多核平台上的线程级猜测执行综述

郭 辉 王 琼 沈 立 王志英
(国防科技大学计算机学院 长沙 410073)

摘 要 多核体系结构的发展,使人们可以以猜测的方式挖掘应用中的粗粒度并行,线程级猜测执行(TLS)就是其中的典型代表。它的最大优点是编程模型非常简洁——程序员只需标识出那些可以猜测执行的代码段即可,运行时系统或硬件负责确保猜测线程之间的数据相关不被破坏。全面分析了现有的TLS技术,总结了当前TLS面临的挑战和未来的主要发展趋势。主要贡献包括:1)按照猜测变量的生命周期提出了一种新的TLS技术分类方法,并比较了各种已有方法的优缺点;2)根据猜测变量的生命周期,归纳了支持TLS的多核平台的设计空间,提出了探索该设计空间的若干方法;3)指出了TLS技术当前面临的挑战和未来的发展趋势。

关键词 多核,体系结构,线程级前瞻,猜测变量

中图法分类号 TP316 **文献标识码** A

Survey of Thread Level Speculation on Multi-core Platform

GUO Hui WANG Qiong SHEN Li WANG Zhi-ying
(Computer College, National University of Defese Technology, Changsha 410073, China)

Abstract The development of multi-core architecture enables researchers to explore coarse-grained parallelism by a speculative mode. Thread level speculation (TLS) is the most representative one among current speculative parallelization techniques. The most abstractive advantage of TLS is the simple programming model in which programmers only need to mark the codes that can be executed speculatively while the runtime system or hardware is responsible for the correctness. This paper analyzed the existing TLS techniques and summarized the challenges TLS confronts and the developmented trend in the future. The main contributions are: 1) we presented a novel taxonomy of TLS techniques based on the life circle of speculative variables and compared their advantages and disadvantages, 2) we summarized the design space of multi-core platform supporting TLS on the basis of the life circle of speculative variables and proposed several ways to explore it, 3) the paper pointed out the challenges TLS confronts and the development trend in the future.

Keywords Multi-core, Architecture, TLS, Speculative variable

1 引言

如何简洁、高效使用处理器中集成的丰富计算资源,始终是多核体系结构面临的严峻挑战,并将随着处理器内核数量的增加而变得越来越复杂。解决这一问题需要编程模型、运行时系统、体系结构三者互相协作、缺一不可。

为使串行应用在多核平台上获得理想的性能提升,程序员/编译器必须严格按照并行编程模型的要求完成并行化。由于运行时环境和体系结构承担的任务有限,这种方式的自动化程度并不高,其效果主要依赖于程序员的经验,特别是线程划分[1],很难做到正确、高效。TLS(thread level speculation)则是一种完全不同的机制[2],大大简化了多线程程序设计[3,4],而且完全有可能在其基础上实现自动的编译框架,特别是以串行脚本语言或二进制代码为输入的动态编译系统[31,33]。这样,许多串行应用就可以被自动并行化,并在多核平台上得到较为明显的性能提升[5]。

TLS的编程模型十分简洁,程序员只需简单地将原始程序划分为多个线程即可。例如Rochester大学Chen Ding等人提出的BOP(Behavior Oriented Parallelization)[6],只需标识出可能并行段(Possible Parallel Region, PPR),就能自动完成并行程序的执行和管理。为确保每个程序都能够跟原先一样正确执行,运行时必须追踪所有线程间的数据相关。当一个“后续”线程由于过早地读取/写入数据而引发一个真实的相关性冲突时,这个线程必须被作废,并确保它能够被重新执行。相比于其他并行编程模型,TLS一个最大的优势就是它允许可能存在数据相关的并行段并行执行,并且提供了一种有效的机制,其能够以简洁的编程模型挖掘应用中的线程级并行。TLS能够并行化那些难以分析的程序结构,比如do-while循环。M. Feng等人开发的SpiceC支持一组类OpenMp[30,32]的编译指导语句,不仅可以实现传统的并行编程模式,而且支持Speculative DOACROSS模式[11]。

人们已经对TLS系统以及建立在其上的自动并行化机

到稿日期:2013-03-25 返修日期:2013-06-17 本文受国家自然科学基金(61272143,61272144),863项目(2012AA010905)资助。
郭 辉(1988-),男,硕士生,主要研究方向为计算机体系结构,E-mail:samgh@outlook.com;王 琼(1989-),女,博士生,主要研究方向为计算机体系结构;沈 立(1976-),男,博士,副教授,主要研究方向为计算机体系结构;王志英(1956-),男,博士,教授,主要研究方向为计算机体系结构。

