存储系统中.如果在存储访问中缺失 cache,存储延迟就出现了同时还带来的额外存储延迟,即代价,即使在命中 cache 时,也可能出现代价.对于缺失延迟,可以通过提高数据的局部性[8]来减少 cache 缺失的次数,也可以通过隐藏存储延迟[9]来降低存储延迟对程序性能的影响.而对于代价,可以避免它的出现.文献[10]分析了 Itanium 2 出现代价的硬件原因,提出了一种能够避免代价的调度算法,但这种算法不是特别针对循环的.文献[11]深入分析了产生代价的条件,并针对这些条件提出了可在模调度[12]中避免代价的 PCPMS 算法,它针对模调度的特殊性,通过指令调度来避免代价,但 PCPMS 算法也存在着一些不足(将在第 2.5 节中介绍).循环分割和循环展开是在循环调度之前,因此,PCPLPU 算法不涉及到指令调度,它不同于 PCPMS 算法.此外,它还能结合 PCPMS 算法,进一步提高程序的性能.

2 相关知识

2.1 循环展开

循环展开通过将循环体代码复制多次实现.循环展开能够增大指令调度的空间,减少循环分支指令的开销.循环展开可以更好地实现数据预取技术[5,13],当循环展开与模调度相结合时,还能实现分数值的启动间距(将在第 2.4 节中介绍).循环展开有一定的开销,既会导致代码膨胀,也会导致调度循环时的寄存器需求变大[5].

2.2 IPF的体系结构特点

指令间存在着依赖,当顺序执行指令时,依赖自然能够得到保证.如果处理机支持乱序执行指令或同时发射多条指令,就需要用硬件保证指令间的依赖.IPF 支持显式并行指令集计算,在一个指令序列中,如果任意指令间都没有依赖,就称这个指令序列为指令组.因此在执行程序的过程中,处理器不用判断同一指令组中的指令间的依赖.但由于存取指令的延迟不固定,而且有时编译器难以准确地分析出它们之间的依赖,因此在执行存取指令时,存储系统需要检查它们的依赖关系.IPF 允许同时发射 6 条指令,但它们在执行顺序上是有先后的.IPF 为软件流水[14]提供了硬件支持,其 cache 分为 3 级:L1,L2 和 L3.L1 分为指令与数据 cache(L1D),其中,L1D 只缓存整数的数据.

2.3 Itanium 2代价的cache条件

Itanium 2 的 L1D 的 cache 块长度是 64 字节,其 L2 有 16 个大小为 16 字节的 Bank.

出现代价的两条存取指令必须具备一定的条件,可将这些条件分为 cache 条件和调度条件.cache 条件主要表现为两条存取指令访问存储系统的相同单元以及是否命中 cache 等,硬件单元的大小称为代价长度(penalty size,简称 S_P).表 1 列出了 Itanium 2 的所有代价类型以及相应的代价长度、cache 条件.

Table 1 The cache conditions of cache penalty on Itanium 2

表 1 Itanium 2 的代价的 cache 条件

Cache penalty kind	Penalty size (bytes)	Conditions on cache
L1D load/store	64	Access the same L1D line and miss L1D
L1D store/load	64	Access the same L1Dline and hit L1D
L1D store/store	64	Access the same L1D line
L2 bank load/load	16	Access the same L2 Bank
L2 bank store/store	16	Access the same L2 Bank
L2 bank store/load	16	Access the same L2 Bank

2.4 软件流水与模调度

软件流水通过同时执行来自不同循环体的指令来加快循环的执行速度.在软件流水中,一个循环体启动于上一循环体结束之前,相邻两个循环体的启动时刻差称为启动间距(initiation interval,简称 II).模调度(modulo scheduling)是一种被广泛采用的软件流水的启发式,在模调度中,所有循环体的调度结果相同,而

且按照一个固定的 *II* 依次启动.模调度的结果由装入(prolog)、核心(kernel)和排空(epilog)这 3 部分组成.在装入部分,前若干个循环体以 *II* 为间隔依次启动,随之出现一个重复模式,即核心,其长度等于 *II*.核心反复执行,直到所有循环体都被启动.离开核心后进入排空部分,直到循环执行完毕.在记录模调度结果时,只需记录一个循环体的各个指令的调度时刻.循环体的最早与最晚调度的指令的调度时刻差被称为调度长度(scheduling length,简称 SL).*II* 和 *SL* 是衡量模调度的重要指标.*II* 的下限(*MII*)由资源限制(*ResMII*)和依赖限制(*RecMII*)决定.

2.5 PCPMS算法

在 PCPMS 算法中,一条存取指令在相邻两循环体的地址差被称为地址增量(address increment,简称 I_A).如果地址增量固定不变,存取指令被称为规则存取指令,否则被称为不规则存取指令.PCPMS 算法认为,规则存取指令始终命中 cache,而不规则存取指令始终缺失 cache.对于可能导致代价的一对存取指令,先执行的被称为代价前者,后执行的被称为代价后者.相同循环体的代价后者与前者的地址差被称为静态地址间隔(static address interval,简称 I_{SA}),如果两条存取指令间出现代价的可能性很小,或者难以分析出它们之间的地址间隔,就认为没有代价,静态地址间隔被标记为∞.表 2 列出了静态地址间隔是∞的一对规则存取指令的条件.对于规则存取指令,出现代价的两条指令可能来自不同循环体;而对于不规则存取指令,出现代价的两条指令在相同循环体中.

Table 2 The conditions of two regular memory instructions whose static address interval are ∞

表 2 静态地址间隔是∞的两条规则存取指令需要满足的条件

Two regular memory instructions	Conditions
A, *B*	*A* is integer memory instructions and *B* is float memory instructions
A, *B*	The address increments of *A* and *B* are different
A, *B*	*A* and *B* access to different data arrays

PCPMS 算法提出了 3 种避免代价的方法:调度法、地址法和顺序法.顺序法不影响模调度结果,地址法不适用于不规则存取指令,调度法会增大 *II*,地址法会增大 *SL*.PCPMS 算法首先要保证不增大 *II*,为此提出了代价组和安全 *II*(safe *II*,简称 SII),当 *SII*>*MII* 时,就需要分拆代价组.在代价组内部使用调度法,在代价组之间使用地址法.但从实验数据来看,PCPMS 算法还是使 *II* 变大了.在本文中,把不分拆代价组时 *SII* 与 *MII* 的比值称为无代价比.

表 3 列出了一对规则存取指令可能出现的代价类型.

Table 3 Implicit cache penalty kind between two regular memory instructions

表 3 一对规则存取指令可能出现的代价类型

Two regular memory instructions	Characters of two memory instructions	Penalty kind
Load, load	Integer	No penalty
Load, load	Float	L2 bank load/load
Load, store	Integer	L1D store/load
Load, store	Float	L2 bank store/load
Store, store	Integer	L1D store/store
Store, store	Float	L2 bank store/store

2.6 循环分割

循环分割主要针对并行系统,将一个执行次数较多的循环分割成若干执行次数较少的子循环,然后把子循环分配到不同的处理机上执行,从而提高循环的执行速度.进入循环前,执行次数已知的循环才能被分割(确定循环).循环分割还能够实现存储优化[3,15,16].PCPLPU 算法只对可被分割的且任意依赖距离固定不变的单层循环或多层循环的最内层作优化,本文称这样的循环为可分循环.

