多核共享缓存 bank 冲突分析及其延迟最小化

张吉赞[1),2)] 古志民[1)]

[1)](北京理工大学计算机科学技术学院 北京 100081)

[2)](鲁东大学数学与信息学院 山东 烟台 264025)

摘 要 在硬实时多核系统中,共享资源冲突的问题为硬实时任务的最差情况下执行时间(WCET)分析带来了新挑战.虽然现有的共享缓存冲突分析技术在 storage 冲突方面已取得研究进展,但对于 bank 冲突而言,现有研究仍局限于通过界定 bank 冲突延迟上限来分析和处理 bank 冲突.该文通过优化核-bank 映射关系来使硬实时多核系统中的 bank 冲突延迟最小化,即在对 bank 冲突延迟进行分析的基础上,首先通过优化核-bank 之间的映射关系来消除 bank 冲突;若无法消除,则需要寻找能使 bank 冲突延迟最小化的核-bank 映射关系解,并为此设计了一种基于多核总线请求时间序列的 bank 冲突延迟求解算法.最后,文中设计了能够对总线访问延迟进行消重的多核硬实时任务 WCET 估算方法.实验结果表明:文中所提的优化方法可消除这类 bank 冲突或使其延迟最小化,文中所提的 WCET 估算方法与现有估算方法相比可获得更精确的最差情况下执行时间(WCET).

关键词 多核系统;硬实时任务;优化;核到 bank 映射;bank 冲突延迟;最差情况下执行时间

中图法分类号 TP303 **DOI 号** 10.11897/SP.J.1016.2016.01883

Analyzing Bank Access Conflict and Minimizing Bank Conflict Delay for Shared Cache in Multicore

ZHANG Ji-Zan[1),2)] GU Zhi-Min[1)]

[1)](*School of Computer Science & Technology, Beijing Institute of Technology, Beijing* 100081)

[2)](*Department of Mathematics & Information, Ludong University, Yantai, Shandong* 264025)

Abstract Inter-task interferences on the shared resources of hard real-time multicore systems bring a new challenge to WCET analysis. The inter-task interferences related to the shared cache are storage interference and bank access conflict. Up to now, storage interference has been solved well. In existing research, however, the treatment of bank access conflict is only confined to bounding upper-bound of the bank conflict delay suffered by one request. As changing the core-to-bank mapping can change the bank access conflict in a hard real-time multicore system using a bank partitioned shared cache, we optimize core-to-bank mapping to minimize bank conflict delay in this paper. We firstly optimize core-to-bank mapping to eliminate bank access conflict. If cannot eliminate the bank access conflict, we optimize core-to-bank mapping to minimize bank conflict delay. To solve the optimization problem, we design an algorithm using the timing sequences of the bus requests to compute the bank conflict delay on one shared bank. We also design a method to estimate the WCETs of hard real-time tasks in multicore systems, which can reduce the time overlapping among the execution time in pipeline, bus access delay and the latency of memory system. Experimental results demonstrate that our approach of optimizing core-to-bank mapping can minimize bank conflict delay and our approaches to estimate WCET are more effective than existing approaches.

收稿日期:2014-11-16;**在线出版日期**:2015-07-23.**本课题得到国家自然科学基金**(61370062)**资助**.**张吉赞**,**男**,1973 **年生**,**博士研究生**,**主要研究方向为计算机体系结构、计算机网络**. E-mail: zhang_zhao_zhang@163.com.**古志民**(**通信作者**),**男**,1964 **年生**,**博士**,**教授**,**中国计算机学会**(CCF)**会员**,**主要研究领域为多核/众核优化**. E-mail: zmgu@x263.net.

Keywords multicore system; hard real-time task; optimization; core-to-bank mapping; bank conflict delay; worst-case execution time (WCET)

1 引 言

硬实时系统对硬实时任务的执行时间有着严格要求,每个硬实时任务必须在确定的截止期之前完成.硬实时任务的最差情况下的执行时间(WCET)是判断硬实时任务是否能够安全运行的重要依据[1],迄今为止,针对硬实时单核系统的 WCET 估算技术已取得重大的研究进展[2],然而,随着嵌入式多核技术(如 ARM11 MPCore①, QorIQ P4080② 等)在硬实时系统领域的广泛应用,这类硬实时多核系统中往往存在着任务之间可以共享的资源,如共享的片上高速缓存和片上总线等,同时运行的硬实时任务在使用这些共享资源时可能会发生 bank 冲突、总线访问冲突等.这些冲突会给硬实时任务的执行带来不可预测的额外执行时间,这为 WCET 估算带来了新的技术挑战[3].

由于基于单核的传统 WCET 估算技术无法支持对这类冲突的时间分析[4-5],为了获取安全的 WCET,我们在对多核系统上的硬实时任务进行 WCET 估算时,必须重新估算这些冲突对执行时间带来的影响.

目前,多 bank 结构已成为共享缓存设计的主要方向[6-8],例如一个多 bank 结构的 L2 缓存由多个 bank 组成,当多个请求同时到达 L2 缓存的一个 bank 时,只能有一个请求使用这个 bank,其他请求必须等待,此时就发生了 bank 访问冲突.在对 bank 访问冲突的处理上,现有技术(如 Paolieri[9] 和 Yoon[10] 等)主要采用界定每个请求遭受的 bank 冲突延迟上限的方法.这种方法虽然可以简化 WCET 的估算,但需要借助于特殊的总线结构或总线仲裁策略将 bank 冲突延迟限制在一定范围内,如两层总线仲裁策略(two hierarchical bus arbitration)[9]、和谐的轮询总线仲裁策略(harmonic round-robin bus arbitration)[10] 等;然而,其他一些常见的总线结构或总线仲裁策略,如简单轮询策略(pure round-robin arbitration),与这类特殊的总线仲裁策略完全不同,采用上述 bank 冲突延迟上限界定法就无法有效界定每个请求遭受的 bank 冲突延迟上限.

另外,这种界定 bank 冲突延迟上限的方法对硬实时任务的 WCET 估算过高.无论请求遭受到 bank 访问冲突与否,该方法为每个访存请求增加一个额外的 bank 冲突延迟上限.实际上,并不是所有请求都会遭受到 bank 访问冲突,并且即使在一组请求中发生了 bank 访问冲突,每个请求遭受的 bank 冲突延迟也不尽相同,如第 1 个访存请求就不会遭受到 bank 访问冲突.

在硬实时多核系统中,运行在同核上的硬实时任务之间不存在 bank 访问冲突,而在不同核上同时运行的硬实时任务若因为共享某个 bank,则它们之间可能存在 bank 访问冲突.因此,不同的核到 bank 映射方式对应的 bank 访问冲突不同,通过改变核到 bank 映射关系就可改变多核系统中 bank 访问冲突的情况.

本文的主要目的是通过优化核到 bank 的映射关系来优化硬实时多核系统中的 bank 冲突延迟,进而对硬实时多核系统进行 WCET 估算.其中,硬实时多核系统中的 L2 缓存采用了多 bank 结构并进行 column 划分③,共享总线为采用简单轮询总线仲裁策略的时分多路复用(TDMA)实时总线.本文主要贡献如下:

(1) 对采用简单轮询总线仲裁策略的硬实时多核系统进行了 bank 冲突延迟分析,给出了 bank 访问冲突发生的条件和 bank 冲突延迟的计算方法;

(2) 根据 bank 访问冲突发生的条件,首先优化核到 bank 的映射关系以消除 bank 访问冲突.若不能消除 bank 访问冲突,则进一步优化核到 bank 的映射关系使 bank 冲突延迟最小化,并为该优化问题提出了一种基于多核总线请求时间序列的 bank 冲突延迟求解算法;

(3) 提出了多核硬实时任务的 WCET 估算方法,该方法综合考虑了任务在流水线上的执行时间、访问存储系统的时间和总线访问延迟之间的相互影响关系.

本文第 2 节介绍相关工作;第 3 节给出硬实时多核系统模型,包括多核结构和应用模型;第 4 节分

① ARM11 MPCore Processor. http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php, 2012,6,18

② Freescale QorIQ P4080 Processor, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080, 2012,6,18

③ Chiou D, Rudolph L, Devadas S, Ang B S. Dynamic cache partitioning via columnization. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.5764, 2013,3,14

析 bank 冲突延迟；第 5 节提出优化问题，并设计求解算法；第 6 节设计多核硬实时任务的 WCET 估算方法；第 7 节给出实验环境及实验验证结果；第 8 节得出结论.

2　相关工作

在硬实时多核系统中，与 L2 缓存相关的任务间冲突主要包括 storage 干扰和 bank 访问冲突. 由于总线访问冲突与 storage 干扰或 bank 访问冲突之间存在着相互影响，因此在分析 storage 干扰或 bank 访问冲突时，一般需要结合总线访问冲突进行分析.

一些现有的研究成果将共享总线设计和共享缓存划分技术结合起来，对请求遭受的总线访问延迟和 bank 冲突延迟进行分析，但采用了界定每个请求遭受的延迟上限的方法. 如 Paolieri 等人[9]在其工作中，提出了一种二层总线仲裁的多核结构，用以界定每个总线请求遭受的总线访问延迟和 bank 冲突延迟，共享 L2 缓存采用缓存划分或 bankization 划分，以消除 storage 干扰或 bank 访问冲突；在采用 bankization 划分时，要求任务独占分配的 bank 以消除 bank 访问冲突，受 bank 数目的影响，这种方法受限制于硬实时多核系统的工作负荷. 然而该方法仅适用于这类特殊的多核结构，且 WCET 估算方法采用了界定延迟上限的方法. 再如在 Yoon 等人[10]的可调 WCET（tunable WCET）、和谐的轮询总线仲裁策略等工作中，共享 L2 缓存采用二级划分结构，整个缓存被划分成多个 bank，每个 bank 又进一步被划分成多个 column. 其中，核向 bank 做映射，硬实时任务向 column 做映射且独占分配的 column 以消除 storage 干扰. 在优化时采用了界定 bank 冲突延迟和总线访问延迟上限的方法. 然而该方法仅适用于和谐的轮询总线仲裁策略，同时采用界定延迟上限的方法造成 WCET 估算过高.

另有一些研究成果是将 storage 干扰分析和总线访问冲突分析结合起来进行 WCET 估算，但在分析时却没有考虑 bank 访问冲突问题. 如 Andrei 等人[11]和 Rosén 等人[12]提出的 TDMA 总线延迟分析和 storage interference 延迟，优化了总线调度策略，其特点是，在这种 TDMA 总线中使用静态调度分析，总线时槽被静态地分配给不同的核. Chattopadhyay 等人[13]提出了融合共享缓存和总线的 WCET 分析框架，在分析总线访问延迟时让循环的开始与总线调度周期的第一个时槽对齐，同时考虑到 L2 缓存的 storage 干扰，因此进行反复迭代与调整，直到结果稳定. 虽然这种方法比 Andrei 等人[11]提出的方法效率高，但是该方法对 WCET 值估算仍过高. Kelter 等人[14]通过界定 TDMA 偏移量（TDMA offset）上界的方法来进一步提高分析效率，具体采用了 Chattopadhyay 等人[13]提出的分析框架来估算 WCET，并用全局收敛性分析（global convergence analysis）来界定 TDMA 总线偏移量的上限. Kelter 等人[15]静态分析（static analysis）了 TDMA 总线给请求带来的总线访问延迟，并给出了形式化证明. 同时结合 storage 干扰分析估算了多核系统的 WCET. Chattopadhyay 等人[16]提出了一种多核系统的 WCET 分析框架，改进了文献[13]对循环结构的处理. 在分析总线访问延迟时不再让循环的开始与总线调度周期的第一个时槽对齐，而是根据执行上下文（execution context），令请求的总线访问延迟为可能遭受的最大总线访问延迟. Li 等人[17]分析了并行任务的 WCET，首先使用信息序列图（Message Sequence Chart，MSC）将并行任务的生命期分成重叠（overlapping）和非重叠（nonoverlapping）两部分，对于重叠部分的分析，采用 Chattopadhyay 等人[13]提出的分析框架和 Kelter 等人[14]提出的界定总线偏移量的方法.