制进行了大量的研究，有些工作完全依赖硬件[9,24,25,34]，也有些采用纯软件方式实现[6,7,18,20,21,25,27]。尽管它们都证明了TLS的性能潜力，但也从不同角度暴露了TLS的缺陷与不足。例如，一些工作基于Cache或硬件缓冲实现TLS[9,10,12,14]，但也增加了处理器核流水线关键路径的长度，大大降低了处理器核的主频。一些工作通过软件实现TLS中的关键模块，如线程确认、猜测错误时的回滚等，但软件自身开销过大且缺乏必要的硬件支持，使得这些方法仍然有相当的提升空间。California大学的Chen Tian等人采用类似的线程确认机制，利用线程空间实现了支持TLS机制的CorD(Copy or Discard)[8]执行模型。

本文分析和比较了现有的TLS技术，总结了TLS技术当前面临的挑战和未来的主要发展趋势。本文的主要贡献有：

1)按照猜测变量的生命周期，提出了一种新的TLS技术分类方法，并比较了现有各种方法的优缺点；

2)根据猜测变量的生命周期，归纳了支持TLS的多核平台的设计空间，提出了探索该设计空间的若干方法；

3)指出了TLS技术当前面临的挑战和未来发展趋势。

本文第2节提出了猜测变量生命周期的概念以及在其生命周期的各个阶段需要解决的关键问题，并以猜测变量的生命周期为基础，提出了一种TLS实现技术的分类方法；第3节提出了支持TLS的多核平台的设计空间，并比较了现有实现技术的优缺点；第4节深入探讨了目前TLS系统尚需解决的问题，以及未来可能的发展趋势；最后对本文工作进行了总结。

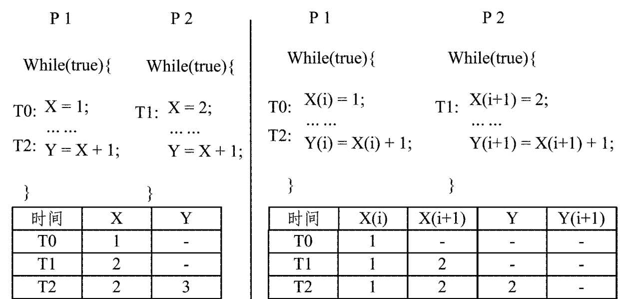
2 猜测变量的生命周期

TLS系统在实现程序并行化、减轻程序员编程负担等方面具有强大的优势，它以一种非常“激进”的方式挖掘程序中的线程级并行——猜测线程执行时可以忽略彼此之间的数据相关。当然，系统必须提供专门机制来作废那些破坏了原有数据相关的猜测结果。因此，要实现一个TLS系统，必须要完成的一项工作就是追踪线程间的所有数据相关，以确保程序正确执行。与传统的非猜测执行方式不同，在TLS执行方式中，会出现多个猜测线程对同一个变量执行写操作的情况。这样的变量称为“猜测变量”。

在TLS系统中，猜测变量经过了创建、使用、提交等3个阶段，构成了一个完整的生命周期：在猜测线程写一个变量之前，它会为这个猜测变量创建一个副本；若多个猜测线程都修改了该变量的值，这个变量将拥有多个副本；其它猜测线程访问该变量时，TLS系统需要在该变量的众多副本中寻找合适的版本；当猜测变量最终要被提交时，TLS系统需要验证被提交的结果的正确性，若结果正确，则将结果写回非猜测数据区。本节将分析现有TLS系统在各个阶段的实现方法。

2.1 猜测数据产生阶段

TLS的执行方式与非TLS方式相比，最大的区别在于TLS方式下，一个变量可能有多个副本(猜测副本)，而非TLS方式则通过Cache一致性机制保持唯一的副本。为了访问对应的猜测副本，每个猜测线程必须建立变量与副本之间的地址映射。在TLS机制下，由于猜测线程的执行速度不同或者执行路径不同，有可能破坏非猜测执行的数据相关。如图1所示，若$T1$时刻早于$T2$时刻，则线程i写入的x值会在线程j写入时被覆盖掉，从而导致线程i的计算错误。当猜测线程首次对某个变量写时，它需要为变量产生相应的副本或者版本，以此来区分变量在不同猜测线程中的状态。例如，Rundberg[20]设计的TLS系统为每个猜测变量建立一个数据结构，记录哪些猜测线程对这个变量进行过读/写操作，以及当前线程中这个变量的值。当某个猜测线程对这个变量进行写操作时，系统会将这个线程的标号记录下来并且更新它的局部值。通过引入猜测变量的副本，TLS系统可以尽可能地减少WAW和WAR相关，并且可以快速检测是否发生了数据相关。



