支持推测多线程的扩展多核模拟器 Prophet+

宋少龙，赵银亮，冯博琴，韦远科，王旭昊，赵恒星
（西安交通大学计算机科学与技术系，710049，西安）

摘要：推测多线程技术是软硬件协同开发非规则应用程序的线程级并行性的有效方法.通过体系结构对推测并行执行模式的支持，编译器产生推测多线程目标代码实现自动并行化加速.文中针对硬件平台如何有效支持程序运行过程的分析及测试，提出了一种支持推测并行执行模式的超标量流水线结构和一种基于独立栈的运行时内存空间管理方法.通过对 Prophet 模拟器扩展实现，并经 Olden 基准程序测试表明，扩展后的 Prophet+ 在保持 Prophet 模拟器性能的前提下，提高了模拟器的精度和灵活性.

关键词：线程级并行；推测多线程；片上多处理器；流水线；模拟器
中图分类号：TP302 **文献标志码**：A **文章编号**：0253-987X(2010)10-0013-05

Prophet+：an Extended Multicore Simulator for Speculative Multithreading

SONG Shaolong，ZHAO Yinliang，FENG Boqin，WEI Yuanke，WANG Xuhao，ZHAO Hengxing
(Department of Computer Science and Technology，Xi′an Jiaotong University，Xi′an 710049，China)

Abstract：Speculative multi-threading (SpMT) is an effective approach to exploit thread level parallelism of irregular programs based on co-design of hardware and software. SpMT compilers generate SpMT target code from sequential source codes，and give acceleration on multi-core architecture with the support of special SpMT execution model. A superscalar pipeline structure for SpMT execution model and a runtime memory management approach，named independent stack，is presented for hardware platform to effectively support empirical study and performance evaluation of SpMT programs. The implementation of Prophet Simulator is extended and is evaluated using Olden benchmark. Results show that the extended Prophet improves the accuracy and flexibility，while maintains the performance of Prophet.

Keywords：thread level parallelism；speculative multi-threading；chip multiprocessor；pipeline；simulator

传统的自动并行化虽然在科学计算领域取得了成功，但对于非规则程序的加速效果却不明显[1-2].为了开发非规则应用程序的线程级并行性，研究人员提出了推测多线程(Speculative Multithreading，SpMT)技术.SpMT 技术是一种软硬件协同的新技术，编译器采用激进的线程生成方法，允许线程间存在依赖，由硬件保证程序执行的正确逻辑.因此，编译器能够充分挖掘串行程序中的线程级并行性，如果目标线程在运行时真正发生了依赖违规，则由硬件支持的推测执行模式保证出现依赖违规的线程被撤销并按照正确的执行逻辑继续.

目前针对 SpMT 执行模式[3-4]的研究特定于具体的推测多线程自动并行化方法，还没有通用的模拟器和多核处理器出现，这就限制了对 SpMT 技术的有效性进行测试与评估的工作.文献[4]用 Verilog 实现了 Pinot 原型系统.通过硬件描述语言实现的模拟系统虽然能够详尽地模拟硬件的操作，但是实现起来复杂费时，不适合快速建立原型系统.文献

收稿日期：2010-06-09. 作者简介：宋少龙(1984—)，男，硕士生；赵银亮(联系人)，男，教授，博士生导师. 基金项目：国家“863 计划”资助项目(2008AA01Z136).

[5]基于 SMTSIM 实现了 Mitosis 模拟器，然而该模拟器只能够模拟执行 Olden 基准程序集中的部分程序. 此外，文献[4]和文献[5]中没有对多线程下的内存管理机制做深入的研究.

本文主要介绍了扩展模拟器 Prophet＋的设计与实现，提出了一种解决 SpMT 技术与超标量流水线相结合时产生的问题的方法，并提出了一种解决多线程同时访存问题的独立栈方法，研究了多线程时的动态内存空间管理机制.