3 循环展开与循环分割的结合能够避免代价

图 1(a)是一个可分循环,其中数组 *a* 是整数数组,它的每个元素占 4 个字节.图 1(b)是可分循环展开 4 次后的新循环,它的循环体由 4 个连续的可分循环体构成,本文称这种展开方式为基本展开.图 1(c)也是可分循环展

开 4 次后的新循环,但与图 1(b)不同的是,它的循环体由 4 个不连续的可分循环体构成,这样展开的实质是将循环分割与循环展开结合起来,把这样的展开称为分割展开.图 1(d)和图 1(e)分别是图 1(b)和图 1(c)的循环体的指令集(省略了分支指令),由于图 1(a)中的数组 *a* 是正数数组,因此,图 1(d)和图 1(e)中的存取指令都是规则存取指令,它们之间的潜在代价发生在 L1D.在图 1(d)中来自不同可分循环体的 load 和 store 指令可能会访问相同的 L1D 块,因此,它们之间可能会出现 L1D store/load 代价.但在图 1(e)中,来自不同可分循环体的存取指令不会访问相同的 L1D 块,因此,它们之间没有代价.由此可见,循环分割展开能够避免代价.图 1(d)的 4 条 load 指令读取的数据基本来自同一个 L1D 块,但图 1(e)的 4 条 load 指令读取的数据来自 4 个不同的 L1D 块,因此,分割展开要求在 cache 中保留更多的数据.但由于 cache 容量越来越大,当分割展开的次数不够大时,不会增加 cache 缺失.大多数处理机都支持数据预取技术,与图 1(d)相比,图 1(e)需要添加数据预取指令,这可能会导致执行更多的冗余预取指令,但利用条件执行或旋转寄存器能够避免执行冗余的预取指令[17].此外,旋转寄存器还能够避免添加多余的预取指令.

根据 PCPMS 算法,图 1(a)可分循环的 *SII* 和 *MII* 分别是 5 和 1,图 1(b)的循环分别是 11 和 3,图 1(c)的循环的分别是 5 和 3,即可分循环.由此可见,基本展开能够降低无代价比,而分割展开能够进一步降低无代价比.

```
for (i=0; i<10000; i++)
    a[i]++;
```

(a) Partitioned loop
(a) 可分循环

```
for (i=0; i<10000; i=i+4) {
    a[i]++;
    a[i+1]++;
    a[i+2]++;
    a[i+3]++;
}
```

(b) Basic loop unrolling
(b) 基本的循环展开形式

```
for (i=0; i<2500; i++) {
    a[i] ++;
    a[2500+i]++;
    a[5000+i]++;
    a[7500+i]++;
}
```

(c) Loop partitioning-unrolling
(c) 循环分割展开后结果

	*r*3=*r*1+4;	*r*5=*r*1+8;	*r*7=*r*1+12;
ld *r*2=[*r*1];	ld *r*4=[*r*3]	ld *r*6=[*r*5]	ld *r*8=[*r*7];
*r*2=*r*2+1;	*r*4=*r*4+1;	*r*6=*r*6+1;	*r*8=*r*8+1;
st [*r*1]=*r*2;	st [*r*3]=*r*4;	st [*r*5]=*r*6;	st [*r*8]=*r*7;
	*r*1=*r*1+16;		

(d) The instructions of loop body in Fig.1(b)
(d) 图 1(b)的循环体的指令集

*r*1=*r*1+4;	*r*3=*r*3+4;	*r*5=*r*5+4;	*r*7=*r*7+4;
ld *r*2=[*r*1];	ld *r*4=[*r*3];	ld *r*6=[*r*5];	ld *r*8=[*r*7];
*r*2=*r*2+1;	*r*4=*r*4+1;	*r*6=*r*6+1;	*r*8=*r*8+1;
st [*r*1]=*r*2;	st [*r*3]=*r*4;	st [*r*5]=*r*6;	st [*r*8]=*r*7;

(e) The instructions of loop body in Fig.1(c)
(e) 图 1(c)图的循环体的指令集

Fig.1 Two forms of loop unrolling
图 1 循环展开的两种形式

4 PCPLPU 算法

与 PCPMS 算法类似,PCPLPU 算法也认为规则存取指令始终命中 cache.分割展开不能避免相同备份循环体的存取指令间的代价,因此,PCPLPU 算法只需对规则存取指令作分析.

4.1 循环分割展开的两种形式

用 *n* 表示可分循环的执行次数,则可分循环的循环体会重复执行 *n* 次.根据可分循环的串行执行顺序去标记这 *n* 个循环体,循环分割会把这些循环体分配到各个子循环中.为了把可分循环和子循环对应起来,就规定在分割前后,循环体的序号不变,可分循环的第 *x* 循环体被分配到任意子循环中后,序号仍然是 *x*.PCPLPU 算法的循环分割的原则是子循环的所有循环体的序号构成等差数列,这样,只用简单变换就能够把可分循环变成子循环.如果等差序列公差为 1,则循环分割被称为连续分割,否则被称为离散分割.在连续分割中,需要考虑存储反依赖.分割可分循环所得的子循环的数目被称为分割展开因子(partition-unrolling factor,简称 F_{PU}).

图 2 给出了分割展开的两种形式,其中,图 2(a)和图 2(b)是两个可分循环,图 2(a)的可分循环可被连续分割,而图 2(b)的可分循环只能被离散分割.图 2(c)是分割展开图 2(a)的可分循环后的结果,分割展开的过程是,先把可分循环分割为两个子循环(分别执行第 1~第 6 000 循环体和第 6 001~第 10 003 循环体),然后按照一定的条件

把这两个子循环合并起来.在所得的结果中共有两个循环,其中,第 1 个循环的循环体由两个可分循环体组成.图 2(d)是分割展开图 2(b)的可分循环后的结果,也是先把可分循环分割为两个子循环(分别执行奇数和偶数循环体),然后把它们合并起来.在所得的结果中共有 3 个循环,其中,第 2 个循环的循环体由两个可分循环体组成.在分割展开的结果中,如果一个循环的循环体由 F_{PU} 个可分循环体组成且任意两个可分循环体都不属于同一子循环,则把这个循环称为 kernel,把 kernel 之前的部分称为 prolog,把 kernel 之后的部分称为 epilog.

```
for (i=0; i<10003; i=i++)
    a[i]++;
```

(a) Partitioned loop 1
(a) 可分循环 1

```
for (i=2; i<10000; i=i++)
    a[i]=a[i−2]+a[i];
```

(b) Partitioned loop 2
(b) 可分循环 2

```
for (i=0; i<4003; i++) {
    a[i]++;
    a[6000+i]++;
}
for (i=4003; i<6000; i++)
    a[i]++;
```

(c) Loop 1 after partitioning-unrolling
(c) 分割展开后的循环 1

```
for (i=2; i<18; i=i+2)
    a[i]=a[i−2]+a[i];
for (i=18; i<10000; i=i+2) {
    a[i]=a[i−2]+a[i];
    a[i−15]=a[i−17]+a[i−15];
}
for (i=10000; i<10016; i=i+2)
    a[i−15]=a[i−17]+a[i−15];
```

(d) Loop 2 after partitioning-unrolling
(d) 分割展开后的循环 2

Fig.2 Two examples of loop partitioning-unrolling
图 2 循环分割展开的两个例子

由此可见,分割展开的结果由两步完成,分别是分割可分循环为若干子循环和将所有子循环合并起来.可以为子循环编号(序号在 1~F_{PU} 之间),规定起始循环体序号越小的子循环的序号越小.对于离散分割展开,各个子循环的启动是有先后顺序的.为了简化设计 PCPLPU 算法,规定当按照子循环的序号从小到大的顺序(对于连续分割展开)或子循环启动的先后顺序(对于离散分割展开)去排列组成 kernel 的循环体的 F_{PU} 个可分循环体时,这些可分循环体的序号形成等差数列,把公差称为分割展开距离(partition-unrolling distance,简称 D_{PU}).由此可得,图 2(c)和图 2(d)的 D_{PU} 分别是 6 000 和−15.连续分割的 D_{PU}≥1,而离散分割展开的 D_{PU}≤1,可以把基本展开看作 D_{PU}=1 的离散分割展开.由于在离散分割中,任意序号差可被 F_P 整除的两个可分循环体会被离散分割到相同子循环中,因此,离散分割展开的 D_{PU} 不被 F_{PU} 整除.在离散分割中,第 x 启动的子循环的序号是

$$(D_{PU}\times(x-1))\%F_{PU}+1.$$

4.2 PCPLPU算法的核心问题

在 kernel 的循环体中,如果两条存取指令来自相同的可分循环体,则称它们是同体存取指令对;否则,称它们是异体存取指令对.PCPLPU 算法不涉及指令调度,它不能避免同体存取指令对之间的代价,但能避免异体存取指令对之间的代价.由此可见,PCPLPU 算法的核心问题是如何利用分割展开因子和分割展开距离来避免异体存取指令对之间的代价.PCPMS 算法涉及到指令调度,PCPLPU 算法在结合 PCPMS 算法后,就能够避免同体存取指令对之间的代价,从而进一步实现存储优化.此外,PCPLPU 算法可以降低无代价比,PCPMS 算法在结合 PCPLPU 算法后,能够得到更好的模调度结果,从而进一步提高程序性能.