时间	X	Y
T0	1	-
T1	2	-
T2	2	3

时间	X(i)	X(i+1)	Y	Y(i+1)
T0	1	-	-	-
T1	1	2	-	-
T2	1	2	2	-

图1 共享变量在多线程中的问题以及解决方案

当前有一些TLS系统会利用程序员不可见的底层硬件(比如Cache或者硬件缓存)来保存猜测变量的副本。其中有一些会为Cache增加一些新的标志位和状态位，以此来标识出新定义的一些状态和索引标志[9,12-14,19]。Steffan提出的STAMPede[9]系统，给每个Cache块增加了2个新的状态，Speculatively Loaded(SL)，Speculatively Modified (SM)，分别用来表示数据是否被猜测读或者猜测写。该系统以Epoch为一个任务单位，如图2(a)所示，在Epoch 3的Cache中存放变量X的同时加了SL和SM来标识该Cache块的猜测状态。Gopal等人设计的SVC[14](Speculative Versioning Cache)为每个Cache块增加一位Load标志，用来表示此Cache块在被写之前已经被前面的任务读过了。同时它还为每个Cache块增加一个指针位，该指针指向拥有下一个备份版本的Cache，如图2(b)所示。另外一些系统会在存储层次结构中间加入Write Buffer[10,18]。Stanford大学研制开发的Hydra[10]原型系统如图2(c)所示，就是在L1 Cache和L2 Cache之间为每个Cache增加了一级Write Buffer，当有猜测线程对变量进行写操作时，不是直接更新到第二级共享Cache中，而是先缓存到Write Buffer里，直到猜测数据确认后才提交。

另外一方面，研究人员通过软件在可访问区域(比如，内存等)为猜测线程分配相应的区域来保存猜测变量副本[6-8,11,15,16,20-23]。Chen Tian提出的Copy Or Discard[7](CorD)系统采用静态编译技术。如图3所示，它将整个存储空间划分为3个状态区域：Non speculative State(D)，Parallel or Speculative State(P)和Coordinating State(C)。在程序编译阶段，编译器为每个猜测线程各分配一个P区域和一个C区域，其中P区域就是猜测线程用来计算的区域，而C区域则主要完成与主线程通信同步等后期处理任务。另一方面，人们也采用了动态的方式来解决TLS系统中数据管理的问

题[6,7,15,16,21,23]。罗切斯特大学的 Chen Ding 根据操作系统中 Copy-on-Write 的原理(见图 4)设计了 BOP[6,28] 系统。BOP 利用进程执行每个任务,同时采用基于页的保护机制,当对一个共享变量进行写操作时,记录下该变量所在页的访问模式,并由操作系统为其分配一个私有页来保存变量修改后的值。普林斯顿大学的 Raman[15,16] 则采用了另外一种方法。它在每个猜测线程中记录一个类似于页表的结构,这个结构会记录数据内存页的索引,包括共享的和私有的页。当一个进程要对其中一个猜测变量第一次进行写操作时,操作系统会动态申请一个新页,并修改私有页的指针指向这个新页。