还有一些研究成果将分析重点仅放在 storage 干扰上，均没有考虑 bank 冲突延迟和总线访问延迟对硬实时任务 WCET 的影响. 如 Yan 等人[18]根据线程的程序控制流信息，计算线程在共享 L2 指令缓存上的 storage 干扰. Chen 等人[19]通过指令的取指时间关系，分析了进程在共享缓存上的 storage 干扰. Ding 等人[20]提出了动态锁指令缓存以消除 storage 干扰，该方法可灵活锁定循环结构对应的缓存空间. Liu 等人[21]应用锁缓存技术来消除 storage 干扰.

3　硬实时多核系统模型

3.1　嵌入式多核模型

如图 1 所示的一个嵌入式多核处理器含有 N_{core} 个同构的有序（in-order）核，表示为 $C=\{c_1,c_2,\cdots,c_{N_{\text{core}}}\}$. 每个核有自己私有的第一级数据缓存和指令缓存. 由所有核共享使用的第二级缓存（unified L2 cache）采用多 bank 结构，由 N_{bank} 个大小相等的 bank 组成，表示为 $B=\{b_1,b_2,\cdots,b_{N_{\text{bank}}}\}$，完成一次请求

需要的时间为 L_M 个时钟周期(cycles). 使用 Yoon 等人[10]提出的缓存两级划分方法将每个 bank 进一步划分成相等的 N_{column} 个 columns. 连接 L2 缓存和核的实时总线是全双工 TDMA 总线,该实时总线采用简单轮询调度策略,每个总线调度周期有 L_{round} 个等长的总线时槽,表示为 $R=\{s_1,s_2,\cdots,s_{L_{\text{round}}}\}$. 每个总线时槽的长度等于总线完成一次请求所需要的时间,表示为 L_B 个时钟周期. 假设 L_M/L_B 是整数,那么一个请求完成一次 L2 缓存访问至少需要 (L_B+L_M) 个时钟周期,设为 L_{lat}. 核到总线时槽的映射是一一映射,且把核 $c_i(\in C)$ 映射到总线时槽 $s_i(\in R)$ 上. 请求访问 L2 缓存,发生缺失时需要访问主存,假设请求访问主存需要的时间为 L_{L2penal} 个时钟周期.

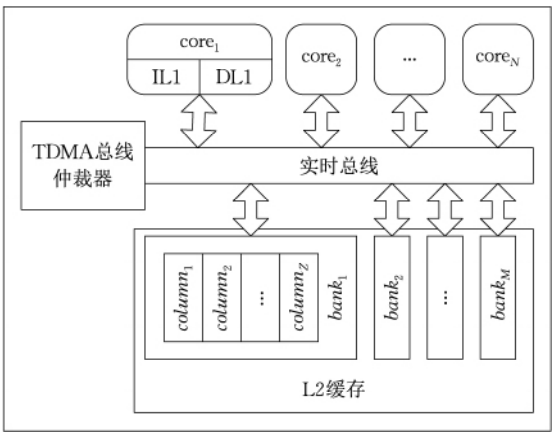


图 1 一个嵌入式多核处理器结构

3.2 多任务应用模型

假设一组硬实时任务已经被分配到 N_{core} 个核上,这些任务在执行过程中不能在核间迁移,同核上的任务将按顺序执行,那么某时间段内最多有 N_{core} 个任务均匀分布在 N_{core} 核上且并发执行. 在确定任务到核的分配后,需要确定核需要的 L2 缓存大小,设 HT_i 是分配到核 c_i 的任务集合,任务 $\tau_j(\in HT_i)$ 需要的 L2 缓存大小为 $Size_j$ 个 columns,则核 c_i 需要的 L2 缓存大小为 $Size_{c_i}=\max(Size_j\mid 1\leq j\leq n_i)$ 个 columns,其中 n_i 为集合 HT_i 中的任务数.

采用类似于 Yoon 等人[10]所提方法来作核到 bank 的映射和硬实时任务到 column 的映射,按照核需要的最大 L2 缓存大小向 bank 作映射,当多个核共享使用某个 bank 时,在这些核上同时运行的任务之间有可能存在 bank 访问冲突. 在作任务到 column 的映射时,任务独占分配给它的 column. 故不存在 storage 干扰.

另外,采用 Li 等人[17]所提方法来处理任务间的共享代码和任务间的通信. 如果多个任务共享使用某个函数或程序段,则为每个任务复制一份以取消任务间的代码共享. 若任务间需要通信则采用邮箱机制来取消由同步带来的影响.

4 Bank 冲突延迟分析

设有 $N_{cb}^k(1\leq N_{cb}^k\leq N_{\text{core}})$ 个核共享 $b_k(\in B)$,表示为 $C_{b_k}=\{c_1',c_2',\cdots,c_{N_{cb}^k}'\}$,对应的总线时槽为 $R_{b_k}=\{s_1',s_2',\cdots,s_{N_{cb}^k}'\}$,其中,$C_{b_k}\subseteq C$,$R_{b_k}\subseteq R$. 令 bcd_{ij} 表示运行在 $c_i'(\in C_{b_k})$ 上的硬实时任务在第 j 个总线周期中遭受的 bank 冲突延迟. 如图 2 所示,当 $i\neq 1$ 时 bcd_{ij} 可以用式(1)表示:

$$bcd_{ij}=\begin{cases}bcd_{(i-1)j}+L_M-(s_i'-s_{(i-1)}')\cdot L_B, & >0\\ 0, & \text{其他}\end{cases}\tag{1}$$

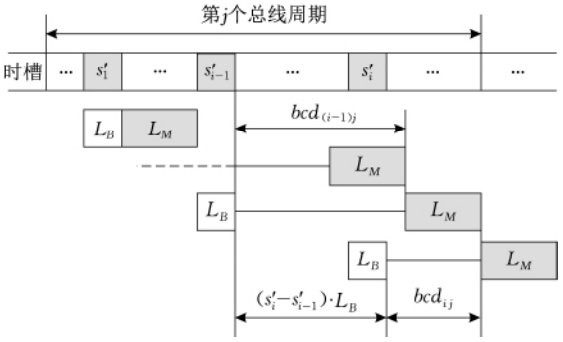


图 2 运行在核 $c_i'(\in C_{b_k})(i\neq 1)$ 上的任务在第 j 个总线周期上遭受的 bank 冲突延迟

在第 1 个总线周期中,核 $c_1'(\in C_{b_k})$ 所遭受的 bank 冲突延迟为 0,即 $bcd_{11}=0$. 一般地,bcd_{1j} 如图 3 所示,当 $c_p'(\in C_{b_k})$ 上的硬实时任务在第 q 个总线周期上有访问 b_k 的请求,且该请求是运行在核 c_1' 上的硬实时任务在第 j 个总线周期上访问 b_k 的前一个请求,bcd_{pq} 是运行在核 c_p' 上的硬实时任务在第 q 个总线周期上遭受的 bank 冲突延迟,s_p' 是其对应的总线时槽,则运行在核 c_1' 上的硬实时任务在第 j 个总线周期上遭受的 bank 冲突延迟可以表示为

$$bcd_{1j}=bcd_{pq}+L_M-(j-q-1)\cdot L_{\text{round}}\cdot L_B-(L_{\text{round}}-s_p'+s_1')\cdot L_B,$$

进一步简化为式(2).

$$bcd_{1j}=\begin{cases}bcd_{pq}+L_M-((j-q)\cdot L_{\text{round}}+s_1'-s_p')\cdot L_B, & >0\\ 0, & \text{其他}\end{cases}\tag{2}$$

定义两核 $c_i(\in C)$ 和 $c_j(\in C)$ 之间的模距离为它们对应总线时槽的最小距离,即 $d_{ij}=\min(|s_j-s_i|,$

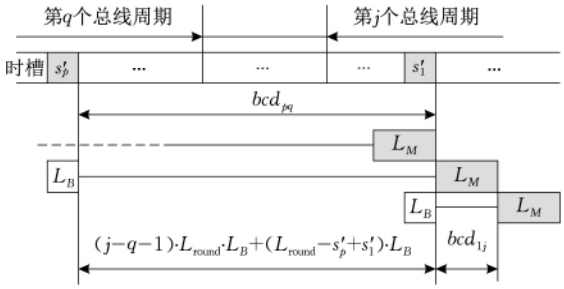


图 3　运行在 $c'_1(\in C_{b_k})$上的任务在第 j 个总线周期上遭受的 bank 冲突延迟

$L_{\text{round}}-|s_j-s_i|$). 为了判断一个 bank 上是否存在 bank 访问冲突，给出定理 1 如下.

定理 1.　$\forall c'_i,c'_j\in C_{b_k}$，若 $d_{ij}\geq L_M/L_B$，则在 b_k 上不存在 bank 访问冲突.

证明.　由于 $d_{ij}\geq L_M/L_B$，在式(1)中，$L_M-(s'_i-s'_{(i-1)})\cdot L_B\leq 0$，在式(2)中，$L_M-((j-q)\cdot L_{\text{round}}+s'_1-s'_p)\cdot L_B\leq 0$. 由此可知，在访问 b_k 的所有请求中，后一个请求所遭受的 bank 冲突延迟一定小于或等于前一个请求所遭受的 bank 冲突延迟. 由于在访问 b_k 的请求序列中，第一个请求的 bank 冲突延迟为 0，又因为 bank 冲突延迟具有非负性，因此所有请求的 bank 冲突延迟都为 0，即在 b_k 上不存在 bank 访问冲突.　证毕.

5　优化核到 bank 的映射

5.1　优化问题的形式化描述

由定理 1 可知，在一个核到 bank 的映射中，若映射到任一个 bank 上的任意两个核之间的模距离大于等于 L_M/L_B，则在整个系统上不存在 bank 访问冲突. 用 x_{ik} 表示 $b_k(\in B)$是否有 column 分配给 $c_i(\in C)$，若有(即 $c_i\in C_{b_k}$)，则 $x_{ik}=1$，否则 $x_{ik}=0$. 用 $ncol_{ik}$ 表示 b_k 分配给 $c_i(\in C)$的 column 数目，如果 $x_{ik}=1$，则 $ncol_{ik}>0$，否则 $ncol_{ik}=0$，由于硬实时任务独占分配给它的 column，因此 $ncol_{ik}$ 是整数. 以 x_{ik} 和 $ncol_{ik}$ 为决策变量，优化核到 bank 的映射使系统不存在 bank 访问冲突的形式化描述如下.