1 Prophet 执行模型

文献[6]详细描述了 Prophet 执行模型. 图 1 展示了 Prophet 的线程模型. Prophet 中有两类线程，分别称为确定线程和推测线程. 确定线程是指执行期间所使用的数据确定、控制流确定的线程. 它从串行语义逻辑上推进整个程序执行的进程，运行时只有一个确定线程. 推测线程由预计算片段[6-7]（pre-computation slice，p-slice）和一段来自串行程序的代码片段组成. 在 Prophet 中 p-slice 是由 Prophet 编译器生成的一小段代码，推测线程使用 p-slice 计算它们的 live-ins. 由于 live-ins 是推测计算的，推测线程的执行路径也是程序运行过程中最可能执行的路径，因此它们都有可能预测错误. 这些错误的检测以及恢复都是由底层硬件负责的. 在图 1 中线程 1 是确定线程，其他两个线程为推测线程. Prophet 编译器会在 SP（Spawn Point）处插入特殊指令，表示此时可以发起一个新的线程. 线程在遇到 SP 时需要检查此时的系统资源，如是否有空闲的处理器等，如果有足够的资源，则发起一个新的推测线程. 准控制无关点 CQIP（Control Quasi-independent Point）既是前一个线程的结束点，同时也标识了下一个线程的开始. CQIP 是由 Prophet 编译器通过对程序的控制流进行分析得到的. 确定线程（线程 1）执行到 CQIP 时验证子线程（线程 2）在 p-slice 中产生的 live-ins，如果验证通过，线程 1 提交自己执行过程中产生的数据，将确定执行权限传递给线程 2，然后交还系统资源并退出. 确定执行权限类似于一个令牌，它在线程中以串行逻辑顺序传递. 如果验证失败，线程 1 撤销其所有子线程，在本例中为线程 2 和线程 3，并将自己的 PC 指向线程 2 预计算片段后的第一条指令继续执行. 推测线程执行到 CQIP 时转入等待验证状态. 如果在推测线程执行期间发生了数据依赖或者控制依赖，则由底层硬件撤销并重新执行这些线程体.

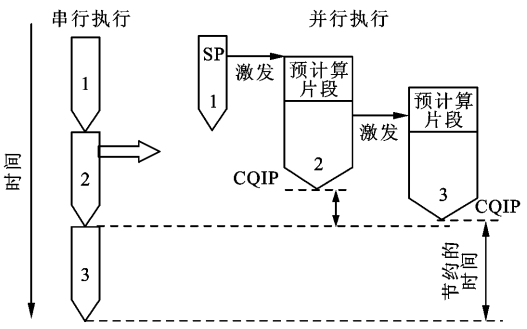

图 1 Prophet 线程模型

已有的 Prophet 模拟器虽然实现了该执行模型，但是更偏重于功能性的模拟，没有研究解决流水线和 SpMT 相结合时可能产生的问题. 此外，已有工作对于过程调用使用的是共享栈结构，而这种结构在多个线程同时发起函数调用时会因为冲突而影响程序的加速比. 现有的工作没有考虑加入 SpMT 技术后动态内存空间管理的问题.

2 SpMT 超标量机制

2.1 推测指令

为了支持 SpMT，我们扩展了 MIPS R3000 ISA[8]，向其中加入了 8 条汇编指令，前 4 条指令均包含一个 26 比特的地址 label. 这些指令的语义如下.

SPAWN 为创建新线程指令，label 是新线程 p_slice 的入口地址. 执行此指令时，父线程要为子线程申请计算资源，将自己的寄存器值拷贝给子线程，子线程跳转到 label 处开始执行.

CQIP 为线程结束指令，label 是线程的入口地址. 确定执行的线程执行到 CQIP 指令要验证子线程的执行是否正确并提交自己的执行结果，推测线程执行到 CQIP 指令转入等待验证状态.

PSLICE_ENTRY 为线程预计算开始指令，label 是线程的入口地址. 该指令会将线程的状态改变为预计算态.

PSLICE_EXIT 为线程预计算结束指令，label 是线程的入口地址. 该指令将线程的状态从预计算态转到推测执行态.

FST 获取栈顶. 该指令将栈加锁，并向 Prophet 模拟器的推测逻辑控制单元请求当前栈顶，sp 保存返回的栈顶值.

UST 释放 FST 指令获取的互斥锁.

GET 获取当前核对应的栈空间的栈顶，即将 DYNAMIC_SP 寄存器的值拷贝到 sp 寄存器. DY-

NAMIC_SP 寄存器是为了实现独立栈而扩展的寄存器.

UPDATE 指令更新当前核对应的栈空间的栈顶并用 rs 寄存器的值更新 DYNAMIC_SP 寄存器的值.

FST 和 UST 指令在共享栈模式使用，GET 和 UPDATE 在独立栈模式使用.

2.2　基本流水线模拟

Prophet+流水线分为 5 个阶段，分别为取指令阶段、发射阶段、执行阶段、写结果阶段和提交阶段，其中 SpMT 机制主要影响的流水段为发射阶段、执行阶段和提交阶段. 我们将着重考虑 2.1 节中前 4 条指令的影响，后 4 条指令的执行情况与普通指令相同. 下面分别介绍这 3 个流水阶段.

(1) 发射阶段. 执行 PSLICE_ENTRY 指令后线程的状态变为预计算态. 预计算态的指令只能访问特定版本的数据，因此译码出 PSLICE_ENTRY 指令后要停止译码，否则由于指令的乱序执行，其后的某条指令将在错误的状态下读取错误版本的数据.

PSLICE_EXIT 指令将线程的状态由预计算态改变为推测执行态. 推测执行态的线程访问的数据总是最新版本的，这与预计算态只能访问特定版本的数据不同. 因此，基于与 PSLICE_ENTRY 同样的原因，译码出 PSLICE_EXIT 时也要停止译码新的指令.

