面向多核处理器的低级并行程序验证

朱允敏[1], 张丽伟[2], 王生原[1], 董 渊[1], 张素琴[1]
(1. 清华大学计算机科学与技术系 北京 100084; 2. 清华大学软件学院 北京 100084)

摘 要: 随着多核处理器的广泛使用以及人们对软件可靠性提出更高要求,多核并行程序验证的重要性日益凸显. 本文提出了一个完整的基于多核的并行程序验证框架,该验证框架包括抽象机器定义、目标代码的形式规范、逻辑推理系统、可靠性定理及其证明. 我们的目标程序使用自旋锁机制来实现线程间对共享内存的互斥访问. 验证框架采用 Hoare 风格的推导方式,使用高阶逻辑来同时描述机器指令的操作语义和所需要的安全策略. 在该框架下,程序员可以对多核并行程序的部分正确性进行验证.

关键词: 程序验证; 多核处理器; 自旋锁; 汇编级代码; 部分正确性
中图分类号: TP301 文献标识码: A 文章编号: 0372-2112 (2009) 4A-001-06

Verifying Parallel Low-Level Programs for Multi-core Processor

ZHU Yun-min[1], ZHANG Li-wei[2], WANG Sheng-yuan[1], DONG Yuan[1], ZHANG Su-qin[1]
(1. *Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China*;
2. *School of Software, Tsinghua University, Beijing* 100084, *China*)

Abstract: As the multi-core processor is widely used and advanced high-trusted software is required, the verification of parallel programs for multi-core processor becomes more and more important. This paper presents a proof framework about the verification of parallel programs, including the definition of our abstract machine, the formal specification for object code, logic inference rules and the proof of soundness theory. The classic spin lock technology is introduced to implement the mutually exclusive access to shared memory. Our proof framework supports Hoare-logic style reasoning. In addition, we use high-order logic to describe both operational semantics and security policy. Programmers can verify the partial correctness of multi-core parallel programs in our framework.

Key words: program verification; multi-core processor; spin lock; assembly level code; partial correctness

1 引言

随着多核处理器的广泛使用以及人们对软件可靠性提出更高要求,多核并行程序的正确性越来越受到重视. 程序验证是保证软件正确性的重要手段,它使用形式化描述的方法给出程序的规范,然后通过一定的验证规则对这些规范以及相关代码进行验证,常用的验证手段之一就是定理证明.

最近十几年程序验证领域有了很大的发展,其中基于类型方法的研究主要为 TAL[1] (Typed Assembly Language), TAL 的做法是把传统的无类型汇编语言扩展成带类型的汇编语言,并在此带类型的汇编语言上研究其安全性质. 基于逻辑方法的研究主要有 PCC[2,3] (Proof-Carrying Code) 框架. PCC 的基本思想是把二进制代码和此份代码的证明放在一起一并交给用户,使得用户能够容易的验证该代码的正确性. 近几年, Yale 大学的 Flint 小组采用语法方式的 FPCC[4] (Foundational PCC), 提出了一种使用 Hoare 风格的汇编代码验证框架,诸如 SCAP[5]、SMC[6] 和 AIM[7] 等项目,这些项目分别实现了基于栈的控制函数、自修改代码和中断程序等内容的验证.

然而,据我们所知,目前还没有任何一个程序验证框架能够在汇编级验证多核上的并行程序. 我们认为借助 FPCC 的优点,在汇编层次上实现多核并行程序的验证,其好处是显而易见的. 首先,这样的框架只需有很少的部分(逻辑系统,证明检查器)必须被信赖就可以验证程序的部分正确性,因此会更加的灵活和实用. 其次,即使程序员在源代码级给出了程序的正确性证明,但是经过复杂的编译优化以后所生成的目标代码的正确性仍难以得到保证,在底层实现证明则可以很好的解决这个问题. 为此,我们在底层为多核并行程序设计了一个完整的验证框架 MCAP (Multi-core-based Certified Assembly Programming), 包括目标机的形式描述、目标程序的形式规范、逻辑推理规则、可靠性定理及其证明等内容.

收稿日期: 2008-10-22; 修回日期: 2008-12-09
基金项目: 国家自然科学基金(No. 60573017)

2 方法概述

在给出验证框架的形式化描述之前, 我们先阐述在抽象机设计中用到的主要方法以及对多核并行程序进行推理时使用的拥有权转移技术.

2.1 抽象机器设计

为了简化模型, 我们设计的机器为双核处理器, 分别编号为核 0 和核 1.

多核多线程编程必须要能够实现内核同步, 我们采用了自旋锁(spin lock) 机制来保护内存的共享空间. 自旋锁是用在多处理器环境中的一种特殊的锁. 如果执行线程发现想要获取的锁空闲, 就会获取锁并继续自己的执行. 相反, 如果发现该锁已被其它的线程占有, 则在周围"自旋", 即反复的执行获取该锁的指令, 直到锁被其它线程释放. 我们使用 spin_ acq *la* 和 spin_ rel *lr* 原语分别表示自旋锁的请求和释放.