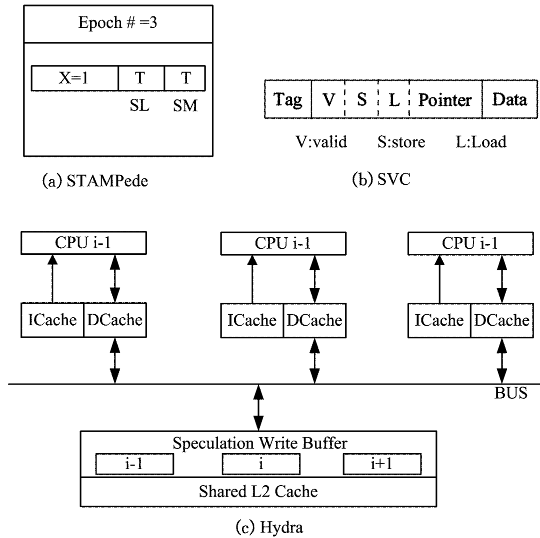


图 2　利用硬件方法实现猜测数据的管理

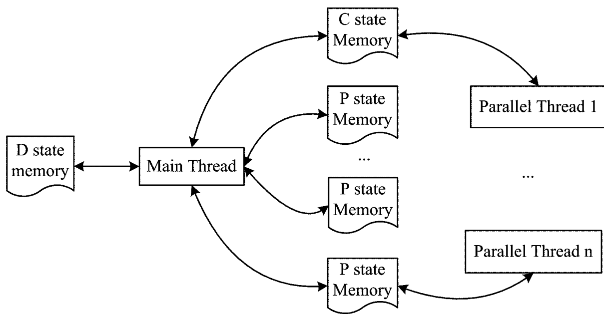


图 3　CorD 的存储空间管理

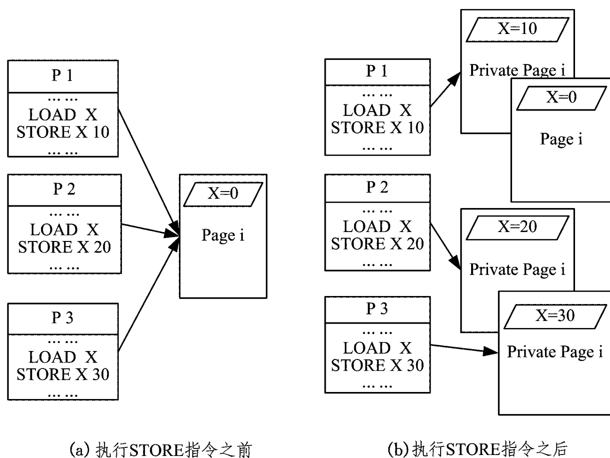


图 4　Copy-on-Write 机制

2.2　猜测数据使用阶段

猜测数据变量产生后,不仅猜测线程自身要访问,有时还会发生跨线程访问,也即存在 RAW 或 WRW 相关,但是 WRW 相关通过第一阶段可以消除,所以这里只考虑 RAW 相关。如图 5 所示,当 *P*1 的条件为真而 *P*2 的条件为假时,*P*2 在执行 else 语句时,*P*1 的变量 *y* 和 *P*2 的变量 *z* 产生了 RAW 相关。遇到这种情况,TLS 系统就要决定在众多猜测版本中哪个猜测线程产生的数据才是最有可能正确的数据。之所以称为最有可能正确,是因为猜测线程真正请求的结果可能还没有产生,但是为保证程序运行就要提供一个最接近正确结果的猜测版本。例如,SVC[14] 为 Cache 增加了一个 Version Control Logic(VCL)的组合逻辑。当请求数据失效时,VCL 会产生一个总线请求并监听 L1 Cache 和下一级存储结构。VCL 会根据总线的应答、任务之间的顺序以及 Version Ordering List(VOL)来选取合适的数据。除此之外,为了提供确认时的信息,TLS 还需要记录变量访问的所有信息,包括操作类型、访问的版本等,以便于确认时比较。

P1 (cond=T)	P2 (cond=F)
if(cond){	if(cond){
X=1;	X=1
II II	II II
Y=X+1;	Y=X+1;
}	}
else	else
Z=2Y;	Z=2Y;

图 5　相邻两次迭代之间产生 RAW 相关

目前在 TLS 系统中,最简单的一种方法是将存在数据相关的程序段放在并行段之外执行[6-8,11,13,15,16,19,23]。BOP 系统采用进程作为任务的执行载体,由于进程空间是相互独立的并且进程通信的开销过大,因此在实现时,BOP 将难以消除掉的存在数据相关的程序段全部放在并行段之外执行。同样,SoftSpec 也采用了相同的策略。

显然,以上的方法没有充分挖掘程序的并行度,降低了程序的性能。在 Hydra 原型系统中,如果所要取的数据是从数据 Cache 中得到的,若写标志未置 1,则读标志置 1。如果数据 Cache 失效,那么读标志和 modified 标志同时置 1,并且向次级缓存请求数据,每个字节都要从不同渠道选择最新的值,组装成新的块送回请求数据的 Cache。这就是利用修改过的 Cache 一致性协议来达到猜测线程间传输数据的目的[10,14,17,18]。

另外,还有一些系统为了能够使用猜测线程产生的数据,就在系统中为每个猜测变量声明了一个共享数据结构,来同步不同猜测线程之间的数据并记录对猜测变量 Load/Store 的操作[9,12,20-22]。在 STAMPede 中,为每个猜测线程提供了一个 Forwarding Frame 的结构,并且提供了一个软件接口 *wait_for_value*()。当线程需要对某个变量进行同步时,只要调用这个函数即可。在 Rundberg 实现的 TLS 系统中,每个猜测变量都有一个数据结构,包含了 Load Vector,Store Vector 等。线程通过 forwarding 来获取最新的猜测变量的值。