4.3 自身代价和交叉代价

可分循环的一条指令在 kernel 中会有多个副本.设 A 和 B 是异体存取指令对,且它们可能存在代价;设 A 是可分循环体中的指令 C 的副本,B 是可分循环体中的指令 D 的副本.如果 C 和 D 是同一条指令,则称 A 和 B 的代价是 C 的自身代价;否则,称 A 和 B 的代价是 C 和 D 的交叉代价.PCPLPU 算法只考虑规则存取指令.对于静态地址间隔是∞的两条存取指令,它们出现交叉代价的可能性很小,因此不用考虑它们之间的交叉代价.根据表 3,规则整数存取指令之间至多出现 L1D 代价,而规则浮点数存取指令之间至多出现 L2 Bank 代价,由于 L1D 代

价和 L2 Bank 代价的条件不同,因此需要对它们分别加以讨论.

4.4 确定prolog,kernel和epilog

在确定 F_{PU} 和 D_{PU} 后,PCPLPU 算法就可以确定唯一的 prolog,kernel 和 epilog.下文分别用 L_{C_o} 和 L_{C_k} 表示可分循环和 kernel 的执行次数.从图 2 可以看出,连续和离散分割展开的结果不同,应当分别讨论.

4.4.1 连续分割展开

在采用连续分割展开时,可以不生成 prolog,则,

$$L_{C\_K}=\min(D_{PU},L_{C\_o}-(k+D_{PU})\times(F_{PU}-1)),$$

其中,k 是可分循环的所有存储反依赖的体差的最大值,在 kernel 中,最早执行的第 1 子循环的循环体是第 1 可分循环体.epilog 中只有一个循环,其循环体由 1 个可分循环体组成(如图 2(c)所示),则这个循环的执行次数是

$$L_{C\_o}-L_{C\_K}\times F_{PU}-k\times(F_{PU}-1).$$

如果 $D_{PU}>L_{C\_o}-(k+D_{\neg PU})\times(F_{PU}-1)$,则把 epilog 看作是第 1 子循环的子循环,那么,epilog 的起始循环体的序号是

$$L_{C\_o}-(k+D_{PU})\times(F_{PU}-1)+1,$$

否则,把 epilog 看作是第 F_{PU} 子循环的子循环,那么,epilog 的起始循环体的序号是

$$D_{PU}\times(F_{PU}-1)+1.$$

4.4.2 离散分割展开

在采用离散分割展开时,$D_{PU}=-1$ 与 $D_{PU}=1$ 在避免自身和交叉代价方面是相同的,而 $D_{PU}=1$ 时的 kernel 执行次数不比 $D_{PU}=-1$ 时的 kernel 的执行次数多,因此在求 D_{PU} 时,可以先求出其绝对值,如果 $|D_{PU}|>1$,则 $D_{PU}=-|D_{PU}|$;否则,$D_{PU}=|D_{PU}|$.下文对于离散分割展开,令 $D_{PU}=|D_{PU}|$,这样就统一了离散分割展开和连续分割展开.当 $D_{PU}=1$ 时,

$$L_{C\_K}=\lfloor L_{C\_o}/F_{PU}\rfloor.$$

在 kernel 中,最早执行的第 1 子循环的循环体是第 1 可分循环体,此时没有 prolog,而 epilog 由若干子循环的结束循环体组成,如果 $\lceil(L_{C\_o}-x+1)/F_{PU}\rceil>\lfloor L_{C\_o}/F_{PU}\rfloor$,则第 x 子循环的结束循环体在 epilog 中.当 $D_{PU}>1$ 时,

$$L_{C\_K}=\lceil L_{C\_o}/F_{PU}\rceil-\lceil(D_{PU}\times(F_{PU}-1))/F_{PU}\rceil.$$

在 kernel 中,最早执行的第 1 子循环的循环体在可分循环中的序号是

$$\lceil D_{PU}\times(F_{PU}-1)/F_{PU}\rceil\times F_{PU}+1.$$

此时,prolog 和 epilog 都存在,如图 2(d)所示,prolog 和 epilog 都可以被分为 $F_{PU}-1$ 个阶段(stage,每个阶段都是循环,阶段从 1 开始编号,越晚执行的序号越大).prolog 的第 x 阶段的循环体由 x 个可分循环体组成,其执行次数是

$$\lceil D_{PU}\times x/F_{PU}\rceil-\lceil D_{PU}\times(x-1)/F_{PU}\rceil.$$

在第 x 阶段中,最早执行的第 1 子循环的循环体在可分循环中的序号是

$$\lceil D_{PU}\times(x-1)/F_{PU}\rceil\times F_{PU}+1.$$

对于 epilog 的第 x 个阶段,其循环体由 $F_{PU}-x$ 个可分循环体组成,循环的执行次数是

$$\lceil(L_{C\_o}-(-D_{PU}\times x)\%F_{PU})/F_{PU}\rceil+\lceil D_{PU}\times x/F_{PU}\rceil-\lceil(L_{C\_o}-(D_{PU}\times(x-1))\%F_{PU})/F_{PU}\rceil-\lceil(D_{PU}\times(x-1))/F_{PU}\rceil.$$

在第 x 阶段中,最早执行的第 $(-D_{PU}\times(F_{PU}-1))\%F_{PU}+1$ 子循环的循环体在可分循环中的序号是

$$(\lceil(L_{C\_o}+D_{PU}\times(x-1))\%F_{PU})/F_{PU}\rceil-\lceil(D_{PU}\times(F_{PU}-1)/F_{PU}\rceil+\lceil D_{PU}\times(x-1)/F_{PU}\rceil)\times F_{PU}-(D_{PU}\times(F_{PU}-1))\%F_{PU}+1.$$

图 3 是离散分割展开的例子.图 3(a)的可分循环只能被离散分割,$F_P=3$.图 3(b)离散分割后的结果,$F_{PU}=3$,$|D_{PU}|=28$.kernel 的执行次数是 3 314,在 kernel 中,第 1 子循环最早执行的循环体是第 58 可分循环.prolog 和 epilog 分别分为两个阶段.对于 prolog,第 1 和第 2 阶段的循环体分别由 1 个或 2 个可分循环体组成,这两个阶段的执行次数分别是 10 和 9.第 1 阶段的第 1 子循环的最早执行的循环体是第 1 可分循环体,第 2 阶段的第 1 子循环的最早执行的循环体是第 31 可分循环体.对于 epilog,第 1 和第 2 阶段的循环体分别由 2 个或 1 个可分循环体组成,第 1 和第 2 阶段的执行次数都是 9.在第 1 阶段中,最早执行的第 2 子循环的循环体是第 9 946 可分循环体;在第 2 阶段中,最早执行的第 2 子循环的循环体是第 9 973 可分循环体.这些结果与本小节的公式相一致.

```
float a[10000];
float b[10000];
…
for (int k=3; k<10000; k++) {
    a[k] = a[k–3] + b[k];
}
```

(a) Original loop / (a) 可分循环

```
//prolog
for (int k=3; k<33; k=k+3) //stage 1
    a[k]=a[k–3]+b[k];
for (int k=33; k<60; k=k+3) {//stage 2
    a[k]=a[k–3]+b[k];
    a[k–28]=a[k–31]+b[k–28];}
//kernel
for (int k=60; k<10000; k=k+3) {
    a[k]=a[k–3]+b[k];
    a[k–28]=a[k–31]+b[k–28];
    a[k–56]=a[k–59]+b[k–56];}
//epilog
for (int k=10002; k<10029; k=k+3) {//stage 1
    a[k–28]=a[k–31]+b[k–28];
    a[k–56]=a[k–59]+b[k–56];}
for (int k=10029; k<10056; k=k+3) //stage 2
    a[k–56]=a[k–59]+b[k–56];
```

(b) After partition-unrolling / (b) 分割展开后的结果

Fig.3 Dispersedly partition-unrolling

图 3 离散分割展开

4.4.3 简单开销模型

如果 kernel 执行的可分循环体数目相对于可分循环的执行次数的比例越大,则 PCPLPU 算法越有效.PCPLPU 算法有编译开销,还会把可分循环的执行过程分为若干个阶段.为此,可以设定一个域值(threshold),当 $L_{C\_K}\times F_{PU}/L_{C\_o}>=threshold$ 成立时,才能作分割展开.