为了避免死锁, 当锁被其它线程占有时, spin_ acq *la* 指令将完成类似于一条 yield 指令的操作, 使得在等待队列中的线程有机会运行. 同时, 我们要求一个线程在任何时候最多只占有一个锁, 对于已经占有锁的线程若再次执行 spin_ acq *la* 指令, 指令语义对此将不做定义, 即程序进入滞留状态. 我们的推理推则也将能判断出这样的程序是非合式的.

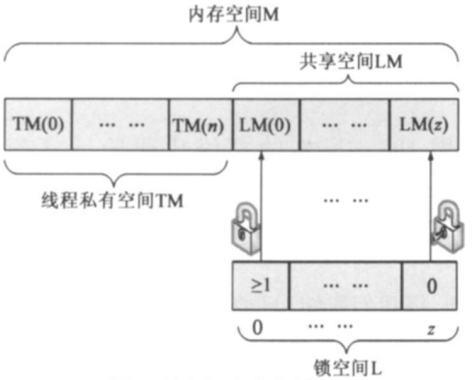

图1 抽象机内存的划分

为了达到更好的并行效果, 在多核编程时我们可以为每个线程分配自己的私有空间. 整个机器的内存 M 的划分如图 1 所示. 图 1 中假定有 $n+1$ 个活动线程, $z+1$ 个锁, *[*TM*(i)]* 为线程 i 的私有空间, *[*LM*(j)]* 表示锁 j 保护的一块共享空间. 在我们的机器中每个自旋锁的数据结构空间占一个内存单位, 并且以该内存的地址作为锁的编号, 所有自旋锁的数据结构空间用 L(以后简称为锁空间) 表示. 需要注意的是将 M 划分后的所有空间之间只要求地址域没有交集, 而无需大小相等, 而且地址空间可以是非连续的.

2.2 内存拥有权的转移

在我们的机器模型中, 只有使用共享内存才能在线程间传递信息. 为了对线程之间的协作进行描述, 在我们的框架中, 程序员可以为每个锁 l 保护的共享空间定义一个不变量 inv_l, 这样就要求线程在获得或失去这块共享空间的拥有权时, 该不变量 inv_l 都必须在这块共享空间上得到满足. 如图 2 所示, 上半部分表示了线程 i 执行 spin_ acq l 指令并成功得锁(此时线程 i 不占有任何锁, 且被请求锁空闲) 的前后状态. 下半部分表示了该线程执行 spin_ rel l 指令释放锁的前后状态. 这四个状态中锁 l 关联的不变量 inv_l 都必需得到满足. 图中也清晰的表示了该线程所拥有的空间(虚线框框起来的部分) 和锁状态(带有钥匙表示空闲, 没有钥匙则表示已被占有) 的变化. 该图中向下箭头处的 others 表示线程 i 在占有锁期间执行的其它非锁操作指令, 这些指令都不会改变它的拥有空间和锁状态, 且 inv_l 在此期间无需保持.

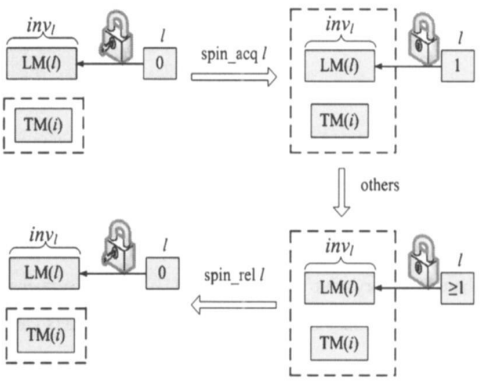

图2 内存拥有权的转移

3 MCAP 验证框架

本节将形式化的给出我们定义的抽象机器模型、推理规则以及目标代码的证明过程.

3.1 元逻辑

我们使用 CIC(Calculus of Inductive Constructions) 来对规范和证明进行编码, 通过 Curry-Howard 同构, 可知 CIC 对应于 Church 的高阶谓词逻辑. 我们使用支持 CIC 的辅助定理证明器 Coq[8] 来实现 MCAP. 为了便于理解, 本文中机器及推理规则的描述使用常见的数学和逻辑符号, 而非严格的 CIC 或 Coq 表示.

3.2 抽象机器定义

一个程序 P 由内存空间 M, 锁空间 L, 两个处理器内核状态 CS_0、CS_1, 两个指令块 I_0 和 I_1(类同于 X86 中 pc 的作用) , 一个等待线程队列 Q 和一个代码堆 C 组成, 见图 3.

锁空间 L 是锁地址 l 到锁状态 n 的映射的集合($n=0$ 表示锁 l 处于空闲状态, $n \geqslant 1$ 则表示锁 l 已被某个线程占有) . 处理器内核状态 CS 记录的信息包括寄存器文件 R、是否已获得锁标记 *loc* 和所占有的锁地址值

la. 等待线程队列 Q 保存所有等待线程的运行上下文,包括内核状态 CS 和指令块 I. 代码堆 C 由若干个指令块 I 组成. 指令块 I 以无条件跳转指令 jd*f* 结尾,类似于编译时代码生成和优化所使用的基本块. 抽象机中含有 16 个通用寄存器(r0-r15)和一个用来保存当前线程 *id* 的特殊寄存器 tn.