2.3　猜测数据确认阶段

在猜测数据写入主存之前,TLS 系统需要对所有可能出现数据相关冲突的变量进行排查。一旦发现有数据相关冲突

出现，就判定该线程猜测失败，所有猜测计算结果都要作废并且线程要重新执行。这就需要考虑冲突检测时机的问题，一种方法是当猜测线程进行读取或写入操作时进行检测。这样做可以提早发现数据相关冲突，但是如果变量被多次读写，就会引起重复检测，抵消了并行带来的性能提升。另一种方法是在猜测线程执行完毕之后。这种方法避免了由前一种方法引起的冗余，但是一旦出现冲突相关，检测的滞后就会导致处理器做很多“无用功”。另一方面，由于要对几乎所有的猜测变量进行检查，因此这项工作非常耗时。高效、准确的检测算法也是决定性能好坏的关键。例如，CorD[8]则采用了基于版本号的检测算法。在线程提交结果之前，将C区域内记录的相关变量的版本号根据信息与D区域相应变量的版本号进行比较，如果符合，则可以提交结果，否则，结果就要作废，线程重新执行。

例如STAMPede[9]的主要思想是在程序执行时由数据产生方将产生的数据地址发放给数据使用方，数据使用方记录数据的使用情况并判断是否有冲突发生。这一方法和写作废协议相似，只要记录下哪些线程猜测读取了哪些变量，当有变量被串行程序语义上应该先执行的线程进行了写操作，那么冲突就发生了。而Hydra[10]则利用Cache块增加的标志位进行检测。当猜测线程执行Store指令时，若该线程属于较早的线程，那么就检查相应的读标志位是否为1，如果为1，那么就发生了数据相关冲突。

与STAMPede和Hydra的方式不同，BOP系统在每个PPR执行完毕之后，会对所有猜测读取的变量进行值比较。所谓值比较就是比较猜测线程读取变量的值是否和串行执行时使用的变量值相同，如果不同就证明猜测线程执行错误，所有计算结果无效。

通过以上分析，可以明确TLS系统在各个阶段需要提供的必须要完成的工作，如表1所列。

表1 各阶段的技术要求及实现方法汇总

阶段	技术要求	实现方法
猜测数据产生阶段	线程首次写入变量时，为其产生副本； 建立变量与副本的地址映射；	增加Cache标志位，例如，文献[9,10]； 在内存中分配相应的空间，例如，文献[6,8]
猜测数据使用阶段	能够在众多变量版本中选取合适的结果； 记录访问信息；	采用共享变量进行同步，例如，文献[15]； 利用Cache一致性协议，例如，文献[10]
猜测数据确认阶段	检测冲突的时机； 高效、准确的检测算法；	数据访问时检测，例如，文献[10]； 线程提交时检测，例如，文献[8]

3 现有技术的分类和比较

3.1 设计空间

根据上节的描述和分析，不难归纳出基于猜测变量生命周期的TLS系统设计空间，如图6所示。其中横轴代表猜测数据的产生阶段，纵轴代表猜测数据的使用阶段。

在猜测数据产生阶段，首先根据猜测变量存储位置对程序员是否可见，可以分为硬件缓存(Cache/Buffer)和内存两种方式。然而，采用内存的方式又因为管理方式的不同分为静态编译和动态管理两种方式。因此，根据以上分析，在这一阶段大致可以分为硬件缓存、静态编译和动态管理3种方法。其中硬件缓存的方法(Hardware Support)是在猜测线程第一次写入变量值时利用Cache或者硬件Buffer保留猜测数据，并记录下相应的标志位、版本号等信息，线程通过这些标志来确定要访问的猜测变量。静态编译的方法(Static Compiler Method)利用编译器为每个猜测线程分配一个与原有数据存储空间相对应的内存空间并建立数据之间的地址映射。所有猜测线程都通过分配好的存储空间进行数据访问和计算，同时计算结果也保存在此等待确认。动态管理方法(Dynamic Method)主要利用操作系统中存储管理的动态机制(如Copy on Write)来实现管理。起初，所有猜测线程共享数据，只有当猜测线程写入某个变量时，操作系统才会为猜测线程分配相应的存储空间以保存猜测数据。另外，变量之间的地址映射由操作系统来完成。

使用阶段 \ 产生阶段	Hardware Support	Dynamic Method	Static Compiler Method
Cache Coherence	Hydra; Speculative Versioning Cache;		
Private Memory	SoftSpec; Software and hardware for exploring speculative parallelism with a multiprocessor	Speculative decoupled software pipelining; BOP; FAST TRACK;	SpiceC; CorD;
Public Data Structure	A Scalable Approach to Thread-Level Speculation; STAMPede;	Lightweight;	LRPD;

图6 根据猜测数据生命周期划分的TLS系统设计空间

在猜测数据使用阶段，根据是否支持猜测线程间通信以及通信采用的技术，可以把这一阶段的实现技术分为3类：私有存储、Cache一致性协议和共享数据结构。私有存储(Private Memory)，即线程的内存空间是完全私有的，不允许其他猜测线程访问其内部数据，所有猜测线程都要从非猜测线程获取，如果线程间存在数据相关，那么只能通过串行执行来处理。Cache一致性协议(Cache Coherence)是在增加Cache状态标志位的基础上，修改Cache一致性协议，并增加硬件逻辑电路选择合适的变量版本。共享的数据结构(Public Data Structure)主要是利用TLS系统中专门设计的共享数据结构，这些共享数据结构可以记录猜测线程的访问信息、猜测线程写入的猜测数据等内容。当然，在访问这些共享数据结构时，需要进行加锁。