4.5 避免规则整数存取指令间的自身代价和交叉代价

用 S_{L1D} 表示 L1D 的 cache 块的长度.用 x 表示[1,F_{PU}]范围内的任意整数.由表 3 可得,不存在规则整数 *load* 指令的自身代价.设 A 是规则整数 *store* 指令,当$|x\times D_{PU}\times I_{A_A}|>=S_{L1D}$时,不会出现 A 的自身代价,其中,I_{A_A}是 A 的地址增量.由此可得,不出现 A 的自身代价的充分条件是

$$D_{PU}>=|S_{L1D}/I_{A\_A}|.$$

设 B 和 C 是两条不同的规则整数存取指令,且不都是 *load* 指令.当$|I_{SA_BC}+x\times D_{PU}\times I_{A_C}|>=S_{L1D}$且$|I_{SA_CB}+x\times D_{PU}\times I_{A_B}|>=S_{L1D}$时,不会出现 B 与 C 的交叉代价,其中,I_{A_B}表示 B 的地址增量,I_{A_C}表示 C 的地址增量,I_{SA_BC}表示 C 与 B 的静态地址间隔,而 I_{SA_CB}表示 B 与 C 的静态地址间隔,则 $I_{SA\_BC}=-I_{SA\_CB}$.根据表 2,I_{A_B}与 I_{A_C}相等,否则,I_{SA_BC}是∞.令

$$y=\max(-I_{SA\_BC}/I_{A\_C},-I_{\neg\neg SA\_CB}/I_{A\_B},0),$$
$$z_B=\max((S_{L1D}-|I_{SA\_CB}+y\times I_{A\_B}|)/|I_{A\_B}|,0),$$
$$z_C=\max((S_{L1D}-|I_{SA\_BC}+y\times I_{A\_C}|)/|I_{A\_C}|,0),$$

则经过公式推理可得,B 和 C 不出现交叉代价的充分条件是

$$D_{PU}\geq\max(\lceil z_B+y\rceil,\lceil z_C+y\rceil)\text{或 }D_{PU}\leq\min(\lfloor y-z_B\rfloor,\lfloor y-z_C\rfloor).$$

4.6 避免规则浮点存取指令间的自身代价和交叉代价

Itanium 2 的 L2 有 16 个大小为 16 字节的 Bank.设 A 和 B 是到达 L2 的两条存取指令,如果 I_{SA_AB}%256∈[0,15]∪[240,255],则 A 和 B 访问相同 L2 Bank,其中,I_{SA_AB}表示 A 和 B 的静态地址间隔.

根据表 2 和表 3,任意一对静态地址间隔不是∞的规则浮点存取指令都可能出现 L2 Bank 代价.用 A 表示一条规则浮点存取指令,用 x 表示[1,F_{PU}]范围内的任意整数,如果

$$(x\times D_{PU}\times I_{A\_A})\%256\notin[0,15]\cup[240,255] \quad (1)$$

则不会出现 A 的自身代价,其中,I_{A_A}表示 A 的地址增量.可以把 I_{A_A}分解为 2 的幂与奇数相乘,其中,用 $I_{A_A_2}$表

示 2 的幂,用 $I_{A\_A\_odd}$ 表示奇数.当 $I_{A\_A\_2}$≥256 时,公式(1)始终不成立,因此不必分析这类情况.令

$$I_{A\_A\_odd\_M}=I_{A\_A\_odd}\%(256/I_{A\_A\_2}),I_{A\_A\_odd\_M\_C}=(D_{PU}\times I_{A\_A\_odd\_M})\%(256/I_{A\_A\_2}),$$

则 $I_{A\_A\_odd\_M\_C}$ 是整数,且 $I_{A\_A\_odd\_M\_C}\in[0,(256/I_{A\_A\_2})-1]$,于是公式(1)变为

$$((x\times I_{A\_A\_odd\_M\_C})\%(256/I_{A\_A\_2}))\times I_{A\_A\_2}\notin[0,15]\cup[240,255] \quad (2)$$

设 $I_{A\_A\_2}$=16,则 $I_{A\_A\_odd\_M\_C}\in[0,15]$,表 4 列出了 x 和 $I_{A\_A\_odd\_M\_C}$ 是否会导致式(2)不成立,其中,x 的最大值是 7,即最大 F_{PU} 是 8(受篇幅所限,下文中 F_{PU} 的最大值被设为 8),N 表示公式(2)不成立.表 4 省略了 $I_{A\_A\_odd\_M\_C}$>8 的情况,因为 $I_{A\_A\_odd\_M\_C}$ 与 $256/I_{A\_A\_2}-I_{A\_A\_odd\_M\_C}$ 的对应分析结果是相同的.由此可得,当 $I_{A\_A\_2}$=16 且 F_{PU} 是 8 时,公式(2)成立的条件是 $I_{A\_A\_odd\_M\_C}\notin\{0,4,8,12\}$.

Table 4 When $I_{A\_A\_2}$=16, whether formula (2) is false as $I_{A\_A\_odd\_M\_C}$ and x are variable

表 4 当 $I_{A\_A\_2}$=16 时,$I_{A\_A\_odd\_M\_C}$ 和 x 的不同取值是否会导致公式(2)不成立

	x=1	x=2	x=3	x=4	x=5	x=6	x=7
$I_{A\_A\_odd\_M\_C}$=0	N	N	N	N	N	N	N
$I_{A\_A\_odd\_M\_C}$=1							
$I_{A\_A\_odd\_M\_C}$=2							
$I_{A\_A\_odd\_M\_C}$=3							
$I_{A\_A\_odd\_M\_C}$=4				N			
$I_{A\_A\_odd\_M\_C}$=5							
$I_{A\_A\_odd\_M\_C}$=6							
$I_{A\_A\_odd\_M\_C}$=7							
$I_{A\_A\_odd\_M\_C}$=8		N		N		N	

表 5 列出了 $I_{A\_A\_2}$ 为 16 且 F_{PU} 取不同值时的公式(2)成立的条件,确定 F_{PU} 后,就能够根据表 5 找出公式(2)成立时 $I_{A\_A\_odd\_M\_C}$ 的取值范围.例如,当 $I_{A\_A\_2}$=16 且 F_{PU}=8 时,$I_{A\_A\_odd\_M}$ 是[1,31]中的奇数,则 $I_{A\_A\_odd\_M\_C}\notin\{0,4,8,12\}$.设 $I_{A\_A\_odd\_M}$=5,则 $D_{PU}\%(256/I_{A\_A\_2})\notin\{3,7,11,15\}$,可以用一个 32 位的数 d(规定最右一位是第 0 位,设最左一位是第 31 位)来表示这个条件,如果 $D_{PU}\%(256/I_{A_A_2})$不能是 k,则 d 的第 k 位是 0,否则是 1.因此 d 是 EEEE,把 d 称为分割展开状态数(vector to restrict partition-unrolling distance,简称 V_{RPUD}).

Table 5 The condition that formula (2) is true as $I_{A\_A\_2}$=8 and F_{PU} is variable

表 5 当 $I_{A\_A\_2}$=16,F_{PU} 取不同值时式(2)成立的条件

$I_{A\_A\_2}$	F_{PU}	Condition: $I_{A\_A\_odd\_M\_C}\notin$
16	2	{0}
	3 or 4	{0,8}
	5~8	{0,4,8,12}

表 6 列出了当 $I_{A\_A\_2}$ 是 16 且 F_{PU}(最大值是 8)和 $I_{A\_A\_odd}$ 取不同值时的 V_{RPUD},V_{RPUD} 的长度是 $256/I_{A\_A\_2}$ 位.为了把所有 V_{RPUD} 统一起来,就把长度不足 64 位的 V_{RPUD} 自展为 64 位.