(Program)	P	::=	$(M, L, CS_0, I_0, CS_1, I_1, Q, C)$
(Memory)	M	::=	$\{ad \rightarrow w\}^*$
(LockSpace)	L	::=	$\{l \rightarrow n\}^*$
(CoreState)	CS_0, CS_1, CS	::=	(R, *loc*, *la*)
(ThreadQueue)	Q	::=	$\{t \rightarrow (CS, I)\}^*$
(CodeHeap)	C	::=	$\{f \rightarrow I\}^*$
(RegFile)	R	::=	$\{r \rightarrow w\}^*$
(Register)	*r*	::=	r0| r1| …| r15| tn
(LockFlag)	*loc*	::=	true| false
(Label)	*f*	::=	n(nat nums)
(WordVal)	*w*	::=	n(nat nums)
(Address)	*ad*, *l*, *la*, *lr*	::=	n(nat nums)
(ThreadID)	*t*	::=	n(nat nums)
(InstrBlock)	I_0, I_1, I	::=	*c*; I| jd*f*
(Command)	*c*	::=	add *rd*, *rs*, *rt*| sub *rd*, *rs*, *rt*|
	movi *rd*, *w*| ld *rd*, *rs*(*w*)| st *rd*(*w*), *rs*| beq *rs*, *rt*, *f*|		
	bgt *rs*, *rt*, *f*| yield| spin_ acq *la*| spin_ rel *lr*		

图 3　MCAP 抽象机语法

在我们的抽象机中,程序的运行用程序 P 的一步转移来表示,如 $P \rightarrow P'$. 由于我们定义的每条指令都是一个原子操作,因此 $P \rightarrow P'$ 的转移是由任意一个核执行一条指令而产生的. 图 4 以核 0 为例给出了每条指令的操作语义.

为了模拟抢占式线程,我们使用了 yield 原语. yield 原语表示当前线程放弃对所在核的控制权,并将运行上下文保存在 Q 中,然后再从 Q 中随机取出一个线程运行之.

spin_ acq *la* 原语用来请求一个锁地址为*la* 的锁. 如果当前锁标志 *loc* 的值为 false,且请求的锁处于空闲状态(L(*la*) = 0),则该线程将获得锁,并把锁置为被占有状态(L(*la*) $\rightarrow$ 1),然后顺序执行下一条指令. 如果当前锁标志为 false,但是所请求的锁 *la* 处于被占有状态(L(*la*) ⩾1),则完成类似于 yield 指令的操作,唯一的区别是该操作保存的线程上下文中的指令块 I 需要包含 spin_ acq *la* 指令,以便于该线程被重新执行后可以继续请求之前未得到的锁.

spin_ rel *lr* 原语用来释放一个锁地址为*lr* 的锁. 如果当前线程占有锁(*loc* = true),且要释放的锁与当前线程占有的锁是同一个(*lr* = *la*),则将该锁置为空闲状态(L(*la*) $\rightarrow$ 0),然后顺序执行下一条指令.

$(M, L, CS_0, I_0, CS_1, I_1, Q, C) \rightarrow P$ where $CS_0 = (R_0, loc_0, la_0)$	
If I_0 =	Then P=
spin_ acq *la*; I"	(M, L', CS$_0$', I", CS$_1$, I$_1$, Q, C) if(la ∈ dom(L) ∧ loc_0 = false ∧ L(*la*) = 0) L' = L{*la* $\rightarrow$ 1} CS$_0$' = (R_0, true, *la*)
	(M, L', CS$_0$', I$_0$', CS$_1$, I$_1$, Q', C) if(*la* ∈ dom(L) ∧ loc_0 = false ∧ L(*la*) ⩾1) L' = L{*la* $\rightarrow$ L(*la*) + 1} (CS$_0$', I$_0$') = (Q{R_0(tn) $\rightarrow$ (CS$_0$, I$_0$)})(*t*) Q' = (Q{R_0(tn) $\rightarrow$ (CS$_0$, I$_0$)}) \ {*t*}
spin_ rel *lr*; I"	(M, L', CS$_0$', I", CS$_1$, I$_1$, Q, C) if(loc_0 = true ∧ *lr* = la_0) L' = L{la_0 $\rightarrow$ 0} CS$_0$' = (R_0, false, la_0)
yield; I"	(M, L, CS$_0$', I$_0$', CS$_1$, I$_1$, Q', C) (CS$_0$', I$_0$') = (Q{R_0(tn) $\rightarrow$ (CS$_0$, I")})(*t*) Q' = (Q{R_0(tn) $\rightarrow$ (CS$_0$, I")}) \ {*t*}
jd*f*	(M, L, CS$_0$, C(*f*), CS$_1$, I$_1$, Q, C)
beq *rs*, *rt*, *f*; I"	(M, L, CS$_0$, C(*f*), CS$_1$, I$_1$, Q, C) if R(*rs*) = R(*rt*) (M, L, CS$_0$, I", CS$_1$, I$_1$, Q, C) otherwise
bgt *rs*, *rt*, *f*; I"	(M, L, CS$_0$, C(*f*), CS$_1$, I$_1$, Q, C) if R(rs) > R(*rt*) (M, L, CS$_0$, I", CS$_1$, I$_1$, Q, C) otherwise
c; I"	(M', L, (R_0', loc_0, la_0), I", CS$_1$, I$_1$, Q, C) (M', R_0') = Next$_c$(M, R_0)