目标函数：

$$d_{ij}\geq L_M/L_B,\ \forall c'_i,c'_j\in C_{b_k},\ \forall b_k\in B \tag{3}$$

约束：

$$N_{\text{bank}}\cdot N_{\text{column}}\geq\sum_{i=1}^{N_{\text{core}}}Size_{c_i} \tag{4}$$

$$Size_{c_i}=\sum_{k=1}^{N_{\text{bank}}}ncol_{ik}\cdot x_{ik},\ \forall c_i\in C \tag{5}$$

$$\sum_{i=1}^{N_{\text{core}}}ncol_{ik}\cdot x_{ik}\leq N_{\text{column}},\ \forall b_k\in B \tag{6}$$

$$Size_{c_i}\geq 0,\ ncol_{ik}\geq 0,\ \forall c_i\in C,\ \forall b_k\in B \tag{7}$$

其中，约束(4)是指在硬实时多核系统中 L2 缓存满足硬实时任务的需求；约束(5)是指映射分配给每个核的 column 数需要满足每个核的需求；约束(6)是指在每个 bank 上分配的 column 数不超过 bank 的大小；约束(7)是非负约束.

下面利用 N_{core}、L_M/L_B 和 N_{column} 参数来探讨通过优化核到 bank 的映射来消除 bank 冲突的判据.

定理 2.　已知 N_{core}、L_M/L_B 和 N_{column}，且 L2 缓存的容量大小满足需求，若 $\lfloor N_{\text{core}}/(L_M/L_B)\rfloor>N_{\text{column}}$，则 bank 访问冲突可以通过优化核到 bank 的映射去消除(证明过程见附录).

用 $Size(C_i)$ 表示集合 $C_i(\subseteq C)$ 中，核需要的总 column 数. 结合核需要的总 column 数，可以给出当 $L_M/L_B=2$ 时能够通过优化核到 bank 的映射消除 bank 冲突的判据，如定理 3.

定理 3.　已知 C、$N_{\text{core}}\bmod 2=0$、$L_M/L_B=2$、N_{column} 和 $Size_{c_i}>0$，$1\leq i\leq N_{\text{core}}$. 将 C 分割成两个互不相交的子集 C_0 和 C_1 且满足：在任一子集中的任意两个核之间的模距离大于等于 L_M/L_B. 若在每个子集中能够找到一个核集合 $C_{si}(\subseteq C_i)$，$0\leq i\leq 1$，且满足 $Size(C_{si})\geq Size(C_i)\bmod N_{\text{column}}$，则 bank 访问冲突可以通过优化核到 bank 的映射去消除(证明过程见附录).

定理 2 和 3 中的条件是判断 bank 冲突是否可以消除的充分条件，在许多应用场景中并不是所有硬实时系统的 bank 访问冲突都能够通过优化核到 bank 的映射消除，例如，在一个 6 核的硬实时多核系统中，L2 缓存被划分 4 个大小相等的 bank，每个 bank 又被划分成 8 个大小相等的 column，核到总线时槽的映射及每个核需要的 L2 缓存的大小如表 1 所示. 在这个例子中，使用上述优化方法对核到 bank 的映射进行优化，就不能使共享每个 bank 的任意两核之间的模距离都大于等于 L_M/L_B. 在这种情况中就无法通过上述优化方法来消除 bank 访问

表 1　硬实时多核系统的应用场景举例

核	对应的总线时槽	缓存大小/column
c_1	s_1	4
c_2	s_2	2
c_3	s_3	2
c_4	s_4	16
c_5	s_5	7
c_6	s_6	1

冲突,但可通过优化核到 bank 的映射使多核系统遭受的 bank 冲突延迟最小化.

设 $Nroc_i$ 是运行在核 $c_i(\in C)$ 上的硬实时任务需要的总线周期数,$Nrob_k=\max(Nroc_i \mid \forall c_i' \in C_{b_k})$ 是共享 b_k 的硬实时任务需要的最大总线周期数.根据式(1)和(2),发生在 b_k 上的所有 bank 冲突延迟可以表示为 $\sum_{j=1}^{Nrob_k}\sum_{\forall c_i' \in C_{b_k}} bcd_{ij}$.由此,可得出式(8).

$$\min\left(\sum_{j=1}^{Nroc_i} bcd_{ij} \mid \forall c_i \in C\right) \Leftrightarrow \min\left(\sum_{k=1}^{N_{\text{bank}}}\sum_{j=1}^{Nrob_k}\sum_{\forall c_i' \in C_{b_k}} bcd_{ij}\right) \tag{8}$$

计算任务遭受的 bank 冲突延迟需要关注硬实时任务的执行特性,如访问 L2 缓存的时间、L2 缓存的地址等,即需要关注硬实时任务的主存块到 L2 缓存的映射以及硬实时任务映射到 L2 缓存的哪些 column上等.为了对 WCET 进行安全估算,当多核共享一个 bank 时,假设这些核上的硬实时任务发出的访问 L2 缓存的请求总是访问这个共享 bank.在此基础上,通过优化核到 bank 的映射来最小化每个硬实时任务遭受的 bank 冲突延迟,根据式(8),该优化问题的形式化描述如下.

目标函数:

$$\min\left(\sum_{k=1}^{N_{\text{bank}}}\sum_{j=1}^{Nrob_k}\sum_{\forall c_i' \in C_{b_k}} bcd_{ij}\right) \tag{9}$$

约束:

$$N_{\text{bank}} \cdot N_{\text{column}} \geq \sum_{i=1}^{N_{\text{core}}} Size_{c_i} \tag{10}$$

$$Size_{c_i} = \sum_{k=1}^{N_{\text{bank}}} ncol_{ik} \cdot x_{ik},\ \forall c_i \in C \tag{11}$$

$$\sum_{i=1}^{N_{\text{core}}} ncol_{ik} \cdot x_{ik} \leq N_{\text{column}},\ \forall b_k \in B \tag{12}$$

$$Size_{c_i} \geq 0,\ ncol_{ik} \geq 0,\ \forall c_i \in C,\ \forall b_k \in B \tag{13}$$

5.2 优化问题求解

目标函数(9)与目标函数(3)的区别在于在目标函数(3)中不需要计算 bank 冲突延迟,而目标函数(9)中则需要计算 bank 冲突延迟.

5.2.1 计算 bank 冲突延迟

根据前面对 bank 冲突延迟的分析,计算 bank 冲突延迟,需要事先确定多核硬实时任务的总线请求时间序列.本文组合使用 Chronos[22] 和 lp_solve① 来获取同核上的硬实时任务的总线请求时间序列,用 $RQ_{c_i}(c_i \in C)$ 表示核 c_i 访问 L2 缓存的总线请求时间序列,bank 冲突延迟的计算过程主要由以下 3 部分组成.

(1)计算总线访问延迟,确定请求访问总线的时间.设 $t_{j-1}, t_j(\in RQ_{c_i'})$ 是来自核 $c_i'(\in C_{b_k})$ 的两个相邻总线请求(分别表示为 rq_{j-1}, rq_j)所对应的总线请求时间,bad_{j-1} 为请求 rq_{j-1} 遭受的总线访问延迟.若 $t_j > (t_{j-1}+bad_{j-1})$,$rq_j$ 遭受的总线延迟表示为 $bad_j=(L_{\text{round}} \cdot L_B+(s_i'-1)L_B-t_j \bmod (L_{\text{round}} \cdot L_B)) \bmod (L_{\text{round}} \cdot L_B)$;否则,$bad_j=L_{\text{round}} \cdot L_B$.

(2)确定当前 bank 冲突延迟所在的总线周期.设 RT_{kn} 为 N_{cb}^k 个总线请求时间序列中的当前总线请求时间的集合,令 $t_{\min}=\min(t_j \mid \forall t_j \in RT_{kn})$ 是 RT_{kn} 中的最小值,则当前总线周期的开始时间为 $t_{\min}-t_{\min} \bmod (L_{\text{round}} \cdot L_B)$.

(3)计算 bank 冲突延迟.如果当前总线周期为第 1 个总线周期,则 RT_{kn} 中第 1 个在该总线周期内访问总线请求遭受的 bank 冲突延迟为 0,否则根据式(2)计算;而 RT_{kn} 中在该总线周期内其他的访问总线请求所遭受的 bank 冲突延迟根据式(1)计算.

算法 1 给出了发生在 b_k 上的 bank 冲突延迟的计算方法.s_i' 为对应的总线时槽.在算法 1 的输出结果中,$T\_b\_delay[i]$ 是运行在核 $c_i'(\in C_{b_k})$ 上的硬实时任务所遭受的总 bank 冲突延迟,以便为估算硬实时任务的 WCET 做准备.第 1、2 行初始化,$current\_q[i]$ 表示当前处理的总线请求时间,$used[i]$ 标记是否可以从请求序列里取第 1 个请求,若 $used[i]=\text{True}$,则表示可以从请求序列里取第 1 个请求.第 5 行判断是否可以从请求序列中取出第 1 个请求到 rq,第 8~14 行计算总线访问延迟;第 15 行更新 $current\_q[i]$ 为当前请求访问总线的时间,为计算 bank 冲突延迟做准备,并将请求标记为处理.第 19、20 行确定总线调度周期,$round1$ 是该总线调度周期的开始时间.第 22 行判断请求是否落在当前总线调度周期内.第 24、25 行计算当前总线周期内第 1 个请求的 bank 冲突延迟,第 27、28 行计算当前总线周期内其他请求的 bank 冲突延迟.第 30 行更新 $T\_b\_delay[i]$,此时,$T\_b\_delay[i]$ 是运行在核 c_i' 上的硬实时任务截止目前遭受的所有 bank 冲突延迟.第 35 行为计算在下一个总线周期中第 1 个请求所遭受的 bank 冲突延迟做准备.第 37 行计算在 bank b_k 上发生的所

① Lp solve version 5.5. http://www.comp.nus.edu.sg/~rpembed/chronos/download.html, 2012, 10, 22

有 bank 冲突总延迟.

算法 1.　计算发生在 b_k 上的 bank 冲突延迟.