Table 6 V_{RPUD} when $I_{A\_A\_2}$=8, F_{PU} and $I_{A\_A\_odd\_M}$ are variable

表 6 当 $I_{A\_A\_2}$=8,F_{PU} 和 $I_{A\_A\_odd\_M}$ 取不同值时的 V_{RPUD}

$I_{A\_A\_2}$	F_{PU}	$I_{A\_A\_odd\_M}$	V_{RPUD}
16	2	1~7	FFFEFFFEFFFEFFFE
	3~4	1~7	FEFEFEFEFEFEFEFE
	5~8	1~7	EEEEEEEEEEEEEEEE

设 B 和 C 是两条静态地址间隔不是∞的规则浮点存取指令,则根据表 2,B 和 C 的地址增量相同,因此,B 和 C 的分割展开状态数是一致的.用 V_{RPUD_B} 表示确定 F_{PU} 后的不出现 B 的自身代价的分割展开状态数,用 $V_{RPUD\_B\_C}$ 表示确定 F_{PU} 后的不出现 B 和 C 的交叉代价的分割展开状态数,令 $k=|I_{SA\_B\_C}/I_{A\_B}|$,则

$$V_{RDPU\_B\_C}=SHR(V_{RDPU\_B},k)\ \&\ SHL(V_{RDPU\_B},k) \quad (3)$$

其中,$SHR(V_{RPUD\_B},k)$/表示把 V_{RPUD_B} 循环右移 k 位,$SHL(V_{RPUD_B},k)$表示把 V_{RPUD_B} 循环左移 k 位,&表示按位与.

在确定 F_{PU} 之后,就能找出任意规则浮点存取指令的自身代价和交叉代价的分割展开状态数,然后再把所有的状态数进行按位与运算,所得到的结果就是整个可分循环的分割展开状态数.

4.7 确定分割展开因子和分割展开距离

由于只有确定了 F_{PU} 才能求出整个 kernel 的分割展开状态数,因此,PCPLPU 算法采用的方法是先确定 F_{PU},再求出 D_{PU}.当 F_{PU} 过大时,除了代码膨胀严重以外,还会导致其他问题.因此,需要确定 F_{PU} 的上限.

4.7.1 根据指令和数据 cache 的局部性以及无代价比确定分割展开因子的上限

- 如果 kernel 的循环体中的指令太多,就会使指令 cache 的局部性变得很差,因此,可以根据可分循环体的指令数和指令 cache 的容量确定 F_{PU} 的一个上限,用 F_{MPU_I} 表示;
- 分割展开会要求 cache 中同时存放更多的数据,过大的 F_{PU} 会使数据 cache 的局部性变差,因此,可以根据可分循环体的存取指令的数量和数据 cache 的 cache 块数量确定 F_{PU} 的一个上限,用 F_{MPU_D} 表示.
- 当 PCPMS 算法结合 PCPLPU 算法时,在无代价比减小到一定程度后,PCPMS 算法对模调度结果的影响就很小了.如果无代价比<1,就没有必要再减小无代价比.设分割展开保证了不出现自身代价和交叉代价,如果 F_{PU} 增大到一个值后,恰好无代价比<=1,这个值也确定 F_{PU} 的一个上限,用 F_{MPU_N} 表示.
- 由 F_{MPU_I},F_{MPU_N} 和 F_{MPU_D} 确定 F_{PU} 的上限(用 F_{MPU} 表示),$F_{MPU}=\min(F_{MPU\_I},F_{MPU\_N},F_{MPU\_D})$.

4.7.2 根据 L1D 代价确定分割展开距离的范围和分割展开因子的上限

由第 4.5 节,根据每个 L1D 自身和交叉代价,都得到一个 D_{PU} 的上/下界,分别用 D_{PU_uppe} 和 D_{PU_low} 表示,则

$$D_{PU}<=D_{PU\_upper} \text{ 或 } D_{PU}>=D_{PU\_low},D_{PU\_low}>=D_{PU\_upper}.$$

为了简化算法的设计,可以认为循环的 D_{PU_low} 是所有自身和交叉代价确定的 D_{PU_low} 的最大值,而循环的 D_{PU_upper} 是所有自身和交叉代价确定的 D_{PU_upper} 的最小值.可以根据 D_{PU_low} 和 D_{PU_upper} 确定 F_{DP} 的一个上限,用 F_{MDP_L1D} 表示,F_{MDP_L1D} 至少应保证 kernel 的执行次数大于 0.如果 $D_{PU\_low}\le 1$ 或 $D_{PU\_upper}>0$,则 D_{PU} 可以是 1,

$$F_{MDP\_L1D}=\lfloor (L_{C\_o}-1)/(k+1)\rfloor+1,$$

否则,

$$F_{MDP\_L1D}=\lfloor (L_{C\_o}-1)/(k+D_{PU\_low})\rfloor+1,$$

其中,采用连续分割展开时,k 是可分循环的所有存储反依赖的体差的最大值;采用离散分割展开时,k 是 0.图 4 的算法能够求解循环的 D_{PU_low} 和 D_{PU_upper}.

```
loop_L1D_penalty_analyze(loop) {
  for (every integer regular memop-pair⟨memop 1,memop 2⟩) {
    if (both memop 1 and memop 2 are load) continue;
    I_SA_12=static-address-interval of memop 1 and memop 2;
    if (I_SA_12==∞) continue;
    D_PU_low_new=low bound of D_PU determined by memop-pair;
  D_PU_upper_new=upper bound of D_PU determined by memop-pair;
    D_PU_low=max(D_PU_low,D_PU_low_new);
    D_PU_upper=min(D_PU_upper,D_PU_upper_new);}
  if (loop can only be partitioned dispersedly) k=0;
  else k=biggest distance of all mem-anti dependences in loop;
  if (D_PU_low<2||D_PU_upper>0)
    F_PU=min(F_PU,⌊(L_C_o−1)/(k+1)⌋+1);
  else F_PU=min(F_PU,⌊(L_C_o−1)/(k+D_PU_low)⌋+1);}
```

Fig.4 Algorithm for analyzing constrains of L1D penalty

图 4 分析 L1D 代价的限制的函数

4.7.3 根据 L2 Bank 代价确定分割展开状态数

当已知 F_{DP} 时,就能得到每一 L2 Bank 自身和交叉代价的分割展开状态数,从而求得循环的分割展开状态数.图 5 的算法能够求出循环的分割展开状态数,当离散分割循环时需要修改 V_{RDPU},以保证 D_{PU} 不能被 D_{PU} 整除.

```
loop_L2 Bank_penalty_analyze(loop) {
  V_RDPUz_res= FFFFFFFFFFFFFFFF;
  for (every regular float memop-pair⟨memop 1,memop 2⟩) {
    I_SA_12=static-address-interval of memop 1 and memop 2;
    if (I_SA_12==∞) continue;
    V_RDPU_new=V_RDPU determined by memop-pair;
    V_RDPU_res=V_RDPU_res & V_RDPU_new;}
   if (loop can only be partitioned dispersedly) {
    k=the lowest common multiple of F_PU and 64;
    Extend V_RDPU to k bits by copying V_RDPU itself;
    Set all bits in V_RDPU whose serial numbers are divided by k to 0;}
  return V_RDPU;}
```

Fig.5 Algorithm for analyzing constrains of L2 Bank penalty

图 5 分析 L2 Bank 代价的限制的函数

4.7.4 确定分割展开距离的最佳上下界

在求出循环的 F_{PU},D_{PU_low},D_{PU_upper} 和 V_{RDPU} 后,就能确定 D_{PU} 的最佳上下界.可以求得 D_{PU} 所能达到的最大值,用 D_{PU_max} 表示,则

$$D_{PU\_max}=\lfloor (L_{C\_o}-1)/(F_{PU}-1)\rfloor .$$

根据第 4.1 节可得,对于离散分割展开,D_{PU} 越小,kernel 的执行次数越大.对于连续分割展开,令 $D_{PU\_min}=\lfloor L_{C\_o}/F_{PU}\rfloor$,则当 $D_{PU}=D_{PU\_mid}$ 时,kernel 的执行次数达到最大值;当 $D_{PU}\le D_{PU\_mid}$ 时,D_{PU} 每减小 1,kernel 的执行次数就减小 1;当 $D_{PU\_mid}<D_{PU}<D_{PU\_max}$ 时,D_{PU} 每增大 1,则 kernel 的执行次数减小 F_{PU}.