在猜测数据确认阶段，根据检测时机的不同，可以分为Online和Offline两类。程序每次对共享变量执行读写操作时，都要进行数据相关的检查，如果发生了数据相关冲突，那么该猜测线程猜测就失效，必须废弃线程之前的工作，重新执行，这种方法是Online的方法。与之对应的Offline的方法则将变量的数据相关检测全部移到猜测线程执行完毕之后进行，将原来分散的数据相关冲突检测集中处理。

通过对TLS系统的分析，可以将现有的典型TLS系统和我们给出的设计空间对应起来，如图6所示。

3.2 设计空间探索方法

上一节围绕猜测数据生命周期对现有TLS系统实现技术进行了分析，得到了TLS系统的设计空间，并以此对现有的TLS系统进行了分类。在此基础上，我们将进一步分析它

们的优缺点，并给出理想情况下最优的 TLS 系统实现方法。

在猜测数据产生阶段，TLS 主要解决了在猜测线程第一次对变量进行写操作时，如何产生变量副本并且建立副本与原始变量地址映射的问题。对于采用硬件缓存的方法来说，对性能的影响主要是增加了处理器核关键路径的长度，大大降低了处理器核的主频。Hydra[10] 系统实现的处理器核的频率大约只有 250MHz。正是这个制约因素导致 TLS 系统不能完全采用硬件实现。而对于软件实现来说，巨大的软件管理开销以及缺乏相应的硬件支持是它的性能瓶颈。具体来说，静态编译的方法过多地依赖于编译器的强大功能，并且对于大型数据结构，会导致它的管理开销急剧增大。CorD[8] 虽然较好地实现了 TLS 的功能，但是由于缺乏相应的源到源编译器支持而不能自动完成程序的并行化。在这一点上，BOP[6] 采用动态管理的方式实现了较为简洁的编程模式。但另一方面，TLS 系统的性能开销也转移到操作系统的调用上。另外，在 C 语言中，动态申请内存的情况需要额外留意，尤其是静态编译方法。因为在编译过程中，编译器只是对静态声明的变量做处理，所以如果要引用动态变量，就要做好相应的地址映射。

在猜测数据使用阶段，主要面临的是猜测线程间通信的问题。虽然有各种手段可以解决通信的问题，但是使用通信的开销都是十分巨大的。比如，通过 Cache 一致性协议来达到通信的目的。Cache 之间通信基本上是将请求发送到总线上，再由相应的 Cache 做出响应。这一过程大概会耗费几十个时钟周期，这样的开销几乎抵消掉了并行执行获得的性能提升。又比如，利用共享数据结构作为通信的接口。这种机制思路比较简单，但是实现时需要进行锁操作，这从另一方面增加了实现的难度，而且引进锁操作就相当于强制串行，降低了并行度。STAMPede[9] 通过设置不同的通信延迟开销，发现当延迟达到 50～60 个周期的时候，大部分通过并行获得的性能提升都被抵消掉了。但是，线程间通信从根本上可以提高线程猜测成功的概率，所以高效的通信解决方案也是值得研究的方面。

猜测数据确认阶段是 TLS 保证程序正确的最后一道关卡，因此也是 TLS 开销的重灾区。TLS 需要在这一阶段完成对猜测变量的数据相关冲突检测。根据检测时机的不同，分为了 Online 和 Offline 两种。采用 Online 的策略可以尽早地发现数据冲突，避免在错误的道路上越走越远。但是，另一方面，每次数据访问都要检测，那么就会出现冗余，因为之后可能还会访问该变量，这样无形中，检测的开销就会加倍。采用 Offline 机制相当于 TLS 的收尾工作，它会在每个猜测线程的末尾集中处理数据相关。因此，如果很早之前就发生了数据相关冲突，那么该猜测线程就做了许多无用功。如何权衡它们之间的关系，也是降低 TLS 开销的主要方面。