输入：C_{b_k}，$RQ_{c_i'}$（$c_i' \in C_{b_k}$），L_{round}，s_i'

输出：发生在 b_k 上的各核 bank 冲突总延迟 $Total\_delay[k]$，核 c_i'（$\in C_{b_k}$）所遭受的 bank 冲突总延迟 $T\_b\_delay[i]$

1. $Total\_delay[k]=0$;
2. $T\_b\_delay[i]=0$, $current\_q[i]=0$, $used[i]=$True, $1\le i\le N_{cb}^k$;
3. WHILE(存在一个 $RQ_{c_i'}$ 不为空) DO
4. FOR($i=1$; $i<=N_{cb}^k$; i++) DO
5. IF($used[i]==$True) THEN
6. 从 $RQ_{c_i'}$ 中取第 1 个请求到 rq;
7. 删除 $RQ_{c_i'}$ 中的第 1 个请求;\\更新 $RQ_{c_i'}$
8. IF($rq<=current\_q[i]$) THEN
9. $busdelay=L_{\text{round}}*L_B$;
10. ELSE
11. $busdelay=(s_i'-1)*L_B-rq \bmod (L_{\text{round}}*L_B)$;
12. $busdelay=L_{\text{round}}*L_B+busdelay$;
13. $busdelay=busdelay \bmod (L_{\text{round}}*L_B)$;
14. END IF
15. $current\_q[i]=rq+busdelay$;
16. $used[i]=$False;
17. END IF
18. END FOR
19. 在 $current\_q[N_{cb}^k]$ 中找最小值，存储到 M_q;
20. $round1=M\_q-M\_q \bmod (L_{\text{round}}*L_B)$;
21. FOR($i=1$; $i<=N_{cb}^k$; i++) DO
22. IF($current\_q[i]<=(round1+(s_i'-1)*L_B)$)THEN
23. IF($current\_q[i]$ 是第 1 个请求) THEN
24. $B\_delay=Init\_delay-(round1+s_i'*L_B)$;
25. IF($B\_delay<0$) THEN $B\_delay=0$;
26. ELSE
27. $B\_delay=B\_delay+L_M-(s_i'-pre)*L_B$;
28. IF($B\_delay<0$) THEN $B\_delay=0$;
29. END IF
30. $T\_b\_delay[i]=T\_b\_delay[i]+B\_delay$;
31. $pre=s_i'$;
32. $used[i]=$True;
33. END IF
34. END FOR
35. $Init\_delay=round1+B\_delay+L_M+pre*L_B$;
36. END WHILE
37. $Total\_delay[k]=\sum_{\forall c_i' \in C_{b_k}} T\_b\_delay[i]$;
38. RETURN $Total\_delay[k]$, $T\_b\_delay[i]$;

5.2.2　优化问题的求解算法

设 Z_k 是共享 b_k（$\in B$）的所有硬实时任务之间模距离小于 L_M/L_B 的模距离的数. 若 $Z_k=0$，则共享 b_k 所有硬实时任务之间的模距离都大于等于 L_M/L_B，根据定理 1，在该 bank 上不存在 bank 访问冲突，否则，在该 bank 上可能存在 bank 冲突.

算法 2 给出了该优化问题的求解算法. 第 4～25 行按照 $c\_seq[\ ]$ 依次做核到 bank 的映射. 根据式(1)和(2)可知，bank 冲突延迟具有积累性，在做核到 bank 映射时，映射到一个 bank 上的核应尽可能的少；另外，算法 1 在计算 bank 冲突延迟时不考虑一个 bank 中的 column 在地址上的区别，因此核到 bank 的映射过程可以简化如下：$c\_seq[\ ]$ 中的核依次向 bank b_1 映射，b_1 分配完后，再向 b_2 映射，b_2 分配完后，向 b_3 映射，依次类推，$c\_b\_mapping[N_{\text{core}}][N_{\text{bank}}]$ 存放当前核到 bank 的映射关系. 第 26 行计算所有的 Z_k，第 27 行判断该映射是否存在 bank 访问冲突. 若存在 bank 访问冲突，第 32 行调用算法 1 计算发生在每个 bank 上的 bank 冲突延迟. 第 36 行计算总的 bank 冲突延迟. 第 38、39 行更新最优结果. 第 43～48 行回溯搜索解空间. 第 50～54 行是主过程，在第 50 行根据定理 2、3 进行判定，若 bank 冲突不能消除，则在第 52 行调用 $F\_M\_Mapping(n)$ 求解.

算法 2.　优化核到 bank 的映射关系，使 bank 冲突延迟最小.

输入：C，N_{core}，L_M，L_B，B，N_{bank}，N_{column}，RQ_{c_i}，$Size_{c_i}$，$\forall c_i \in C$

输出：最小的 bank 冲突延迟（M_delay）、相应的核到 bank 的映射（$M\_mapping$）

1. 设置 Z_k、M_delay 初值、$used[]$ 为 False;
2. FUNCTION $F\_M\_Mapping(n)$
3. IF ($n>N_{\text{core}}$) THEN
4. $n\_bank=1$, $n\_col=N_{\text{column}}$;
5. WHILE ($i<=N_{\text{core}}$) DO
6. 将核 $c\_seq[i]$ 在核集 C 中对应的序号存放在 j 中，需要的 column 数存放在 n_core;
7. IF ($n\_core>=n\_col$) THEN
8. WHILE ($n\_core>=n\_col$) DO
9. $c\_b\_mapping[j][n\_bank]=n\_col$;
10. $n\_core=n\_core-n\_col$;
11. n_bank++;
12. $n\_col=N_{\text{column}}$;
13. END WHILE
14. IF($n\_core==0$) THEN i++;
15. ELSE
16. $c\_b\_mapping[j][n\_bank]=n\_core$;
17. $n\_col=n\_col-n\_core$;
18. i++;

```
19.        END IF
20.      ELSE
21.        c_b_mapping[j][n_bank]=n_core;
22.        n_col=n_col-n_core;
23.        i++;
24.      END IF
25.    END WHILE
26.    计算 Z_k,1≤k≤N_bank;
27.    IF(所有的 Z_k 都为 0) THEN
28.      M_delay=0;
29.    ELSE
30.      FOR (k=1; k<=N_bank; k++) DO
31.        IF (Z_k>0) THEN
32.          调用算法 1 计算发生在 bank b_k 上的 bank 冲
           突延迟 b_delay[k];
33.        END IF
34.      END FOR
35.    END IF
36.    b_delay=Σ_{k=1}^{N_bank} b_delay[k];
37.    IF(b_delay<M_delay) THEN
38.      M_delay=b_delay;
39.      M_mapping[ ][ ]=c_b_mapping[ ][ ];
40.    END IF
41.    RETURN
42.  END IF
43.  FOR (i=1; i<=N_core; i++) DO
44.    IF( !used[i]) THEN
45.      c_seq[n]=c_i; used[i]=True;
46.      F_M_Mapping(n+1); used[i]=False;
47.    END IF
48.  END FOR
49. END FUNCTION
    //以下为主过程
50. 利用定理 2、3 判断 bank 冲突是否可以消除;
51. IF(bank 冲突不能消除) THEN
52.    F_M_Mapping(1);
53. END IF
54. RETURN M_delay, M_mapping;
```

6 WCET 估算

6.1 WCET 估算的预备知识

Theiling 等人[23]提出的共享缓存的抽象解释(abstract interpretation)分析法,是将指令根据访问共享缓存是否命中分成:Always-Hit(AH)、Always-Miss(AM)、PerSistence(PS)和 Not-Classified(NC)四类. AH 是指访问共享缓存总是命中的,AM 是指访问共享缓存总是缺失的,PS 是指第 1 次访问共享缓存是缺失而以后的访问都是命中的,而其他情形则属于 NC 类.

一个五级流水线模型[24]由取指(IF)、译码(ID)、执行(EX)、写回(WB)和提交(CM)组成. 在取指阶段中,按指令在程序中的顺序将指令从存储系统中依次取出,存放到取指缓存(I-buffer);在译码阶段中,将取指缓存中的指令进行译码操作并按在程序中的顺序发送到 ROB(Re-Order Buffer). 在执行阶段,ROB 中的指令发送到相应的执行单元进行执行. 对于 Load 指令,执行阶段只计算有效存储地址,在写回阶段取操作数. 在写回阶段,一方面 Load 指令从存储系统中取操作数,另一方面将执行阶段的执行结果写回 ROB. 在提交阶段,指令按照在程序中的顺序提交. Li 等人[25]提出了执行图(Execution Graph)的概念,该执行图描述了控制流图(CFG)的一个基本块(basic block)在五级流水线模型上执行状态.

本文设计的多核多任务 WCET 估算方法是在 Chronos 的基础上,增加了对多核共享资源冲突延迟语义的分析支持. Chronos 是单核硬实时任务的开源 WCET 估算工具,该工具可对二进制执行文件进行反汇编,以形成控制流图及执行图,并利用该图对指令在流水线阶段间的依赖关系进行分析处理,以及估算基本块的最差情况下执行时间. 一般地,利用 Chronos 可获得如下内容:(1)基本块的最差情况下执行时间;(2)任务的控制流图;(3)基本块中每个请求的总线请求时间;(4)指令的 AH、AM、PS 和 NC 分类等.

6.2 WCET 估算方法

设 $c_i(\in C)$的执行时间是指运行在 c_i 上某硬实时任务的执行时间. 设 $T_{c_i}^{p}$ 是 c_i 在流水线上的执行时间,$T_{c_i}^{m}$ 是 c_i 访问主存所需要的时间,$T_{c_i}^{\mathrm{L1}}$ 是 c_i 访问 L1 缓存所需要的时间,nq_{c_i} 是 c_i 访问 L2 缓存的次数,$D_{c_i}^{\mathrm{bus}}$ 是 c_i 遭受的所有总线访问延迟,$D_{c_i}^{\mathrm{bank}}$ 是 c_i 遭受的所有 bank 冲突延迟,c_i 在最差情况下的执行时间可以表示为 $WCET_{c_i}=T_{c_i}^{p}+T_{c_i}^{m}+T_{c_i}^{\mathrm{L1}}+nq_{c_i}\cdot L_{lat}+D_{c_i}^{\mathrm{bus}}+D_{c_i}^{\mathrm{bank}}$,其中,$(T_{c_i}^{p}+T_{c_i}^{m}+T_{c_i}^{\mathrm{L1}}+nq_{c_i}\cdot L_{lat})$可以直接用单核 WCET 估算工具估算,$D_{c_i}^{\mathrm{bank}}$ 可以用算法 1 计算得到. 令 $WCET'_{c_i}=T_{c_i}^{p}+T_{c_i}^{m}+T_{c_i}^{\mathrm{L1}}+nq_{c_i}\cdot L_{lat}+D_{c_i}^{\mathrm{bus}}$. $WCET_{c_i}$ 可以用式(14)表示.

$$WCET_{c_i}=WCET'_{c_i}+D_{c_i}^{\mathrm{bank}} \tag{14}$$

由于多核多任务在流水线、请求访问总线和请

求访问存储系统中可以并发执行，那么某个任务在流水线上的执行时间、总线访问延迟和存储系统访问时间之间可能存在着时间重叠问题. 另外，由于核与 TDMA 总线时槽之间已确立对应关系，在计算多核总线访问延迟时，总线访问冲突延迟可转换为请求等待自己对应的总线时槽.

为此，我们在 Chronos 基本块最差执行时间分析模块的基础上，增加了对多核总线访问延迟、时间消重等语义的支持，实现了下面的算法 3.