Where

if *c* =	Next$_c$(M, R) =
add *rd*, *rs*, *rt*	(M, R{*rd* $\rightarrow$ R(*rs*) + R(*rt*)})
sub *rd*, *rs*, *rt*	(M, R{*rd* $\rightarrow$ R(*rs*) − R(*rt*)})
movi *rd*, *w*	(M, R{*rd* $\rightarrow$ *w*})
ld *rd*, *rs*(*w*)	(M, R{*rd* $\rightarrow$ M(R(*rs*) + *w*)}) if R(*rs*) + *w* ∈ dom(M)
st *rd*(*w*), *rs*	(M{R(*rd*) + *w* $\rightarrow$ R(*rs*)}, R) if R(*rd*) + *w* ∈ dom(M)

图 4　MCAP 抽象机操作语义

3.3　目标程序性质的证明框架

本小节将介绍我们设计的目标程序性质证明框架,主要包括程序规范定义、合式公式、推理规则、可靠性定理及其证明. 在抽象机器 MCAP 上对程序进行 Hoare 风格的推理,首先要用断言对程序进行标注,然后使用相应的推理规则进行程序合式性的证明.

3.3.1　程序规范

程序规范是用来描述程序性质的谓词的集合,它直接体现了程序员对程序正确性的具体要求,图 5 给出了程序规范的形式化定义.

LI 是锁到不变量的映射的集合,由于每个锁对应一块共享空间,该空间都有一个描述它的不变量 *inv*,因此每个锁就都对应着一个不变量. LM 描述了每个锁与每块共享空间之间的一一对应关系. TM 描述了每个线程所拥有的私有空间. LTD 描述了每个代码块与所属线程 *id* 的对应关系,用于确保本线程的跳转指令不会跳转到其它线程的代码块. 代码堆规范 Ψ 描述了每个

代码块的前条件 a. 任意程序点处的断言 p 是以当前线程所拥有的内存和内核状态为参数的谓词.

Π中保存了每个活动线程获得内核控制权时所应该满足的前条件, 需要注意的是 Π并不是程序规范 Φ 的一部分, 只是用于可靠性定理的证明.

程序规范	Φ	::=	(LI, LM, TM, LTD, Ψ)
锁对应不变量	LI	::=	$\{l \xrightarrow{} inv\}^*$
共享空间	LM	::=	$\{l \xrightarrow{} \{ad\}^*\}^*$
线程私有空间	TM	::=	$\{t \xrightarrow{} \{ad\}^*\}^*$
指令块所属线程	LTD	::=	$\{f \xrightarrow{} t\}^*$
代码堆规范环境	Ψ	::=	$\{f \xrightarrow{} a\}^*$
锁单元不变量	*inv*	::=	$M \xrightarrow{} Prop$
指令块规范	*a*	::=	*p*
断言	*p*	::=	$M \xrightarrow{} CS \xrightarrow{} Prop$
活动线程断言	Π	::=	$\{t \xrightarrow{} p\}^*$

图 5 MCAP 程序规范定义

3.3.2 推理规则

我们用如下的合式公式来定义推理规则:

$$\Phi;\ \Pi;\ p_i;\ p_j \ \vdash\!\!\xrightarrow{}\ P \quad (\text{合式程序})$$
$$\Phi;\ \Pi \ \vdash\!\!\xrightarrow{}\ Q \quad (\text{合式等待队列})$$
$$\Phi \ \vdash\!\!\xrightarrow{}\ C \quad (\text{合式代码堆})$$
$$\Phi;\ p_i \ \vdash\!\!\xrightarrow{}\ I \quad (\text{合式指令块})$$

这些合式公式本质上和 Hoare 逻辑的合式公式是一致的, 下面结合图 6 中具体的推理规则来进行解释.

合式程序: 一个程序是否是合式的, 是根据程序规范 Φ, 活动线程规范 Π和两个核当前运行线程的断言 p_i 和 p_j 来判定的. 在第一行中, 我们给出了等待队列中所有线程的状态. 第二行的第一个前提表示所有线程上下文的寄存器 tn 中保存了该线程的 *id*. 接下来, 我们要求 Π中包含所有线程的当前规范, 每个锁保护一块共享空间, 并对应一个描述该共享空间的不变量. 抽象机内存被划分为互不相交的每个线程的私有空间以及每个锁所保护的共享空间. 倒数第二行的第一个前提要求代码堆是合式的, 因为 Φ和 C 在程序运行过程中保持不变, 所以该前提只需一次证明便永远成立. 该行接下来的前提要求等待队列 Q 是合式的, 两个核中运行的当前线程也是合式的. Inv_ Satisfied 要求所有的空闲锁对应的不变量应在其相应空间得到满足; L_ Initial 要求在初始条件下, 已被占有的锁对应的锁空间里的值应大于或等于 1, 而空闲锁对应锁空间的值应为 0. 最后四个前提表示了当前状态应满足断言 p_i 和 p_j. 最后一行六个前提的具体定义如下:

$$\text{Inv\_ Satisfied} \triangleq \forall l \in \mathrm{dom(L)}\ \ \mathrm{L}(l) = 0 \xrightarrow{} \mathrm{LI}(l)[\mathrm{LM}(l)]$$

$$\text{L\_ Initial} \triangleq \forall i \in \mathrm{dom(T)}\ \ \mathrm{if}(loc_i = \mathrm{true})\ \ \mathrm{L}(la_i) \geqslant 1 \ \mathrm{else}\ \mathrm{L}(la_i) = 0$$

$$p_i^{\mathrm{true}} \triangleq ([p_i]_{loc} = \mathrm{true}) \xrightarrow{} p_i[\mathrm{TM}(i)] \cup [\mathrm{LM}(la_i)](\mathrm{R}_i, loc_i, la_i)$$

$$p_i^{\mathrm{false}} \triangleq ([p_i]_{loc} = \mathrm{false}) \xrightarrow{} p_i[\mathrm{TM}(i)](\mathrm{R}_i, loc_i, la_i)$$

$$[\mathrm{TM}(i)] \triangleq \{ad \xrightarrow{} \mathrm{M}(ad)\}^*\ \ \forall ad \in \mathrm{TM}(i)$$

$$[\mathrm{LM}(i)] \triangleq \{ad \xrightarrow{} \mathrm{M}(ad)\}^*\ \ \forall ad \in \mathrm{LM}(i)$$

其中 $[p_i]_{loc}$ 表示 p_i 中必须含有关于 *loc* 的谓词并将 *loc* 的值取出. 另外 [TM(*i*)]、[LM(la_i)] 分别表示了线程 *i* 所拥有的私有空间以及可能拥有的锁 la_i 保护的共享内存空间.

合式等待队列: 合式队列要求 Q 中的每个线程 *i*, 当其获得内核控制权时都能继续执行(即满足前条件 p_i), 且当前指令序列 I_i 也是合式的. 当线程不占有锁时, 前条件 p_i 描述的当前可访问空间范围仅为其私有空间[TM(*i*)], 若占有锁 la_i, 则 p_i 描述的内存范围还包括了锁 la_i 保护的共享空间[LM(la_i)].

合式代码堆: 一个代码堆是合式的当且仅当所有指令块关于程序规范 Φ是合适的.

(合式程序) (PROG)

$$Q = \{t \xrightarrow{} (\mathrm{R}_t, loc_t, la_t\}, \mathrm{I}_t)\}^*\ \forall t \in \mathrm{dom(T)} \wedge t \neq i, j$$

$$\mathrm{R}_i(\mathrm{tn}) = i$$

$$\mathrm{dom}(\Pi) = \mathrm{dom(T)}\ \ \mathrm{dom(L)} = \mathrm{dom(LM)} = \mathrm{dom(LI)}$$

$$M = \left(\bigcup_0^n [\mathrm{TM}(i)]\right) \cup \left(\bigcup_0^n [\mathrm{LM}(\mathcal{J})]\right)$$

$$\Phi \ \vdash\!\!\xrightarrow{}\ C \quad \Phi;\ \Pi \ \vdash\!\!\xrightarrow{}\ Q \quad \Phi;\ p_i \ \vdash\!\!\xrightarrow{}\ I_i \quad \Phi, p_j \ \vdash\!\!\xrightarrow{}\ \mathrm{I}_j$$

$$\frac{\text{Inv\_ Satisfied} \quad \text{L\_ Initial} \quad p_i^{\mathrm{true}} \quad p_i^{\mathrm{false}} \quad p_j^{\mathrm{true}} \quad p_j^{\mathrm{false}}}{\Phi;\ \Pi;\ p_i;\ p_j \ \vdash\!\!\xrightarrow{}\ (\mathrm{M}, \mathrm{L}, (\mathrm{R}_i, loc_i, la_i), \mathrm{I}_i, (\mathrm{R}_j, loc_j, la_j), \mathrm{I}_j, \mathrm{Q}, \mathrm{C})}$$

(合式等待队列) (QUEUE)

$$\forall i \in \mathrm{dom(Q)}, (\mathrm{CS}_i, \mathrm{I}) = \mathrm{Q}(i), p_i = \Pi(i)$$

$$\frac{p_i^{\mathrm{true}} \quad p_i^{\mathrm{false}} \quad \Phi, p_i \ \vdash\!\!\xrightarrow{}\ \mathrm{I}_i}{\Phi;\ \Pi \ \vdash\!\!\xrightarrow{}\ \mathrm{Q}}$$

(合式代码堆) (CODEHEAP)

$$\mathrm{dom(LTD)} = \mathrm{dom}(\Psi) = \mathrm{dom(C)}$$

$$\frac{\Phi;\ \Psi(f) \ \vdash\!\!\xrightarrow{}\ \mathrm{C}(f)\ \ \forall f \in \mathrm{dom}(\Psi)}{\Phi \ \vdash\!\!\xrightarrow{}\ \mathrm{C}}$$