如何将 3 个阶段的技术进行排列组合，实现编程模型简单、性能优越的 TLS 系统是未来 TLS 技术发展需要解决的主要问题。

根据前面的分析，可以发现 BOP[6]，SpiceC[11]，CorD[8] 等这类采用软件方式管理猜测变量副本的 TLS 系统都具有较为简洁的编程接口，尤其是 BOP，它只需程序员标示出可能的并行段就可以按照要求完成并行化工作。这类系统一般都采用静态编译技术或者动态管理技术。所不同的是，静态编译技术需要一个与之配合的源到源编译器来完成代码转换的工作，这也是目前工作的难点。另一方面，动态管理技术大部分采用进程作为执行单元，虽然省去了开发编译器的麻烦，但是它需要解决系统开销大的问题。还有一个问题就是实现猜测线程间的通信，以此减少由于猜测失败导致的不必要的回滚。对于采用软件实现的 TLS 系统，目前要么不允许通信，要么就是利用共享数据结构。在一些情况下，这样的实现非但没有提高系统的性能，反而会成为性能的瓶颈。在这样的前提下，可以考虑 Cache Coherence 或者类似的机制，配合相应的硬件完成数据同步。采用这样的实现的一个主要问题是如何在不改变处理器主频的情况下，添加必要的硬件。在确认猜测正确性上，软件方法一般会采用 Offline 的方式，这样不会带来过多的额外开销。而检测的算法基本上是基于版本号或者基于值的比较。

4 多/众核平台下 TLS 面临的挑战

TLS 技术的提出，本质上是为了给程序员提供简洁的编程模型，在这个基础上尽可能地挖掘程序中 TLP。但是，目前实现的 TLS 系统还与这个要求相差一段距离。要达到这个目标就需要妥善处理好编程模型、运行时系统和体系结构 3 者之间的关系。通过研究未来 TLS 系统可能的实现方式，我们发现了实现 TLS 系统尚需解决的一些问题。

首先，现有的研究不仅证明了 TLS 机制可以有效解决简洁、高效利用多核/众核平台计算资源的挑战，也指出了只有解决 TLS 自身的问题，如复杂的数据相关检测、串行的猜测线程确认、猜测线程的调度与回滚、猜测结果的缓存与访问、性能与功耗的平衡等，才能真正发挥其作用。

其次，无论采用何种方式实现 TLS，都必须提供高效的猜测数据存储和管理机制，因为这些数据访问次序和值不仅是检测线程猜测执行结果是否正确的主要依据，也是其它与之相关的猜测线程能顺利执行的前提。猜测线程的执行会引发大量有关猜测数据的存储和管理操作，且其复杂度将随处理器核和猜测线程数量的增加而变大。

第三，当前的 TLS 研究多以纯软件方法为主，缺乏必要的硬件支持，自身开销较高，而纯硬件方法其实现复杂度过高，难以真正实现。因此，实现高效的 TLS 机制应采用软硬件协同的方法，并由体系结构和运行时环境提供有效的支撑。

第四，现有基于 TLS 的静态编译技术取得了不小的进展，也进一步证明了 TLS 机制的潜力，但这种方式自动化程度仍然不高，而且受到 TLS 机制自身开销的限制，划分得到的线程粒度也无法令人满意。而动态编译的自身开销过大，其效果也远远无法令人满意。

结束语 本文提出了一种新颖的针对 TLS 机制的分类方法，即按照猜测数据生命周期中的产生、使用和确认 3 个阶段，对每一阶段中不同的实现方法进行归类。通过分析每一类别的基本思想和具体实例，总结出了各类的优势和缺陷，为今后的 TLS 系统设计提供了重要的参考价值。最后，讨论了 TLS 系统未来可能的发展趋势。