算法 3 给出了估算一个基本块(记为 blk)最差情况下执行时间的方法，该算法是在 Chronos 分析工具中实现的. 由于 PS 指令在第 1 次执行中是缺失的且在后续执行中总是命中的，若一个基本块在循环结构中，它的第 1 次执行和后续执行的最差情况下执行时间是不同的. 在算法 3 中，用 first 标识基本块是否为第 1 次执行，$first=0$ 表示该基本块的第 1 次执行，$first=1$ 表示该基本块的非第 1 次执行. 令 $offset$ 为基本块开始执行时对应的总线偏移量，若基本块的开始时间为 t_b，对应的总线偏移量可以表示为 $offset=t_b \bmod (L_{\text{round}} \cdot L_B)$，且 $0 \leq offset < L_{\text{round}} \cdot L_B$，对于不同的 $offset$ 值，基本块有一个最差情况下执行时间与之对应，存放在 $MET[first][blk][offset]$ 中. $T_{stage(i)}(sta)$ 和 $T_{stage(i)}(fin)$ 分别是指令 i 在 $stage$ 阶段的开始时间和完成时间，若指令 i 在 $stage$ 阶段需要访存，则 $T_{stage(i)}(sta)$ 为请求申请总线的时间. $T_n(fin)$ 是基本块最后一条指令的完成时间，$T_1(ready)$ 是基本块的第一条指令的准备时间. 第 1～10 行定义计算总线访问延迟的函数 $Com\_bdelay()$，pre 是前一个请求访问总线的时间，其值为对应流水阶段的开始时间与总线访问延迟的和. 在第 2～8 行计算总线访问延迟，若当前请求申请总线的时间小于或等于前一个请求访问总线的时间，则消重，总线访问延迟为 $(L_{\text{round}} \cdot L_B)$ 个时钟周期(第 3 行). 否则，在第 6、7 行计算总线访问延迟. 第 13 行先利用 Chronos 流水线分析对基本块 blk 进行分析(包括 L1 缓存分析)，可得到指令在各个流水线阶段上的执行时间及依赖关系，但此时并未涉及总线访问延迟的影响. 对于基本块中的每条指令及每个流水线阶段，第 19～25 行和第 27～33 行分别处理 first 取不同值的情况，第 20 和 28 行分别调用函数 $Com\_bdelay()$ 计算总线访问延迟. 在第 22、24、30 和 32 行分别更新相应流水线阶段的完成时间. 使用 Chronos 的原有处理依赖关系的方法，第 36 行更新指令 i 在后续流水线阶段上的依赖关系，第 38 行更新指令 i 的后续指令的依赖关系. 第 40 行计算该基本块的最差情况下执行时间.

算法 3.　多核环境下某任务基本块最差情况下的执行时间分析.

输入：基本块 blk 的执行图，L_{round}，对应的总线时槽 s'_j

输出：基本块 blk 最差情况下执行时间 $MET[first][blk][\ ]$

```
1.  FUNCTION Com_bdelay(start, offset, pre);
2.   IF (start<= pre) THEN
3.    busdelay=L_round*L_B;
4.   ELSE
5.    cur=start+offset;
6.    busdelay=(s'_j-1)*L_B-cur mod(L_round*L_B);
7.    busdelay=(L_round*L_B+busdelay)mod(L_round*L_B);
8.   END IF
9.   RETURN busdelay;
10. END FUNCTION
//以下为主过程
11. FOR (first=0; first<=1; first++) DO
12.  FOR (offset=0;offset<(L_round*L_B);offset++)
     DO
13.   pipeline_analysis();
14.   pre=0;
15.   FOR (依次取 blk 中的每条指令 i) DO
16.    FOR(stage=0;stage<pipe_stages;stage++)
       DO
17.     IF (i 在 stage 阶段访问存储系统) THEN
18.      IF (first==0) THEN
19.       IF (AH 指令) THEN
20.        delay=Com_bdelay(T_stage(i)(sta),offset,
                 pre);
21.        pre=T_stage(i)(sta)+delay;
22.        T_stage(i)(fin)=T_stage(i)(fin)+delay+L_lat;
23.       ELSE
24.        T_stage(i)(fin)=T_stage(i)(fin)+L_L2penal;
25.       END IF
26.      ELSE
27.       IF (AH 指令或 PS 指令) THEN
28.        delay=Com_bdelay(T_stage(i)(sta),offset,
                 pre);
29.        pre=T_stage(i)(sta)+delay;
30.        T_stage(i)(fin)=T_stage(i)(fin)+delay+L_lat;
31.       ELSE
32.        T_stage(i)(fin)=T_stage(i)(fin)+L_L2penal;
33.       END IF
34.      END IF IF(first==0) THEN
35.     END IF
36.     更新后续流水线阶段上的依赖关系;
37.    END FOR
```

38. 更新后续指令的依赖关系；
39. END FOR
40. $MET[first][blk][offset]=T_n(fin)-T_1(ready)$；
41. END FOR
42. END FOR
43. RETURN $MET[first][blk][\ ]$；

利用算法3可以估算出每个基本块在不同开始时间下的最差情况下执行时间(同时考虑到基本块是否在循环中且是否有PS指令)，共有$(2\cdot L_{\text{round}}\cdot L_B)$个值.根据基本块在控制流图中的执行次序和开始时间(设第一个执行的基本块的开始时间为0)，从每个基本块的$(2\cdot L_{\text{round}}\cdot L_B)$个值中选择一个用来估算硬实时任务的$WCET'_{c_i}$.

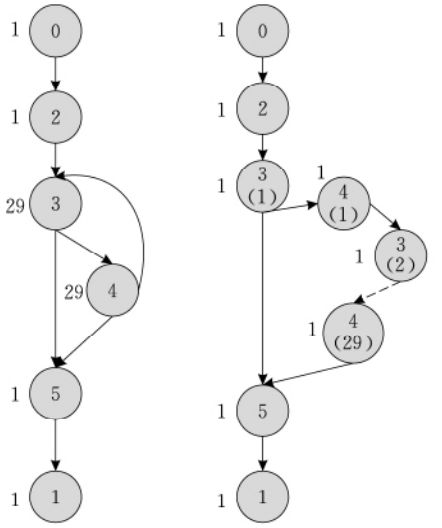
对于一个基本块blk，设$T_{blk}(sta)$、$offset_{blk}$和$T_{blk}(fin)$分别为该基本块的开始时间、总线偏移量和完成时间.开始时间$T_{blk}(sta)$是其直接前驱的最迟完成时间，总线偏移量$offset_{blk}$可以表示为$offset_{blk}=T_{blk}(sta)\bmod(L_{\text{round}}\cdot L_B)$，设$pre$为基本块$blk$的直接前驱，$offset_{pre}$为基本块$pre$的总线偏移量.$offset_{blk}$可以表示为式(15)，完成时间$T_{blk}(fin)$可以表示为式(16).

$$offset_{blk}=(offset_{pre}+MET[first][pre][offset_{pre}])\cdot \bmod(L_{\text{round}}\cdot L_B) \tag{15}$$

$$T_{blk}(fin)=T_{blk}(sta)+MET[first][blk][offset_{blk}] \tag{16}$$

若控制流图存在循环结构，则全部展开，对于事先不能确定循环次数的循环结构，按照Chronos处理方法将循环次数的最大上限作为循环次数展开.此时，整个控制流图仅存在分支和顺序结构，且一个基本块的开始时间等于其直接前驱的最大完成时间.反复使用式(15)和(16)可以计算最后一个基本块的完成时间.

以Mälardalen WCET benchmark[26]测试程序集中的fibcall测试程序为例，说明利用算法3的结果估算WCET的方法.图4(a)是fibcall的控制流图，其中，圆形代表基本块，圆形内的数字为基本块的编号，旁边的数字为该基本块的执行次数，有向边代表基本块的先后次序.将循环展开后的控制流图如图4(b)所示，若基本块在循环中，标识出该基本块是第几次执行.设fibcall在该例中对应的总线时槽为s'_i，开始执行时间为T_{st}.则基本块0的开始时间$T_0(sta)=T_{st}$，对应的总线偏移量$offset_0=((s'_i-1)\cdot L_B+T_0(sta))\cdot \bmod(L_{\text{round}}\cdot L_B)$，利用式(16)可以得到基本块0的完成时间$T_0(fin)=T_0(sta)+MET[0][0][offset_0]$.



(a) 控制流图 (b) 循环展开的控制流图

图4 fibcall的控制流图及循环展开

基本块2的开始时间$T_2(sta)=T_0(fin)$，根据式(15)和(16)，可得基本块2的总线偏移量和完成时间，即$offset_2=(offset_0+MET[0][0][offset_0])\cdot \bmod(L_{\text{round}}\cdot L_B)$，$T_2(fin)=T_2(sta)+MET[0][2][offset_2]$.接下来，可以获得基本块3第1次执行的开始时间(记为$T_{3(1)}(sta)$)、总线偏移量(记为$offset_{3(1)}$)和完成时间(记为$T_{3(1)}(fin)$).处理完基本块3的第1次执行，进入分支结构的处理，反复利用式(15)和(16)可以得到基本块4第29次执行对应的总线偏移量$offset_{4(29)}$和完成时间$T_{4(29)}(fin)$，$T_{4(29)}(fin)=T_{4(29)}(sta)+MET[1][4][offset_{4(29)}]$.基本块5的开始时间$T_5(sta)=\max(T_{3(1)}(fin),T_{4(29)}(fin))$，总线偏移量$offset_5=(offset_{3(1)}+T_5(sta)-T_{3(1)}(sta))\cdot \bmod(L_{\text{round}}\cdot L_B)$，最终可以得到基本块1的完成时间$T_1(fin)=T_1(sta)+MET[0][1][offset_1]$，即该例的$WCET'_{c_i}=T_1(fin)$.

算法4是基于算法3的多核环境下硬实时任务最差情况下执行时间的分析算法.第1～21行定义了处理分支结构的函数$Com_branch()$，输入参数Pn指向分支结构的开始基本块，Nbr_Pn是分支结构的分支数，$offset$是分支结构开始时对应的总线偏移量，$first$标识该基本块是否是第一次执行，blk是基本块编号，$br_exe[i]$是分支i的总执行时间(相对于分支结构的开始时间)，第7行根据式(16)计算当前基本块的完成时间，第8、9行根据式(15)计算总线偏移量，第10～15行处理嵌套分支结构，第10行判断是否有分支结构，若有，则在第11行读取分支的数目Nbr_Pb，在第12行调用$Com_branch()$处理分支结构.第19行获得分支结构的最大完成时间

(相对于分支结构的开始时间). 第 23 行建立链表 $TSLinkList$,若存在分支结构,则在链表中用分支结构的开始基本块对应的链表结点指向分支结构的结束基本块对应的链表结点,分支结构的分支存放在分支结构的开始基本块对应的链表结点中. 第 28、33 行分别计算当前基本块的完成时间(相对于任务开始时间). 第 38 行根据式(14)计算任务的最差情况下执行时间.

算法 4.　多核环境下硬实时任务最差情况下执行时间的分析.