图 6(*a*) 推理规则

合式指令序列: 在合式指令序列中我们要求事先知道该指令序列属于特定的线程(假定其线程 *id* 为 *i*), 以便推理时给出线程对应的私有空间[TM(*i*)].

spin_ acq 规则要求该线程此时未占有任何锁, 同时请求的锁必须是有效的(在 LI 域中). 第二个前提要求 p_i 中必须有关于 *loc* 的描述, 且此时 *loc* 值为 false. 第二行的前提表示若此时该锁未被占有且对应的不变量满足, 那么所寻找的中间断言 p_i' 应在更大的内存范围(包括私有空间和锁所保护的空间)内成立.

spin_ rel 规则要求该线程已经占有锁, 且要释放的锁与占有的锁为同一个, 指令执行的后条件只在该线程的私有空间上成立且释放的锁对应锁空间满足相应不变量. 第二和第三个前提要求 *ip* 中必须有关于 *loc* 和 *la* 的谓词, 且此时 *loc* 的值必须为 true, 释放的锁 *lr* 必须是当前占有的锁 *la*.

yield 规则表示任何情况下运行 yield 指令都是安全

的. 因为合式的 Q 的规则保证了任何从 Q 中取出的线程都何以在核中继续执行.

jd 规则要求目的指令块规范存在, 且该规范在当前状态下得到满足. 而且跳转指令所跳到的标签对应指令块应和该跳转指令属于同一个特定线程, 即要求指令块不能被不同的线程共享. 分支跳转指令规则与 jd 规则类似, 不再详述.

simp 规则包括以 add, sub, movi, ld, st 指令开始的指令序列的验证. 我们首先要求这些指令不能更改寄存器 tn 的值; 其次 $\mathrm{Next}_c(M, R)$ 确保了状态成功转移, 如 ld, st 指令访问的内存地址是有效的; 最后, 寻找到的中间断言 pi' 应在新的状态下成立, 并且剩余指令序列在 pi' 和 Φ 下是合式的. 指令执行前后该线程拥有的内存空间不会发生改变, 所以 pi 和 pi' 所描述的内存空间大小是相等的.

(合式指令序列)

$inv = \mathrm{LI}(la) \quad [p_i]_{loc} = \mathrm{false}$

$\forall M, R.\ p_i[\mathrm{TM}(i)](R, \mathrm{false}, \_) \wedge inv[\mathrm{LM}(la)] \rightarrow p_i'[\mathrm{TM}(i)] \downarrow [\mathrm{LM}(la)](R, \mathrm{true}, la)$

$$\frac{\Phi;\ p_i' \vdash I}{\Phi;\ p_i \vdash \mathrm{spin\_rel}\ lr;\ I} \quad (\mathrm{spin\_acq})$$

$inv = \mathrm{LI}(lr) \quad [p_i]_{loc} = \mathrm{true} \quad lr = [p_i]_{la}$

$\forall M, R.\ p_i[\mathrm{TM}(i)] \downarrow [\mathrm{LM}(lr)](R, \mathrm{true}, lr) \rightarrow p_i'[\mathrm{TM}(i)](R, \mathrm{false}, \_) \wedge inv[\mathrm{LM}(lr)]$

$$\frac{\Phi,\ p_i' \vdash I}{\Phi;\ p_i \vdash \mathrm{spin\_rel}\ lr;\ I} \quad (\mathrm{spin\_rel})$$

$$\frac{\Phi;\ p_i \vdash I}{\Phi;\ p_i \vdash \mathrm{yield};\ I} \quad (\mathrm{yield})$$

$p_i' = \Psi(f) \quad i = \mathrm{LTD}(f)$

$\forall M, CS.\ p_i^{\mathrm{fasle}} \rightarrow p_i'[\mathrm{TM}(i)]CS$

$$\frac{\forall M, CS.\ p_i^{\mathrm{true}} \rightarrow p_i'[\mathrm{TM}(i)] \downarrow [\mathrm{LM}(CS.la)]CS}{\Phi;\ p_i \vdash \mathrm{jd}\ f} \quad (\mathrm{jd})$$

$p_i'' = \Psi(f) \quad i = \mathrm{LTD}(f)$

$\forall M, CS.\ p_i^{\mathrm{fasle}} \rightarrow (CS.R(rs) = CS.R(rt) \rightarrow p_i''[\mathrm{TM}(i)]CS) \wedge (CS.R(rs) \neq CS.R(rt) \rightarrow p_i'[\mathrm{TM}(i)]CS)$

$\forall M, CS.\ p_i^{\mathrm{true}} \rightarrow (CS.R(rs) = CS.R(rt) \rightarrow p_i''[\mathrm{TM}(i)] \downarrow [\mathrm{LM}(CS.la)]CS) \wedge (CS.R(rs) \neq CS.R(rt) \rightarrow p_i'[\mathrm{TM}(i)] \downarrow [\mathrm{LM}(CS.la)]CS)$

$$\frac{\Phi;\ p_i' \vdash I}{\Phi;\ p_i \vdash \mathrm{beq}\ rs, rt, f;\ I} \quad (\mathrm{beq})$$