图 6 是求 D_{PU} 的最佳上下界的算法,其中,图 6(a)和图 6(b)分别是求上界和下界的算法.

```
get_best_D_PU_low(loop) {
  compute D_PU_max and D_PU_mid;
  k=the length of V_RDPU;
  if (loop can only be partitioned dispersedly
    ||D_PU_Low>D_PU_mid) {
    low=num_first_1_from_r
    (SHR(V_RDPU,D_PU_Low%k));
    D_PU_Low=D_PU_Low+low;}
  else {low=k–num_first_1_from_l
      (SHR(V_RDPU,(D_PU_mid+1)%k))–1;
    upper=num_first_1_from_r
        (SHR(V_RDPU,D_PU_mid%k));
    if (low>uppe×F_PU–L_C_o%F_PU
      ||D_PU_mid–low<D_PU_Low)
      D_PU_low=D_PU_mid+upper;
    else D_PU_low=D_PU_mid–low;
  } if (D_PU_Low<=D_PU_max)
      return D_PU_low; return 0;}
```

(a) Algorithm for computing best low bound of D_{PU}

(a) 求 D_{PU} 的最佳下界的算法

```
get_best_D_PU_upper(loop) {
  compute D_PU_max and D_PU_mid;
  if (loop can only be partitioned dispersedly)
    D_PU_upper=num_first_1_from_r(V_RDPU);
  else {k=the length of V_RDPU;
    if (D_PU_mid>=D_PU_upper)
      D_PU_upper+=num_first_1_from_l
          (SHR(V_RDPU,(D_PU_upper+1)%k))–k+1;
    else {low=k–num_first_1_from_l
        (SHR(V_RDPU,(D_PU_mid+1)%k))–1;
      upper=num_first_1_from_r
          (SHR(V_RDPU,D_PU_mid%k));
      if (low>uppe×F_PU–L_C_o%F_PU &&
        D_PU_mid+upper<=D_PU_upper)
       D_PU_upper=D_PU_mid+upper;
      Else D_PU_upper=D_PU_mid–low;
  }} if (D_PU_upper>0)
      return D_PU_upper; return 0;}
```

(b) Algorithm for computing best upper bound of D_{PU}

(b) 求 D_{PU} 的最佳上界的算法

Fig.6 Algorithm for computing best bounds of D_{PU}

图 6 求 D_{PU} 的最佳上、下界的算法

函数 num_first_1_from_*l* 的作用是从左向右扫描分割展开状态数的每一位,求出第 1 个是 1 的位数的序号;函数 num_first_1_from_*r* 的作用是从右向左扫描分割展开状态数的每一位,求出第 1 个是 1 的位数的序号.用 L 表示循环的分割展开状态数 V 的长度,当 V 的第 k 位是 1 时,如果 $D_{PU}=x\times L+k$,则 kernel 中不会出现 L2 Bank 的交叉和自身代价,其中,x 是非负整数.用 m 表示一个正整数,设需要求整数 n,当 $D_{PU}=n$ 时,不会出现 L2 Bank 的交叉和自身代价.如果 n 是不大于 m 的最大整数,那么求 n 的方法是,先把 V 循环左移 $(m+1)\%L$ 位,然后求出 V 的

从左向右的第 1 个是 1 的位数的序号,用 o 表示,则 $n=m+o-L+1$,即

$$n=m+\text{num\_first\_1\_from\_}r(SHR(V,(m+1)\%L))-L+1.$$

同理,如果 n 是不小于 m 的最小整数,则 $n=m+\text{num\_first\_1\_from\_}l(SHR(V,m\%L))-L+1$.

在图 6(a)所示的算法中,求得的 $D_{PU}\geq D_{PU\_max}$.当循环只能被离散分割或 $D_{PU\_low}\geq D_{PU\_mid}$ 时,D_{PU} 越小,kernel 的执行次数越大.因此,最佳下界 $D_{PU\_Low\_b}=D_{PU\_Low}+\text{num\_first\_1\_from\_}r(SHR(V_{RDPU},D_{PU\_Low}\%k))$,其中,$k$ 是 V_{RDPU} 的长度.当循环只能被连续分割且 $D_{PU\_low}\leq D_{PU\_mid}$ 时,则最佳下界必然是能够避免 L2 Bank 的交叉和自身代价不大于 D_{PU_mid} 的最大数或不小于 D_{PU_mid} 的最小数中的一个.由此可见,图 6(a)的算法能够通过下界找到最优 D_{PU}.同理,图 6(b)所示的算法能够通过上界找到最优 D_{PU}.

4.7.5 确定分割展开因子和分割展开距离

图 7 是最终确定分割展开因子和分割展开距离的算法,其步骤是:

1. 根据指令和数据 cache 的局部性以及无代价比确定 F_{PU} 的上限.

2. 根据图 4 所示的算法,即根据 L1D 代价的限制,确定 F_{PU} 的上限 F_{MPU} 以及 D_{PU_low} 和 D_{PU_upper}.

3. 对于离散分割展开,每个候选 F_{PU} 是不大于 F_{MPU} 的 F_{DP} 的因数;对于连续分割展开,每个候选 F_{PU} 是不大于 F_{MPU} 的自然数.按照从大到小的顺序依次搜索每一个候选 F_{PU},如果能够根据当前的候选 F_{PU} 计算出合理的 D_{PU},就把这个 D_{PU} 作为最终的 D_{PU},把当前的候选 F_{PU} 作为最终的 F_{PU}.

4. 根据当前的候选 F_{PU} 计算 D_{PU} 的过程是,先根据图 5 的算法,即 L2 Bank 代价找出分割展开状态数,然后根据图 6 的算法确定 D_{PU} 的最佳上下界:如果最佳上下界都是 0,则当前候选 F_{PU} 不是最终 F_{PU};否则,根据最佳上下界确定一个最佳 D_{PU},原则是让 kernel 的执行次数达到最大,再根据 kernel 的执行次数与可分循环的执行次数的对比关系决定这个最佳 D_{PU} 是否合理.这就是一个简单的开销模型.

```
determine_partition_unrolling_factor_distance(loop) {
  F_PU=get_biggest_factor_of_partition_unrolling(loop);
  loop_L1D_penalty_analyze(loop);
  while (F_PU>1) {
    V_RDPU=loop_L2Bank_penalty_analyze(body);
    if (V_RDPU==0)  goto L1;
    D_PU_low=get_best_D_PU_low(loop);
    D_PU_upper=get_best_D_PU_upper(loop);
    if (D_PU_upper<=0 && D_PU_low<=0)  goto L1;
    if (loop can only be partitioned dispersedly)
      D_PU=D_PU_upper>0? D_PU_upper: D_PU_low;
else if (D_PU_upper==0||D_PU_low<D_PU_mid||D_PU_upper<D_PU_mid &&
      D_PU_mid-D_PU_upper>=(D_PU_upper-D_PU_mid)*F_PU-L_C_o%F_PU)
      D_PU=D_PU_low>0? D_PU_low: D_PU_upper;
      L_C_k=the loop count of kernel;
    if (trip count of kernel is smaller than threshold)  goto L1;
    break;
L1: if (loop can only be partitioned dispersedly)
      F_PU=the biggest factor of F_DP which smaller than F_PU;
    else F_PU=F_PU-1;}}
```

Fig.7 Algorithm of determining D_{PU} and F_{PU}

图 7 求 D_{PU} 和 F_{PU} 的算法

4.8 PCPLPU算法流程

图 8 是 PCPLPU 算法的流程,它首先判断循环是否能被分割,然后求出 F_{PU} 和 D_{PU}.如果求出的 F_{PU} 是 1,则循环不作分割展开;否则,根据 F_{PU} 和 D_{PU} 生成分割展开后的循环.

```
static D_PU_low, D_PU_upper, D_PU_mid, F_PU, V_RDPU, L_C_o, body;
PCPLPU(loop){
  if (loop can not be partitioned) return false;
  body=the loop body of loop; L_C_o=the loop count of loop;
  D_PU_mid=⌊L_C_o/F_PU⌋; D_PU_Low=0; D_PU_Upper=+∞;
  determine_partition_unrolling_factor_distance(loop);
  if (F_PU==1) return false;
  generate the partition-unrolling result of loop;}
```