输入:循环已展开的控制流图,每个基本块的执行图,L_{round},对应的总线时槽 s'_j,利用算法 1 计算的 bank 冲突延迟 D_{bank}

输出:硬实时任务的最差情况下执行时间 MT_exe

```
1.  FUNCTION Com_branch(Pn, Nbr_Pn, offset) DO
2.    FOR (i=1; i<=Nbr_Pn; i++) DO
3.      boffset=offset, br_exe[i]=0;
4.      Pb 指向 Pn 的第 i 个分支的第一个基本块;
5.      WHILE(Pb 不为空) DO
6.        获得由 Pb 指向的基本块标识 blk 和执行信息 first;
7.        br_exe[i]=br_exe[i]+MET[first][blk]
                              [boffset];
8.        boffset=boffset+MET[first][blk][boffset];
9.        boffset=boffset mod (L_round * L_B);
10.       IF (Pb 有分支) THEN
11.         获得 Pb 指向基本块的分支数 Nbr_Pb;
12.         tmp_exe=Com_branch(Pb, Nbr_Pb, boffset);
13.         br_exe[i]=br_exe[i]+tmp_exe;
14.         boffset=(boffset+tmp_exe) mod(L_round * L_B);
15.       END IF
16.       Pb=Pb->next;
17.     END WHILE
18.   END FOR
19.   Mb_exe=max(br_exe[i] | 1<=i<=Nbr_Pn);
20.   RETURN Mb_exe;
21. END FUNCTION
//以下为主过程
22. 调用算法 3 估算每个基本块的最差情况下执行时间,存放在 MET[ ][ ][ ]中;
23. 读取控制流图,建立链表 TSLinkList(有头结点);
24. offset=(s'_j - 1) * L_B, MT_exe=0;
25. Pn=TSLinkList->next;
26. WHILE(Pn 不为空) DO
27.   获得由 Pn 指向的基本块标识 blk 和执行信息 first;
28.   MT_exe=MT_exe+MET[first][blk][offset];
29.   offset=(offset+MET[first][blk][offset]) ·
                mod(L_round * L_B);
30.   IF (Pn 有分支) THEN
31.     获得 Pn 指向基本块的分支数 Nbr_Pn;
32.     tmp_exe=Com_branch(Pn, Nbr_Pn, offset);
33.     MT_exe=MT_exe+tmp_exe;
34.     offset=(offset+tmp_exe) mod(L_round * L_B);
35.   END IF
36.   Pn=Pn->next;
37. END WHILE
38. MT_exe=MT_exe+D_bank;
39. RETURN MT_exe;
```

7　实验验证

使用 Mälardalen WCET benchmark[26] 测试程序集分别设计无 bank 访问冲突、存在 bank 访问冲突的两个实验场景,来验证前面提出的算法的正确性.

7.1　无 bank 访问冲突的应用场景

7.1.1　实验环境和测试程序

由 6 个同构核 $\{c_1, c_2, \cdots, c_6\}$ 组成的多核系统中,每个核有一个有序(in-order) 5 级流水线,无分支预测功能,取指队列大小为 4,取指宽度为 2,指令窗大小为 8. 每个核有私自 L1 数据和 L1 指令缓存,大小均为 64 字节,1 个 bank,2 路关联,每 line 有 8 字节,1 个时钟周期的访问时间,采用 LRU 替换策略. L2 缓存为所有核共享,大小为 4 KB,被均匀划分成 4 个 bank,每个 bank 的大小为 1 KB,4 路关联,每 line 有 32 Bytes,4 个时钟周期访问时间(即 $L_M=4$),采用 LRU 替换策略. 每个 bank 又被均匀划分成 8 个 column. 每个 column 的大小为 128 Bytes(即 1 组 4 路关联的 line). 连接 L2 缓存和核的总线为 TDMA 实时总线,采用简单轮询总线调度策略,总线完成一次请求所需要的时间为 2 个时钟周期,即 $L_B=2$. 请求访问主存需要的时间为 $L_{\text{L2penal}}=30$ 个时钟周期.

使用的测试程序是 Mälardalen WCET benchmark 测试程序集中的一部分,测试程序的特性如表 2 所示. 为了给测试程序分配合适的 L2 缓存大小,我们使用 Chronos 测量这些测试程序在分配给

表 2　使用的测试程序特性

测试程序	字节数	代码行数
bsort100	2779	128
cnt	2880	267
fibcall	3499	72
expint	4288	157
insertsort	3892	92
prime	797	47

不同 L2 缓存大小时的 WCET，测量结果和采用的 L2 缓存大小如表 3 所示.

表 3 不同 L2 缓存大小时测量的 WCET(时钟周期)和采用值(字节数/column 数)

缓存大小	bsort100	cnt	expint	fibcall	insertsort	prime
128 B	6 888 050	35 752	16 921	970	13 635	75 000
256 B	6 888 330	50 633	16 873	970	11 499	75 000
512 B	10 027 500	39 040	17 191	970	11 451	75 000
1024 B	9 263 600	29 152	17 191	970	11 451	75 000
2048 B	2 965 080	29 152	17 191	970	11 451	75 000
4096 B	2 965 080	29 152	17 191	970	11 451	75 000
采用值	2048/16	1024/8	256/2	128/1	512/4	128/1

7.1.2 实验结果

实验中采用的任务到核映射和核到总线时槽映射如表 4 所示. 用 Chronos 得到各任务访问 L2 缓存的总线请求时间序列如图 5 所示. 使用算法 2 做核到 bank 映射，若可能存在 bank 访问冲突，则用算法 1 计算 bank 冲突延迟，结果如图 6 所示，解空间为 720，总的 bank 冲突延迟的范围为[0，149 560]，其中一个不存在 bank 访问冲突的映射关系如表 5 所示. 在表 5 所示的核到 bank 映射关系中，映射到任意一个 bank 上的任务之间的模距离都大于等于 2. 由于 $L_M/L_B=2$，因此，在该映射中不存在 bank 访问冲突.

表 4 任务到核映射和核到总线时槽映射

测试程序	核	总线时槽
insertsort	c_1	s_1
expint	c_2	s_2
bsort100	c_3	s_3
fibcall	c_4	s_4
prime	c_5	s_5
cnt	c_6	s_6

表 5 一个没有 bank 冲突延迟时核到 bank 映射关系

测试程序	b_1	b_2	b_3	b_4
insertsort	4	0	0	0
ixpint	0	0	0	2
bsort100	3	8	5	0
fibcall	0	0	0	1
prime	1	0	0	0
cnt	0	0	3	5

	单位: 时钟周期	总访问次数
insertsort:	31 93 124 155 217 248 279 341 …	1163
expint:	31 93 124 160 161 162 163 166 …	2013
bsort100:	31 93 124 1952 1954 1957 1959 …	394 210
fibcall:	62 93 129 364 366 367	6
prime:	62 93 124 186 217 248 281 310 …	9335
cnt:	31 98 102 105 106 112 113 115 …	3477

图 5 各任务的总线请求时间序列

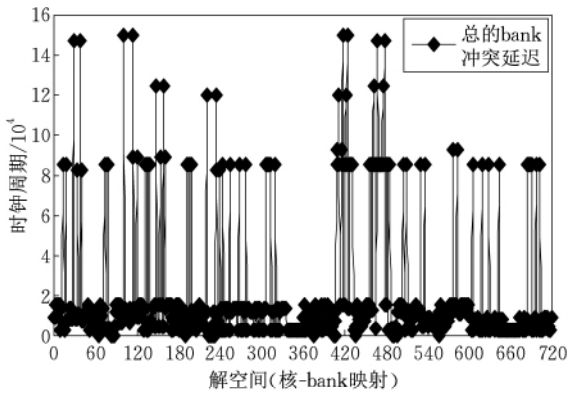


图 6 表 4 中 6 个任务时算法 2 的执行结果

为了考察优化核到 bank 映射后对 WCET 的影响，取表 6 所示的核到 bank 映射作为未优化时的映射，在该映射下各任务遭受的 bank 冲突延迟如表 7 所示. 使用算法 4(调用算法 3)估算了在两个映射下各任务的 WCET，结果如图 7 所示，所有测量结果都是相对于任务在单核系统中测量的结果(下同). 在图 7 中，*Opt* 表示优化核到 bank 映射后估算的结果，*no_Opt* 表示未优化核到 bank 映射时估算的结果. 从图 7 中可以看出，相对于未优化时的估算结果，优化映射后对所有任务的 WCET 有不同程度的改善，平均提高了约 15%. 对 expint 的 WCET 改善程度最大，提高了大约 50%. 虽然 bsort100 遭受的 bank 冲突延迟为 43 736 个时钟周期(如表 7 所示)，但由于其规模较大，改善效果相对不明显(提高了约 1%). 另外，在这两个映射中，insertsort 和 fibcall 都没遭受到 bank 访问延迟，估算的 WCET 未发生变化.

表 6 一个未优化时的核到 bank 映射关系

测试程序	b_1	b_2	b_3	b_4
insertsort	4	0	0	0
expint	0	0	2	0
bsort100	4	8	4	0
fibcall	0	0	0	1
prime	0	0	1	0
cnt	0	0	1	7

表 7 在表 6 所示的映射中各任务遭受的 bank 冲突延迟

测试程序	bank 冲突延迟/时钟周期
insertsort	0
expint	24 070
bsort100	43 736
fibcall	0
prime	43 684
cnt	35 738

AI+ILP(抽象解释加整数线性规划)方法是现有文献估算 WCET 的常用方法，如 Chattopadhyay 等人[13]、Kelter 等人[14-15]等. AI+ILP 方法的共同

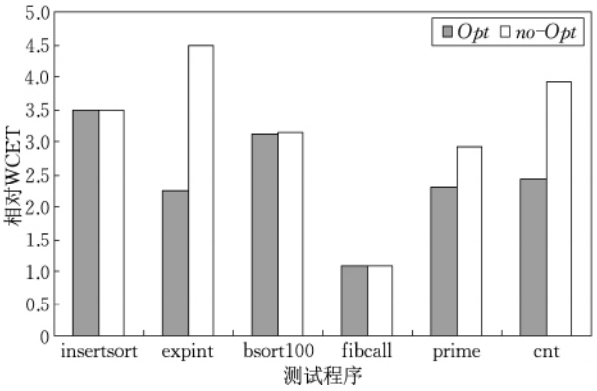


图 7　优化核到 bank 映射后和未优化时的估算结果对比

点是分别估算每个基本块在流水线上的执行时间、访问共享 L2 缓存的时间、访问主存的时间和总线访问延迟，然后用线性规划求解工具将这些时间组合起来得到任务的 WCET，这里，我们将这类方法统称为“AI+ILP”方法.

分别使用算法 4 和 AI+ILP 方法估算了在表 5 所示的映射中各任务的 WCET 值（没有 bank 冲突），结果如图 8 所示. 其中，Alg4 代表使用算法 4 对总线访问延迟进行了消重的估算结果，AI+ILP 代表使用 AI+ILP 未对总线访问延迟进行消重的估算结果. 相对于 AI+ILP 方法而言，算法 4 对任务的 WCET 有不同程度的改善，平均提高了约 30%. 影响估算结果的主要因素有访问 L2 缓存的次数、访问密集度和指令在流水线上的依赖关系等. insertsort 的多数访存指令都集中在基本块 3 中（使用 Chronos 获得），需要进行消重的计算较多，改善程度最大，提高了大约 50%. bsort100 访问 L2 缓存的次数为 394210 次（如图 5 所示），消重效果也比较明显（约 45%）. 由于 fibcall 的访问次数很少，因此改善效果不明显. 虽然 prime 的访问次数也较大（9335 次），但受到其指令间依赖关系的影响，抵消了对总线延迟消重后的效果.

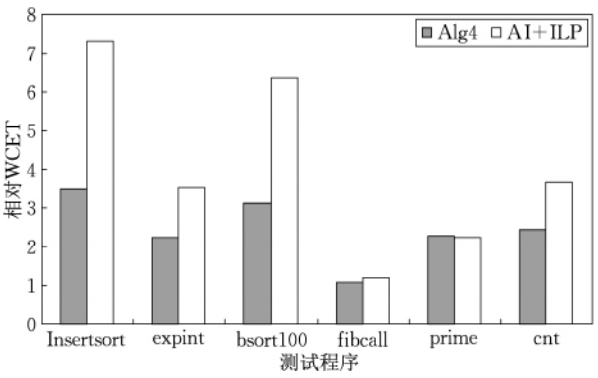


图 8　算法 4 和 AI+ILP 方法估算的结果对比

分别使用两种方法对每个任务进行 20 次估算，20 次运行时间的平均值如表 8 所示. 由此可以看出，由于算法 4 需要调用算法 3 并参与时间消重计算，因此其运行时间比 AI+ILP 方法有所提高.