参考文献

[1] Lee J，Wu Hai-cheng，Ravichandran M，et al. Thread tailor：dy-

namically weaving threads together for efficient, adaptive parallel applications[C]//ISCA. 2010
[2] Speculative Multithreading[OL]. http:// en. wikipedia. org/wiki/Speculative_multithreading
[3] Prabhu M, Olukotun K. Using thread-level speculation to simplify manual parallelization[C]// Proc. of the 2003 Principles and Practices of Parallel Programming. 2003
[4] Prabhu M, Olukotun K. Exposing speculative thread parallelism in SPEC 2000[C]//Proc. of the 2005 Principles and Practices of Parallel Programming. 2005
[5] Olukotun K, Hammond L, Laudon J. 片上多处理器体系结构，改善吞吐率和延迟的技术[M]. 王东升，王海霞，李鹏，译. 北京：机械工业出版社，2009
[6] Ding Chen, Shen Xi-peng, Kelsey K, et al. Software behavior oriented parallelization[C]// Proc. of ACM SIGNPLAN Conf. on Programming Language Design and Implementation. 2007
[7] Kelsey K, Bai T, Ding C, et al. Fast track: A software system for speculative program optimization[C]//Proceedings of the International Symposium on Code Generation and Optimization (CGO). 2009: 157-168
[8] Tian Chen, Feng Min, Nagarajan V, et al. Copy or discard execution model for speculative parallelization on multicores[C]//the 41st IEEE/ACM Intl. Symp. on Microarchitecture. 2008
[9] Steffan J G, Colohan C, Zhai A, et al. The STAMPede approach to thread-level speculation[J]. ACM Trans. on Computer Systems, 2005, 23(3)
[10] Hammond L, Hubbert B, Siu M, et al. The Stanford Hydra CMP [J]. IEEE MICRO Magazine, 2000, 20(2): 71-84
[11] Feng M, Gupta R, Hu Y. SpiceC: scalable parallelism via implicit copying and explicit commit[C]//Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2011: 69-80
[12] Steffan J, Colohan C, Zhai A, et al. A Scalable Approach to Thread-Level Speculation[C]// Proc. 27th Annual Intl. Symp. on Computer Architecture. June 2000: 1-12
[13] Oplinger J, Heine D, Liao S-W, et al. Software and hardware for exploiting speculative parallelism with a multiprocessor[M]. Computer Systems Laboratary, Stanford University, 1997
[14] Gopal S, Vijaykumar T N, Smith J E, et al. Speculative Versioning Cache[C]// Proc. 4th Intl. Symp. on High-Performance. Computer Architecture, February 1998: 195-205
[15] Raman A, Kim Han-jun, Mason T R, et al. Speculative parallelization using software multi-threaded transactions[C]// Proc. of the 15th Architecture Support for Programming Languages and Operating Systems. 2010
[16] Vachharajani N, Rangan R, Raman E, et al. Speculative decoupled software pipelining[C]//PACT. 2007: 49-59
[17] Cintra M, Martinez J F, Torrellas J. Architectural support for scalable speculative parallelization in shared memory systems [C]//Proc. of the 27th Int. Symp. on Computer Architecture. 2000
[18] Tsai J Y, Huang J, Amlo C, et al. The Superthreaded Processor Architecture[J]. IEEE Trans. on Computers, 1999, 48(9): 881-902
[19] Bruening D, Devabhaktuni S, Amarasinghe S. Softspec: Software-based speculative parallelism[C]// Proceedings of the 3rd ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO3). 2000
[20] Rundberg P, Stenström P. An all software thread-level data dependence speculation system for multiprocessors[J]. Journal of Instruction-Level Parallelism, 2001, 3(1)
[21] Oancea C E, Mycroft A, Harris T. A lightweight in-place implementation for software thread-level speculation[C]// Proc. of the 21st Annual Symp. on Parallelism in Algorithms and Architectures. 2009
[22] Rauchwerger L, Padua D. The LRPD test: Speculative run-time paralle-lization of loops with privatization and reduction parallelization[C]//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. La Jolla, CA, June 1995
[23] Pyla H K, Ribbens C, Varadarajan S. Exploiting Coarse-Grain Speculative Parallelism[C]// Proceedings of the ACM international conference on Object oriented programming systems languages and applications. 2011: 555-574
[24] Olukotun K, Hammod L, Willey M. Improving the performance of speculatively parallel applications on the Hydra CMP[C]// Proc. of the 1999 ACM Intl. Conf. on Supercomputing. 1999
[25] Steffan J G, Goldstein S C. Hardware support for thread-level speculation[D]. Carnegie Mellon University, 2003
[26] Kazi H, Lilja D J. Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors [J]. IEEE Trans. on Parallel and Distributed Systems, 2001, 12 (9)
[27] Pickett C J F, Verbrugge C. Software thread level speculation for the java language and virtual machine environment[C]// Proc. of the 18th Intl. Workshop on Languages and Compilers for Parallel Computing. 2005
[28] Ke Chuan-le, Liu Lei, Zhang Chao, et al. Safe parallel programming using dynamic dependence hints[C]// OOPSLA. 2011: 243-258
[29] Garzaran M J, Prvulovic M, Llaberia J M, et al. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessor[J]. TACO, 2005, 2(3)
[30] OpenMP[OL]. http://www. openmp. org/
[31] Kim H, Johnson N P, Lee J W, et al. Automatic speculative DOALL for clusters[C]// Proceedings of the Tenth International Symposium on Code Generation and Optimization. ACM, 2012: 94-103
[32] Aldea S, Llanos D R, González-Escribano A. Support for thread-level speculation into OpenMP, OpenMP in a Heterogeneous World[M]. Berlin Heidelberg: Springer, 2012: 275-278
[33] Hertzberg B, Olukotun K. Runtime automatic speculative parallelization[C]//Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society, 2011: 64-73
[34] 赖鑫，刘聪，王志英. 支持线程级猜测的存储体系结构设计的存储体系结构设计[J]. Computer Engineering, 2012, 38(24)