Fig.8 PCPLPU algorithm

图 8 PCPLPU 算法

4.9 PCPLPU算法举例

如图 9 所示,图 9(a)是可分循环,其中 *a* 数组是整数数组,每个数据占 4 个字节;*b* 数组是浮点数组,每个元素占 8 个字节;可分循环只能被离散分割,离散分割因子是 3.图 9(b)是可分循环的指令集,省略了分支指令和转换数据类型的指令,其中 op3 和 op8 分别是地址增量为 4 的规则整数 load 和 store 指令,op3 与 op8 的静态地址间隔是–12;op4 和 op10 分别是地址增量为 8 的规则浮点 load 和 store 指令,op4 与 op10 的静态地址间隔是–16.“*r*3(3)”表示 3 个循环体之前计算出来的 *r*3,“*r*3(0)”表示当前循环体计算出来的 *r*3,“*r*4=*r*3(0)+*r*3(3)”表示把 3 个循环体之前计算出来的 *r*3 与当前循环体计算出来的 *r*3 的和放到 *r*4 中,即 op3 与 op6 之间存在体差为 0 和 3 的寄存器输入依赖,这可以用旋转寄存器实现(*r*3(0)和 *r*3(3)对应着相同的逻辑寄存器,但对应着不同的物理寄存器).

下面根据 PCPLPU 算法进行分析:

1. 确定 F_{MPU}.循环体的指令和存取指令数目都不多,则 F_{MPU_I} 和 F_{MPU_D} 不决定 F_{MPU}.可分循环只能被离散分割,分割因子是 3,则 F_{PU} 只能是 1 或 3.当 F_{PU} 是 1 时,无代价比是 3,因此 F_{MPU} 是 3.

2. 分析 L1D 的自身和交叉代价.Op8 是增量为 4 的规则整数 store 指令,根据第 4.5 节,D_{PU}≥16.op3 和 op8 分别是增量为 4 的规则整数 load 和 store 指令,静态地址间隔是–12,则 y=3,z3=10,z8=16,D_{PU}≥19 或 D_{PU}≤–13,即 D_{PU}≥19.不出现 L1D 自身和交叉代价的条件是 D_{PU}≥19,则 F_{MPU_L1D}=910,F_{MPU} 还是 3.

3. 分析 L2 Bank 自身和交叉代价.op4 和 op10 分别是地址增量为 8 的规则浮点 load 和 store 指令,当 F_{MPU} 是 3 时,根据第 4.6 节,op4 和 op10 的自身代价的分割状态数是 7FFEFFFC7FFEFFFC,op4 与 op10 的静态地址间隔是–16,它们之间交叉代价的分割状态数是 1FFBBFF11FFBBFF1,因此,循环的分割状态数是 1FFABFF01FFABFF0,由于循环是离散分割,因此需要把分割状态数自展为 192 位,并把其中所有序号模 3 为 0 的位置为 0,最终得到的分割状态数是

1B68B6D00DB29B6016DA2DB01B68B6D00DB29B6016DA2DB0,

由此求得最佳分割展开因子是 19.最终确定的分割展开因子和距离分别是 3 和 19.

4. 生成分割展开结果.根据第 4.4.2 节,kernel 的执行次数是 6 653,在 kernel 中,最早执行的第 1 子循环的循环体是第 40 可分循环.prolog 和 epilog 分别分为 2 个阶段.对于 prolog,第 1 阶段和第 2 阶段的循环体分别由 1 个或 2 个可分循环体组成,这两个阶段的执行次数分别是 7 和 6.在第 1 阶段中,最早执行的第 1 子循环的循环体是第 1 可分循环;在第 2 阶段中,最早执行的第 1 子循环的循环体是第 22 可分循环.对于 epilog,第 1 阶段和第 2 阶段的循环体分别由 2 个或 1 个可分循环体组成,第 1 阶段和第 2 阶段的执行次数分别是 7 和 5.在第 1 阶段中,最早执行的第 2 子循环的循环体是第 19 961 可分循环;在第 2 阶段中,最早执行的第 2 子循环的循环体是第 19 982 可分循环.最终得到分割展开后的结果如图 9(c)所示.

```
int a[20003];
float b[20003];
…
for (int k=3; k<20000; k++) {
   b[k]=b[k+2]+a[k]+b[k–3];
   a[k] = a[k] + a[k + 3];
}
```

(a) Partitioned loop
(a) 可分循环

```
Op1:   r1=r1+4;
Op2:   r2=r2+8;
Op3:   ld4   r3(0)=[r1];
Op4:   ldfd   f1(0)=[r2];
Op5:   f2(0)=f1(2)+r3(3)+f2(3);
Op6:   r4=r3(0)+r3(3)
Op7:   r5=r1–12;
Op8:   st4   [r5]=r4;
Op9:   r6=r1–16
Op10:   stfd   [r2]=f2(0)
```

(b) Instructions set of the partitioned loop body
(b) 可分循环体的指令集

```
for (int k=3; k<24; k=k+3) {//stage 1
   b[k]=b[k+2]+a[k]+b[k–3];
   a[k]=a[k]+a[k+3];}
for (int k=24; k<42; k=k+3) {//stage 2
   b[k]=b[k+2]+a[k]+b[k–3];
   a[k]=a[k]+a[k+3];
   b[k–19]=b[k–17]+a[k–19]+b[k–22];
   a[k–19]=a[k–17]+a[k–14];}
for (int k=42; k<20000; k=k+3) {
   b[k]=b[k+2]+a[k]+b[k–3];
   a[k]=a[k]+a[k+3];
   b[k–19]=b[k–17]+a[k–19]+b[k–22];
   a[k–19]=a[k–17]+a[k–14];
   b[k–38]=b[k–36]+a[k–38]+b[k–41];
   a[k–38]=a[k–38]+a[k–35];}
for (int k=20000; k<20021; k=k+3) {//stage 1
   b[k–19]=b[k–17]+a[k–19]+b[k–22];
   a[k–19]=a[k–17]+a[k–14];
   b[k–38]=b[k–36]+a[k–38]+b[k–41];
   a[k–38]=a[k–38]+a[k–35];}
for (int k=20021; k<20036; k=k+3) {//stage 2
   b[k–38]=b[k–36]+a[k–38]+b[k–41];
   a[k–38]=a[k–38]+a[k–35];}
```

(c) The result of loop partition-unrolling
(c) 分割展开后的结果

Fig.9 Example of partition-unrolling loop by PCPLPU algorithm
图 9 使用 PCPLPU 算法分割展开循环的例子

4.10 动态分割展开

根据第 4.4 节,确定了可分循环的分割展开模式、分割展开因子和分割展开距离后,就能求出确定分割展开后的结果.分割展开模式可以在编译阶段确定.对于执行次数在编译时已知的可分循环,可以在编译阶段确定其分割展开因子和分割展开距离.根据第 4.7 节,对于执行次数在编译时未知的可分循环,只能在程序执行过程中求出其分割展开因子和分割展开距离,把此时的分割展开称为动态分割展开.在动态循环分割展开中,当分割展开因子在程序运行确定时,kernel 的循环体也在程序运行时确定,这就要求动态调度,这会使 PCPLPU 算法变得很复杂.为此,在我们的实现中,执行次数在编译时未知的可分循环的分割展开因子是在编译阶段确定的,这样就能静态调度 kernel 以及 prolog 和 epilog 的每个 stage.

在动态分割展开中,编译阶段所做的辅助工作有:

1. 确定分割展开模式,分割展开因子(不考虑 L1D 代价的限制)和循环的分割展开状态数(不能是 0);
2. 根据分割展开因子和 L1D 的自身与交叉代价求出循环执行次数的取值范围;

3. 把循环执行次数和分割展开距离作为未知参数,生成分割展开后的结果,并保留可分循环.