表 8　两种估算方法的运行时间/s

测试程序	Alg4	AI+ILP
insertsort	0.480 684 40	0.463 274 40
expint	1.057 321 40	0.968 170 20
bsort100	0.353 742 45	0.343 767 40
fibcall	0.174 488 90	0.178 401 10
prime	0.609 260 60	0.596 688 75
cnt	0.810 853 00	0.787 557 20

7.2　存在 bank 访问冲突的应用场景

为了验证 bank 访问冲突不能消除时的优化效果，我们设计了相应的应用场景. 在实验环境中，L2 缓存的容量大小为 3 KB，被均匀划分成 3 个 bank，每个 bank 的大小为 1 KB. 每个 bank 又被均匀划分成 8 个 column，每 column 的大小为 128 Bytes. 其他参数的设置值采用 7.1.1 节中实验环境的相应参数值. 使用了表 2 中的 5 个测试程序，如表 9 所示.

表 9　测试程序与采用的 L2 缓存大小

测试程序	缓存大小/columns
insertsort	4
expint	2
bsort100	16
fibcall	1
prime	1

7.2.1　实验结果

在实验中采用的任务到核映射和核到总线时槽映射如表 10 所示. 在该应用场景中，核 c_6 空闲，不占用总线时间，即每个总线调度周期的长度为 $5 \cdot L_B = 10$ 个时钟周期. 各任务访问 L2 缓存的总线请求时间序列如图 5 所示. 执行算法 2 做核到 bank 映射，结果如图 9 所示，解空间为 120，总的 bank 冲突延迟范围为[33 704，110 660]. 具有最小 bank 冲突延迟的核到 bank 的映射如表 11 所示，在该映射中，只在 bank b_1 上存在 bank 访问冲突，各个任务遭受的 bank 冲突延迟如表 12 所示.

表 10　使用的任务到核的映射和核到总线时槽的映射

测试程序	核	总线时槽
insertsort	c_1	s_1
expint	c_2	s_2
bsort100	c_3	s_3
fibcall	c_4	s_4
prime	c_5	s_5

表 11 一个 bank 冲突延迟最小的核到 bank 映射

测试程序	b_1	b_2	b_3
insertsort	4	0	0
expint	2	0	0
bsort100	0	8	8
fibcall	1	0	0
prime	1	0	0

表 12 在表 11 所示的映射中各任务遭受的 bank 冲突延迟

测试程序	bank 冲突延迟/时钟周期
insertsort	9108
expint	12344
bsort100	0
fibcall	6
prime	12246

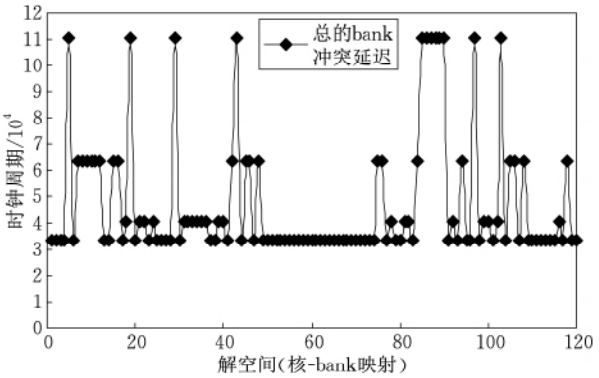


图 9 有 5 个任务时算法 2 的执行结果

取表 13 所示的核到 bank 映射作为未优化时的映射，在该映射下各任务遭受的 bank 冲突延迟如表 14 所示. 使用算法 4 估算在两个映射下各任务的 WCET，结果如图 10 所示. 从图 10 中可看出，相对于未优化时的估算结果，优化映射后对所有任务的 WCET 有不同程度的改善，平均提高了约 10%. 对 insertsort 的 WCET 改善程度最大，提高了大约 20%. 虽然 bsort100 遭受的 bank 冲突延迟为 37840 个时钟周期(如表 14 所示)，但相对效果不明显(约 1%).

表 13 一个没有优化的核到 bank 映射

测试程序	b_1	b_2	b_3
insertsort	4	0	0
expint	2	0	0
bsort100	1	8	7
fibcall	0	0	1
prime	1	0	0

表 14 在表 13 所示的映射中各任务遭受的 bank 冲突延迟

测试程序	bank 冲突延迟/时钟周期
insertsort	14428
expint	20604
bsort100	37840
fibcall	8
prime	37780

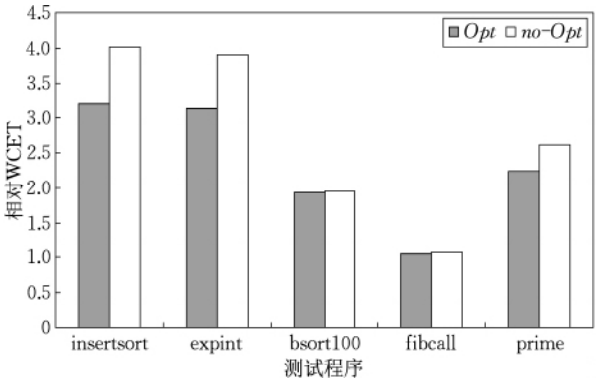


图 10 优化核到 bank 映射后和未优化时的估算结果对比

分别使用算法 4 和 AI+ILP 方法估算在表 11 所示的映射中各任务的 WCET(各任务遭受的 bank 冲突延迟如表 12 所示)，结果如图 11 所示. 相对于 AI+ILP 方法而言，算法 4 对任务的 WCET 估算结果有不同程度的改善，平均提高了约 25%，例如对 bsort100 WCET 的改善程度约为 45%，对 fibcall WCET 的改善程度约为 5%，对 prime WCET 的改善程度约为 8%. 另外，在该场景下，分别使用两种方法对每个任务进行 20 次估算，20 次运行时间的平均值如表 15 所示，由此可以看出，由于算法 4 需要调用算法 3 并参与时间消重计算，因此其运行时间比 AI+ILP 方法有所提高.

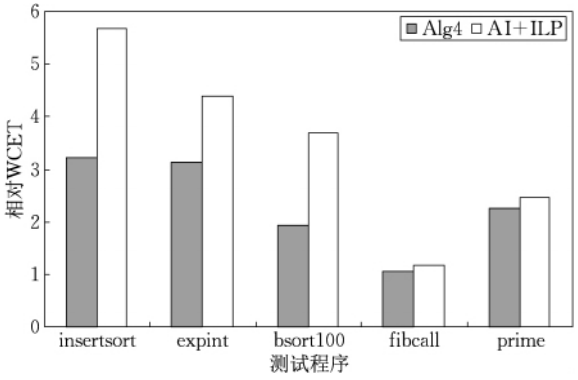


图 11 算法 4 和 AI+ILP 方法估算的结果对比

表 15 两种估算方法的运行时间/s

测试程序	Alg4	AI+ILP
insertsort	0.472859	0.458355 35
expint	1.142646 7	0.970681 90
bsort100	0.377174 5	0.341838 55
fibcall	0.176137 3	0.173120 95
prime	0.652208 7	0.597015 15

8 结 论

本文提出了通过优化核到 bank 映射来最小化

硬实时多核系统的 bank 冲突延迟方法，旨在通过消除 bank 访问冲突或最小化 bank 冲突延迟来改善多核系统中硬实时任务的 WCET.

通过对硬实时多核系统中 bank 冲突延迟的分析，我们得出了硬实时任务间不存在 bank 访问冲突的判断条件，并用优化核到 bank 映射的方法来消除 bank 访问冲突. 然而，并不是所有的 bank 访问冲突都可以消除，此时需要优化核到 bank 映射来最小化 bank 冲突延迟. 为此，我们设计了求解该优化问题的相应算法. 另外，还设计了能够对总线访问延迟进行消重的 WCET 估算方法.

实验结果表明，本文提出的优化方法可以消除硬实时多核系统中的 bank 访问冲突或使 bank 冲突延迟最小化. 与现有 WCET 估算方法比较，本文提出的 WCET 估算方法可以获得更精确的 WCET 值.

参 考 文 献

[1] Thiele L, Wilhelm R. Design for timing predictability. Real-Time Systems, 2004, 28(2-3): 157-177

[2] Wilhelm R, Mitra T, Mueller F, Puaut I, et al. The worst-case execution-time problem: Overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 2008, 7(3): 36: 1-36: 53

[3] Cullmann C, Ferdinand C, Gebhard G, Grund D, et al. Predictability considerations in the design of multi-core embedded systems//Proceedings of the Embedded Real Time Software and Systems. San Diego, USA, 2010: 36-42

[4] Zhang Wei, Yan Jun. Static timing analysis of shared caches for multicore processors. Journal of Computing Science and Engineering, 2012, 6(4): 267-278

[5] Guan Nan, Stigge M, Yi Wang, Yu Ge. Cache-aware scheduling and analysis for multicores//Proceedings of the 7th ACM International Conference on Embedded Software. Grenoble, France, 2009: 245-254

[6] Ho R, Mai K W, Horowitz M A. The future of wires. Proceedings of the IEEE, 2001, 89(4): 490-504

[7] Sylvester D, Keutzer K. Getting to the bottom of deep submicron II: A global wiring paradigm//Proceedings of the 1999 International Symposium on Physical Design. Monterey, USA, 1999: 193-200

[8] Kaseridis D, Stuecheli J, John L K. Bank-aware dynamic cache partitioning for multicore architectures//Proceedings of the International Conference on Parallel Processing. Vienna, Austra, 2009: 18-25

[9] Paolieri M, Quiñones E, Cazorla F J, et al. Hardware support for WCET analysis of hard real-time multicore systems//Proceedings of the 36th IEEE/ACM International Symposium on Computer Architecture. Austin, Texas, USA, 2009: 57-68

[10] Yoon Man-Ki, Kim Jung-Eun, Sha Liu. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems//Proceedings of the 32nd IEEE Real-Time Systems Symposium. Vienna, Austra, 2011: 227-238

[11] Andrei A, Eles P, Peng Z, Rosen J. Predictable implementation of real-time applications on multiprocessor systems-on-chip//Proceedings of the 21st International Conference on VLSID. Hyderabad, India, 2008: 103-110

[12] Rosén J, Andrei A, Eles P, Peng Z. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip//Proceedings of the 28th IEEE Real-Time Systems Symposium. Tucson, Arizona, USA, 2007: 49-60

[13] Chattopadhyay S, Roychoudhury A, Mitra T. Modeling shared cache and bus in multi-cores for timing analysis//Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems. St. Goar, Germany, 2010: 1-10

[14] Kelter T, Falk H, Marwedel P, et al. Bus-aware multicore WCET analysis through TDMA offset bounds//Proceedings of the 2011 Euromicro Conference on Real-Time Systems. Porto, Portugal, 2011: 3-12

[15] Kelter T, Falk H, Marwedel P, et al. Static analysis of multi-core TDMA resource arbitration delays. Real-Time Systems, 2014, 50(2): 185-229