$c \in \{\mathrm{add}\ rd, rs, rt;\ \mathrm{sub}\ rd, rs, rt;\ \mathrm{movi}\ rd, w;\ \mathrm{ld}\ rd, rs(w);\ \mathrm{st}\ rd(w), rs\}\ rd \neq \mathrm{tn}$

$\mathrm{Next1} \triangleq \mathrm{Next}_c([\mathrm{TM}(i)], CS.R)$

$\mathrm{Next2} \triangleq \mathrm{Next}_c([\mathrm{TM}(i)] \downarrow [\mathrm{LM}(CS.la)], CS.R)$

$\forall M, CS.\ p_i^{\mathrm{false}} \rightarrow p_i'\ \mathrm{Next1}\ M\ (\mathrm{Next1}.R, CS.loc, CS.la)$

$\forall M, CS.\ p_i^{\mathrm{true}} \rightarrow p_i'\ \mathrm{Next2}\ M\ (\mathrm{Next2}.R, CS.loc, CS.la)$

$$\frac{\Phi;\ p_i' \vdash I}{\Phi;\ p_i \vdash c;\ I} \quad (\mathrm{simp})$$

图 6(*b*)　推理规则(续)

3.3.3　可靠性定理

在定义了推理规则之后, 将证明这些静态推理规则对 MCAP 抽象机操作语义的可靠性.

可靠性定理(soundness): 如果 $\Phi; \Pi; p_i; p_j \vdash P$, 那么对任何自然数 n, 存在程序 P', 使得 $P \rightarrow^n P'$.

一个程序 P 是合式的, 则表明 P 可以一直安全运行, 即该合式程序可以在 MCAP 的操作语义下不断的执行下去而不会陷入滞留状态. 为了证明该可靠性定理, 我们需要引入前进性引理和保持性引理.

前进性引理(progress): 如果 $\Phi; \Pi; p_i; p_j \vdash P$, 那么存在 P', 使得 $P \rightarrow P'$.

保持性引理(preservation): 如果 $\Phi; \Pi; p_i; p_j \vdash P$, 并且存在 P', 使得 $P \rightarrow P'$ (假定由核 0 上执行的一条指令引起的状态迁移), 那么存在 Π'、p_x', 使得 $\Phi; \Pi'; p_x'; p_j \vdash P'$.

我们已经使用辅助定理证明工具 Coq 完成了可靠性定理的证明. 限于篇幅, 证明细节在此不作详述.

4　目标代码验证举例

我们设计一个跷跷板竞赛的例子来说明在 MCAP 框架下的编程和验证. 整个程序中有三个活动线程 T0, T1, T2. 地址为 left 和 right 的内存空间均初始化为 50, 由自旋锁 0 来保护, 锁初始化为空闲($\mathrm{L}(0) \rightarrow 0$), 该锁关联的不变量 *inv* 为: M(left) + M(right) = 100. 地址为 Lscore 和 Rscore 的内存空间是线程 T2 的私有空间, 均初始化为 0. T0 每次循环的主要任务是当 left 和 right 处的值均小于 100 时, 则把 left 处的内存值加 1, right 处内存值减 1, 否则不做任何操作. T1 循环体的主要任务则刚好相反. 我们规定最先到达 100 的为获胜方, 这样哪个线程被调度执行的次数越多, 执行时间越长, 则获胜几率越大. T2 循环体的主要任务是当有一方获胜时在其相应内存记录处加 1(T0、T1 分别对应于 Lscore 和 Rscore), 并将 left 和 right 处的值重置为 50, 这样 T2 就相当于一名记分裁判, Lscore 和 Rscore 则分别是 T0 和 T1 的记分牌, 具体程序见图 7. 需要说明的是, 我们可在程序任意位置插入 yield 指令来模拟线程抢占, 为了简化描述, 例子中只在释放锁指令之后显式的给出 yield 指令.

程序的安全策略由每个指令块的前条件来描述, 为了便于理解和证明, 我们还给出了程序关键点处的断言, 具体见图 7 阴影部分, 其中用到的定义如下示:

p0: loc = false　　p1: loc = true　　p2: la = 0

p3: r1 = left ∧ r2 = right　　p4: r1 = right ∧ r2 = left

p5: r14 = Lscore ∧ r15 = Rscore

p6: M(left) = 100　　p7: M(right) = 100

同样, 我们将用定理证明器 Coq 并根据推理规则对整个程序进行合式性证明. 限于篇幅, 将不再细述.

```
entryT0:
    [p0]
    spin_acq 0
    [p1∧p2∧inv]
    movi $r1, LEFT
    movi $r2, RIGHT
    [p1∧p2∧inv∧p3]
    ld    $r3, $r1(0)
    movi $r4,0
    beq   $r3, $r4, outT0
    movi $r4, 100
    beq   $r3, $r4, outT0
    movi $r4, 1
    add   $r3, $r3, $r4
    st    $r1(0), $r3
    ld    $r3, $r2(0)
    sub   $r3, $r3, $r4
    st    $r2(0), $r3
    jd    outT0
outT0:
    [p1∧p2∧inv]
    spin_rel 0
    [p0]
    yield
    jd    entryT0
```

```
entryT1:
    [p0]
    spin_acq 0
    [p1∧p2∧inv]
    movi $r1, RIGHT
    movi $r2, LEFT
    [p1∧p2∧inv∧p4]
    ld    $r3, $r1(0)
    movi $r4,0
    beq   $r3, $r4, outT1
    movi $r4, 100
    beq   $r3, $r4, outT1
    movi $r4, 1
    add   $r3, $r3, $r4
    st    $r1(0), $r3
    ld    $r3, $r2(0)
    sub   $r3, $r3, $r4
    st    $r2(0), $r3
    jd    outT1
outT1:
    [p1∧p2∧inv]
    spin_rel 0
    [p0]
    yield
    jd    entryT1
```

图7 (*a*) T0,T1线程代码

```
entryT2:
    [p0]
    movi $r14, LSCORE
    movi $r15, RSCORE
loopT2:
    [p0∧p5]
    spin_acq 0
    [p1∧p2∧p5]
    movi $r1, LEFT
    movi $r2, RIGHT
    [p1∧p2∧p3∧p5]
    ld    $r3, $r1(0)
    movi $r4, 0
    beq   $r3, $r4, r_win
    movi $r4, 100
    beq   $r3, $r4, l_win
    [p1∧p2∧inv∧p5]
    spin_rel 0
    [p1∧p5]
    yield
    jd    loopT2
```

```
l_win:
    [p1∧p2∧p3∧p5∧p6]
    ld    $r3, $r14(0)
    movi $r4, 1
    add   $r3, $r3, $r4
    st    $r14(0), $r3
    jd    outT2
r_win:
    [p1∧p2∧p3∧p5∧p7]
    ld    $r3, $r15(0)
    movi $r4, 1
    add   $r3, $r3, $r4
    st    $r15(0), $r3
    jd    outT2
outT2:
    [p1∧p2∧p3∧p5]
    movi $r4, 50
    st    $r1(0), $r4
    st    $r2(0), $r4
    [p1∧p2∧inv∧p5]
    spin_rel 0
    [p1∧p5]
    yield
    jd    loopT2
```

图7 (*b*) T2线程代码

5 总结

本文介绍了我们为实现多核并行程序验证而设计的逻辑证明框架 MCAP. 我们用 Coq 证明了该框架的可靠性定理, 并列举了跷跷板竞争实例来证明了我们框架的正确性和可行性. 本文为多核程序验证提供了一种新的思路, 为之后的多核验证打下了一定基础.

接下来的工作中我们将对 MCAP 框架进行完善和扩展, 主要在两个方面进行. 首先我们将对 spin_ acq *la* 和 spin_ rel *lr* 两条锁操作原语用实际的汇编指令(如 X86 中的以 lock 为前缀的硬件指令等) 代码重写, 并利用本文关键思想为这些指令定义推理规则, 使得我们的抽象机更接近于真实机器. 其次为了增强规范的表达能力, 我们将引入 A-G[6] (Assume-Guarantee) 技术以达到更好的形式化描述多个线程间分工协作的目的, 同时还准备引入 SCAP[7] 中的 guarantee 技术, 使得验证能够函数模块化进行.

参考文献:

[1] G Morrisett, D Walker, K Crary, N Glew. From system F to typed assembly language[A]. In Proc ACM POPL' 98[C]. New York: ACM Press, 1998. 85- 97.

[2] G Necula. Proof-carrying code[A]. In Proc ACM POPL' 97 [C]. New York: ACM Press, 1997. 106- 119.

[3] C Colby, P Lee, G Necula, et al. A certifying compiler for Java [A]. In Proc ACM PLDI' 00[C]. New York: ACM Press, 2000. 95- 107.

[4] A W Appel. Foundational proof-carrying code[A]. In Proc 16th IEEE Symp on Logic in Computer Science[C]. IEEE Computer Society, 2001. 247- 258.

[5] Xinyu Feng, Zhong Shao, et al. Modular verification of assembly code with stack-based control abstractions[A]. In Proc ACM PLDI' 06[C]. New York: ACM Press, 2006. 401- 414.

[6] Hongxu Cai, Zhong Shao, et al. Certified self-modifying code [A]. In Proc ACM PLDI' 07[C]. New York: ACM Press, 2007. 66- 77.

[7] Xinyu Feng, Zhong Shao, Yuan Dong, Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads [A]. In Proc ACM PLDI' 08[C]. New York: ACM Press, 2008. 170- 182.

[8] Y Bertot, P Casteran. Coq' Art: The Calculus of Inductive Constructions[M]. Berlin: Springer-Verlag, 2004.

作者简介:

朱允敏 男, 1983 年生于海南省海口市, 清华大学计算机科学与技术系硕士, 研究方向为程序验证等.
E-mail: zhuyunmin@ gmail. com

张丽伟 女, 1982 年生于河北石家庄, 清华大学软件学院硕士, 研究方向为程序验证等.
E-mail: lifelong830114@ gmail. com