在动态分割展开中,程序运行过程中所做的工作有:

1. 如果循环执行次数在取值范围中,转步骤 2;否则,执行可分循环;

2. 根据图 7 的算法求出分割展开距离,由于此时分割展开因子在编译时已确定,图 7 所示算法中的循环只会执行 1 次.如果求出的分割展开因子满足开销模型,则执行分割展开后的结果;否则,执行可分循环.

动态分割展开把编译时的部分开销带入了程序的执行过程中,为了避免动态分割展开导致性能降低,可以把图 7 算法中的域值增大.

5 实验数据分析

我们在 ORC[18]2.1 中实现了 PCPLPU 算法.ORC 是一个开放源码的编译器,它为依赖的分析提供了很好的支持,而且我们已在 ORC 中实现了 PCPMS 算法,这为 PCPLPU 算法的实现提供了很好的基础.我们通过 SPEC 2000 中的部分测试程序,对 PCPLPU 算法的性能进行了测试和分析.编译的结果在 Itanium 2 上执行.

表 7 列出了 PCPLPU 算法、PCPMS 算法以及两种算法相结合的性能加速比.可以看出,PCPLPU 算法最终提高了编译器的性能,这是因为对于可分循环,尽管 PCPLPU 算法不能避免相同循环体的存取指令间的 cache 代价,但却能够避免不同循环体间的存取指令间的 cache 代价.PCPLPU 算法的性能加速比较小的主要原因有:

(1) Itanium 2 的 CPU 与 cache 的速度差相对较小,因此存储优化带来的性能提高也会小些;

(2) 确定循环可以被分割且循环体中有可能导致 cache 代价的存取指令是 PCPLPU 算法的前提条件.满足前提条件的循环中所占的比例不高;

(3) ORC 是研究用编译器,它的各个优化模块都已经设计得比较好了,从而导致剩余的优化机会相对小些.

通过表 7 可以看出,PCPMS 算法和 PCPLPU 算法都能够提高编译器的性能,而且 PCPLPU 算法与 PCPMS 算法结合后,能够进一步优化编译器.表 7 中之所以只列出了部分 benchmark,是因为在其他 benchmark 中可以被分割展开的循环很少,在使用 PCPLPU 算法优化后,性能基本没有发生变化,因此就没有列出.

Table 7 Comparison between PCPMS and the combination of PCPMS and PCPLPU

表 7 PCPMS 算法的单独使用与 PCPMS 算法和 PCPLPU 算法的结合的对比

Benchmark	Speedup of PCPLPU	Speedup of PCPMS	Speedup of the combination of PCPMS and PCPLPU
164.gzip	1.013	1.000	1.016
173.applu	1.013	1.021	1.029
175.vpr	1.006	1.009	1.021
188.ammp	1.008	1.024	1.030
191.fma3d	1.006	1.006	1.008
301.apsi	1.014	1.013	1.021
AV	1.010	1.012	1.021

6 结 论

本文提出了一种利用循环分割和循环展开的结合来避免代价的 PCPLPU 算法.首先在第 3 节举例说明了分割展开避免代价的可行性,再在第 4.2 节提出了两种展开分割模式,然后在第 4.3 节~第 4.6 节分析了如何用分割展开来避免 L1D 和 L2 bank 自身代价和交叉代价,最后在第 4.6 节形成了 PCPLPU 算法.

文献[10]中的调度算法主要针对无环调度.文献[11]中的 PCPMS 算法主要针对循环进行优化,但它是通过指令调度来避免 cache 代价的,但 PCPLPU 算法不涉及到指令调度,而是利用循环展开来避免 cache 代价.实验结果表明,PCPLPU 算法能够有效提高编译器的性能,而且与 PCPMS 算法结合,可以进一步提高编译器的性能.

References:

[1] Intel Corp. Intel Itanium 2 Processor Reference Manual for Software Development and Optimization. Intel Corp Press, 2002.

[2] Intel Corp. Intel Pentium 4 and Intel Xeon Processor Optimization. Reference Manual. Intel Corp Press, 2002.
[3] Chen F, Sha EHM. Loop scheduling and partitions for hiding memory latencies. In: Proc. of the IEEE 12th Int'l Symp. on System Synthesis. 1999. 64−70. http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/6603/17629/00814262.pdf?tp=&isnumber=&amumber=814262
[4] Sarkar V. Optimized unrolling of nested loops. In: Proc. of the 14th Int'l Conf. on Supercomputing. New Mexico: ACM Press, 2000. http://portal.acm.org/citation.cfm?id=335246
[5] Li WL, Liu L, Tang ZZ. Loop unrolling optimization for software pipelining. Journal of Beijing University of Aeronautics and Astronautics, 2004,30(11):1111−1115 (in Chinese with English abstract).
[6] Intel itanium architecture software developer's manual. Revision 2.0. 2001.
[7] Huck J, Morris D, Ross J, Knies A, Mulder H, Zahir R. Introducing the IA-64 architecture. IEEE Micro, 2000,20(5):12−23.
[8] Song YH, Xu R, Wang C, Li ZY. Improving data locality by array contraction. IEEE Trans. on Computers, 2004,53(9):1073−1084.
[9] Liu L, Li WL, Chen Y, Li SM, Tang ZZ. Hiding memory access latency in software pipelining. Journal of Software, 2005,16(10):1833−1841 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/1833.htm
[10] Collard JF, Lavery D. Optimizations to prevent cache penalties for the Intel® Itanium® 2 processor. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO 2003). 2003. 105−114. http://portal.acm.org/citation.cfm?id=776273
[11] Liu L, Li WL, Guo ZY, Li SM, Tang ZZ. Optimization to prevent cache penalty in modulo scheduling. Journal of Software, 2005,16(10):1842−1852 (in Chinese with English abstract). http://jos.org.cn/1000-9825/16/1842.htm
[12] Rau BR. Iterative modulo scheduling. HPL-94-115. Hewlett-Packard Laboratories, 1994.
[13] Callahan D, Kennedy K, Porterfield A. Software prefetching. In: Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM Press, 1991. http://portal.acm.org/citation.cfm?id=106979&coll=portal&dl=ACM
[14] Allan VH, Jones RB, Lee RM, Allan SJ. Software pipelining. ACM Computing Surveys, 1995,27(3):367−432.
[15] Wang Z, Kirkpatrick M, Sha EHM. Optimal two level partition and loop scheduling for hiding memory latency for DSP applications. In: Proc. of the ACM 37th Design Automation Conf. 2000. 540−545. http://portal.acm.org/citation.cfm?id=337571
[16] Wu CC, Chen CF. A loop partition technique for reducing cache bank conflict in multithreaded architecture. Computers and Digital Techniques (IEE Proc.), 1996,143(1):30−36.
[17] Doshi G, Krishnaiyer R, Muthukumar K. Optimizing software data prefetches with rotating registers. In: Hurson AR, ed. Proc. of the 2001 Int'l Conf. on Parallel Architecture and Compilation Techniques. IEEE Press, 2001. 257−267. http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/7564/20612/00953306.pdf?arnumber=953306
[18] Roy J, Sun C, Wu CY. Open research compiler for Itanium processor family (IPF). MICRO-34 Tutorial. ACM Press, 2001. http://www.microarch.org/micro34/tutorials/orc/

附中文参考文献:

[5] 李文龙,刘利,汤志忠.软件流水中的循环展开优化.北京航空航天大学学报,2004,30(11):1111−1115.
[9] 刘利,李文龙,陈彧,李胜梅,汤志忠.软件流水中隐藏存储延迟的方法.软件学报,2005,16(10):1833−1841. http://www.jos.org.cn/1000-9825/16/1833.htm
[11] 刘利,李文龙,郭振宇,李胜梅,汤志忠.避免模调度中代价的优化方法.软件学报,2005,16(10),1842−1852. http://jos.org.cn/1000-9825/16/1842.htm

刘利(1981－),男,四川沐川人,博士生,主要研究领域为指令级并行算法.

乔林(1972－),男,博士,副教授,主要研究领域为并行编译,数据挖掘.

陈彧(1981－),男,博士生,主要研究领域为指令级并行算法.

汤志忠(1946－),男,教授,博士生导师,主要研究领域为计算机系统结构,指令级并行算法,并行编译技术.