[16] Chattopadhyay S, Chong L K, Roychoudhury A, et al. A unified WCET analysis framework for multicore platforms. ACM Transactions on Embedded Computing Systems, 2014, 13(4s): 124:1-124:29

[17] Li Yan, Suhendra V, Liang Yun, et al. Timing analysis of concurrent programs running on shared cache multi-cores//Proceedings of the 30th IEEE Real-Time Systems Symposium. Washington, USA, 2009: 57-67

[18] Yan Jun, Zhang Wei. WCET analysis for multi-core processors with shared L2 instruction caches//Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium. St. Louis, Mo, USA, 2008: 80-89

[19] Chen Fang-Yuan, Zhang Dong-Song, Wang Zhi-Ying. Static analysis of run-time inter-thread interferences in shared cache multi-core architectures based on instruction fetching timing//Proceedings of the IEEE International Conference on Computer Science and Automation Engineering. Shanghai, China, 2011: 208-212

[20] Ding Hu-Ping, Liang Yun, Mitra T. WCET-centric dynamic instruction cache locking//Proceedings of the Conference on Design, Automation & Test in Europe. Dresden, Germany, 2014: 1-6

[21] Liu Tian-Tian, Li Min-Ming, Xue C J. Instruction cache locking for multi-task real-time embedded systems. Real-Time Systems, 2012, 48(2): 166-197

[22] Li Xian-Feng, Liang Yun, Mitra T, Roychoudhury A. Chronos: A timing analyzer for embedded software. Science of Computer Programming, 2007, 69(1-3): 56-67

[23] Theiling H, Ferdinand C, Wilhelm R. Fast and precise WCET prediction by separated cache and path analyses. Real-Time Systems, 2000, 18(2-3): 157-179

[24] Burger D, Austin T M. The simplescalar tool set, version 2.0. ACM SIGARCH Computer Architecture News, 1997, 25(3): 13-25

[25] Li Xian-Feng, Roychoudhury P, Mitra P. Modeling out-of-order processors for WCET analysis. Real-Time Systems, 2006, 34(3): 195-227

[26] Gustafsson J, Betts A, Ermedahl A, Lisper B. The Mälardalen WCET benchmarks: Past, present and future//Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis. Brussels, Belgium, 2010: 137-147

附 录.

定理 2. 已知 N_{core}、L_M/L_B 和 N_{column}，且 L2 缓存的容量大小满足需求，若 $\lfloor N_{core}/(L_M/L_B)\rfloor > N_{column}$，则 bank 访问冲突可以通过优化核到 bank 的映射去消除.

证明. 令 $DM=L_M/L_B$ 为整数(在 3.1 节中已作规定)，将核的集合 C 划分成 DM 个部分，$C=\{C_0,C_1,\cdots,C_{(DM-1)}\}$，$C_0\cap C_1\cap\cdots\cap C_{(DM-1)}=\varnothing$，且满足 $\forall C_i\subset C$，$\forall c_j\in C_i$，$j \bmod DM=i$. 可得 $d_{jq}\geq DM$，$\forall c_j,c_q\in C_i$，$0\leq i\leq(DM-1)$. 若 $\forall c_j,c_q\in C_{b_k}$ 且 $c_j,c_q\in C_i$，$0\leq i\leq(DM-1)$，则根据定理 1 可知：在 bank b_k 上不存在 bank 访问冲突.

先构造一个映射，将 $C_i(0\leq i\leq(DM-1))$ 中的核按序号大小排序，并按下述方法做核到 bank 的映射：(1) 将 C_0 中的核依次向 bank 做映射，先向 b_1 做映射，b_1 分配完后，向 b_2 做映射，依次类推. 设 C_0 的最后一个核映射到 b_k，且 b_k 有 column 未分配出去，即 $b_1\sim b_{(k-1)}$ 已分配完；(2) 将 C_1 中的核依次向 bank 做映射，先向 b_k 做映射，并按照(1)方法做映射. 依次类推，最后，完成 $C_{(DM-1)}$ 到 bank 的映射. 现只需证明在按上述方法确立的核到 bank 的映射中不存在 bank 访问冲突即可.

用 $M:(C\rightarrow B)$ 表示按上述方法确定的核到 bank 的映射. 对于映射 $M:(C\rightarrow B)$，$\forall C_i\subset C$，$0\leq i<(DM-1)$，现考察 $C_{(i+1)}$ 中的核与 C_i 中的核共享 bank 的情况. 用 $\{\{c_{iq},c_{i(q+1)},\cdots,c_{i(n_i)}\},\{c_{(i+1)1},c_{(i+1)2},\cdots,c_{(i+1)p}\}\}$ 表示 C_i 和 $C_{(i+1)}$ 中共享一个 bank 的核集合(用 $\hat{A}_{i(i+1)}$ 表示)，其中 $\{c_{iq},c_{i(q+1)},\cdots,c_{i(n_i)}\}$ 是 C_i 的子集，n_i 为 C_i 中的核数，$\{c_{(i+1)1},c_{(i+1)2},\cdots,c_{(i+1)p}\}$ 是 $C_{(i+1)}$ 的子集. 现在需要证明如下两点即可：(1) $q\geq(p+2)$；(2) c_{iq} 和 $c_{(i+1)p}$ 之间的模距离大于等于 L_M/L_B.

证明 $q\geq(p+2)$. 假设 $q\leq(p+1)$，$\hat{A}_{i(i+1)}$ 可以表示为 $\{c_{(i+1)1},c_{(i+1)2},\cdots,c_{(i+1)p},c_{iq},c_{i(q+1)},\cdots,c_{i(n_i)}\}$，则 $\hat{A}_{i(i+1)}$ 中的核数(用 $N_{i(i+1)}$ 表示)大于等于 $\lfloor N_{core}/(L_M/L_B)\rfloor$. 由于 $\lfloor N_{core}/(L_M/L_B)\rfloor > N_{column}$，所以 $N_{i(i+1)}>N_{column}$，又因为是以 column 为单位向核分配 L2 缓存(消除 storage 干扰)，即每个核至少需要 1 个 column(核是空闲的除外)，核集合 $\hat{A}_{i(i+1)}$ 需要的总 column 数大于等于 $N_{i(i+1)}$，所以 $\hat{A}_{i(i+1)}$ 需要的总 column 数大于 N_{column}，即核集合 $\hat{A}_{i(i+1)}$ 需要的总 column 数不可能由一个 bank 容纳，与 $\hat{A}_{i(i+1)}$ 中的核共享一个 bank 矛盾，故 $q\leq(p+1)$ 不成立，因为 q、p 为整数，$q\geq(p+2)$.

证明 c_{iq} 和 $c_{(i+1)p}$ 之间的模距离大于等于 L_M/L_B. 在集合 C 中，c_{iq} 的序号可以表示为 $(q-1)\cdot DM+i+1$，$c_{(i+1)p}$ 的序号可以表示为 $(p-1)\cdot DM+(i+1)+1$. c_{iq} 和 $c_{(i+1)p}$ 之间的模距离为 $((q-1)\cdot DM+i+1)-((p-1)\cdot DM+(i+1)+1)$，可简化为 $(q-p)\cdot DM-1$，因为 $q\geq(p+2)$，则 $((q-p)\cdot DM-1)\geq(2\cdot DM-1)\geq DM$.

综上所述，在 $\hat{A}_{i(i+1)}$ 中，核间的模距离都大于等于 L_M/L_B，根据定理 1，在这些核共享使用的 bank 上不存在 bank 访问冲突，且 $C_i(0\leq i<(DM-1))$ 具有一般性，因此，在映射 $M:(C\rightarrow B)$ 中不存在 bank 访问冲突. 证毕.

定理 3. 已知 C、$N_{core} \bmod 2=0$、$L_M/L_B=2$、N_{column} 和 $Size_{c_i}>0$，$1\leq i\leq N_{core}$. 将 C 分割成两个互不相交的子集 C_0 和 C_1，且满足：在任一子集中的任意两个核之间的模距离大于等于 L_M/L_B. 若在每个子集中能够找到一个核集合 $C_{si}(\subseteq C_i)$，$0\leq i\leq 1$，且满足 $Size(C_{si})\geq Size(C_i) \bmod N_{column}$，则 bank 访问冲突可以通过优化核到 bank 的映射去消除.

证明. 将 C_0 和 C_1 分别整理为 $C_0=\{c_{01},c_{02},\cdots,C_{s0}\}$ 和 $C_1=\{C_{s1},c_{1i},c_{1(i+1)},\cdots,c_{1(n_1)}\}$，并按照定理 2 的证明方法构造核到 bank 的映射 $M:(C\rightarrow B)$. 由于 $Size(C_{si})\geq Size(C_i) \bmod N_{column}$，则 $\hat{A}_{01}\subseteq\{C_{s0},C_{s1}\}$ 且 $\forall c_j,c_q\in C_{si}$，$d_{jq}\geq L_M/L_B$，$0\leq i\leq 1$，因此在映射 $M:(C\rightarrow B)$ 中不存在 bank 访问冲突. 证毕.

ZHANG Ji-Zan, born in 1973, Ph. D. candidate. His research interests include computer architecture and computer network.

GU Zhi-Min, born in 1964, Ph. D., professor. His research interest is optimization of multicore/many core.

Background

The usage of multicore processor in hard real-time systems brings new challenges to WCET analysis. The hard real-time task, running on hard real-time multicore systems at the same time, can interfere with each other on shared resources. Shared cache is one important shared resource for hard real-time multicore systems, the existing analysis methods of bank access conflict are only confined to bounding the upper-bound of bank conflict delay. However, the methods of bounding the upper-bound are only suitable for their special bus arbitration policy. Moreover, the WCET estimated by bounding the upper-bound suffered by each request has more overestimation.

Now the banked cache architectures becoming the typical design direction, one cache has more than one bank. A core-to-bank mapping partitions the hard real-time tasks running in multicore system into several groups according to whether the tasks share one bank or not. Bank access conflicts can only happen among the tasks sharing one bank.

The goal of this paper is to eliminate or minimize the bank conflict delay in hard real-time multicore system through optimizing core-to-bank mapping. The major contributions of this paper are as follows:

(1) We firstly optimize core-to-bank mapping to eliminate bank access conflict. If not, we optimize core-to-bank mapping to minimize bank conflict delay;

(2) We design an algorithm to find the core-to-bank mapping with the minimum bank conflict delay. In this algorithm, we compute the bank conflict delay according to the timing sequences of the multicore bus requests;

(3) We propose WCET analysis approach for the hard real-time tasks running on multicore system at the same time.

(4) Experimental results demonstrate that our approach of optimizing core-to-bank mapping can minimize bank conflict delay and our approach to estimate WCET is more effective than existing approaches.

This work is supported by the National Natural Science Foundation of China under Grant No. 61370062. This project studies the optimization model on bounding of interferences in on-chip shared resources for energy-efficient and low-delay embedded multi-core. The research results can be widely used in high-end multi-core real-time systems with battery powered. The feasibility of this research has been verified sufficiently in our preliminary work, for examples, "Prefetching in Mobile Embedded System Can be Energy Efficient, IEEE Computer Architecture Letters, Vol. 10, No. 1, 8－11, 2011", "Acceleration of XML Parsing Through Prefetching. IEEE Transactions on Computers, IEEE Xplore Digital Library, online 10. 1109/TC. 2012. 88" and so on.