CQIP 指令标志着一个线程的结束，因此译码出该指令时要冲洗流水线，即将指令队列中剩余的指令全部清除，并停止取指和译码工作.

(2) 执行阶段. PSLICE_ENTRY 和 PSLICE_EXIT 指令在该阶段执行完毕后会使译码器重新开始工作，因为此时的机器状态已经稳定. 非推测线程在此阶段执行 CQIP 后如果对子线程的验证通过，则要重置核的状态，并交还计算资源，这时该线程所在的核处于空闲状态，可以在其上发起新的线程；如果子线程验证没有通过，则将核的 PC 指向子线程线程体开始的位置，从该处取指令继续执行. SPAWN 指令在此阶段不执行，我们将在指令提交阶段对此进行讨论.

(3)提交阶段. SPAWN 指令必须在此阶段执行是因为 SPAWN 需要确定的寄存器状态. 只有当 SPAWN 指令到达重定序缓冲区的首部时，SPAWN 前面的指令才全部执行结束，寄存器的值才能最后确定. SPAWN 指令将本线程所有寄存器的值复制给子线程并让子线程开始执行，执行结束后回收 SPAWN 指令占据的重定序缓冲行.

3　内存空间对推测多线程的扩展方法

Prophet+模拟器的 RAM 通过在宿主机上分配的 8 Mb 大小的字符数组来实现，这个大小是可配置的. RAM 的地址范围为 0xa0000000 至 0xa0800000，分为全局数据区、栈区和堆区，图 2 显示了具体的 RAM 布局. 模拟器中通过 VMIPS[9] 模拟器已有的 range 和 mapper 模块来完成模拟器地址空间到物理地址空间的映射.

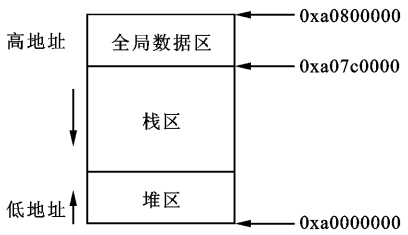

图 2　RAM 地址空间布局

3.1　共享栈

在 Prophet 模拟器共享栈方案中，所有线程共享同一栈空间. 该方案所要解决的关键问题是当多个线程同时请求分配栈空间时的冲突问题. 为此，必须为栈空间设计一个锁，只有持有锁的线程才可以成功分配栈空间. 持有锁的线程在请求栈空间时，必须发消息给所有处理器以获取当前栈空间的栈顶，或者通过访问一个存储全局栈顶的寄存器获取当前栈顶，之后，从该栈顶处为请求线程分配栈空间并更新当前栈顶寄存器(如使用全局栈顶寄存器). 最后，线程释放互斥锁. 为了实现共享栈，我们添加了 2 条汇编指令 FST 和 UST 作为编译器和模拟器之间的接口.

共享栈的优点在于其实现非常简单，只需要在过程调用前将栈空间加锁，进入过程体并分配好栈空间后将栈解锁即可. 但是由于共享栈使用了互斥锁，所以在多个核频繁请求该互斥锁，或者某一个线程在较长的时间内持有互斥锁时，发生冲突的可能性将大大增加，这显然成为了该方案的性能瓶颈.

3.2　独立栈

所谓独立栈是指在模拟器初始化时就为每一个核分配固定大小的一段内存空间，在该核上执行的线程单元发生过程调用时直接在这个核对应的内存空间上分配栈区. 在 Prophet+模拟器中我们采用了通过指针连接堆栈的独立栈方案. 这种方案要解决以下几个关键问题.

(1)由于每个核对应一段独立的内存空间,因此首先要解决的问题是如何将多个核与一段连续的内存空间对应起来.为每个核设置了 2 个额外的寄存器 BP、TOP,BP 记录每个核对应的内存基地址,它是该核上的线程建立活动记录的开始位置;TOP 记录每个核对应的内存地址的最顶端地址,分配活动记录时若 sp 指针的值超过了 TOP 寄存器中记录的值,那么就发生了栈溢出异常,程序转入异常处理.

(2)如果父线程已经提交而其子线程还在运行,那么父线程中的堆栈记录就需要继续保留,因为从串行语义上说,子线程还会继续访问父线程的堆栈,这样,父线程所在的核就会因为它所对应的栈空间不能释放而进入空闲状态.通过为每个核增加一个 DYNAMIC_SP 寄存器,可以很好地解决这个问题.该寄存器保存核所对应的栈空间的动态栈顶,所有在该核上执行的线程在为函数分配活动记录时都以 DYNAMIC_SP 的值为依据.

(3)设计两条汇编指令 GET 和 UPDATE 作为 Prophet 编译器与 Prophet+模拟器之间处理栈管理的接口.

3.3 动态内存空间管理

Prophet+模拟器为在其上运行的应用程序提供了动态内存管理支持.这样做的优点在于模拟器不用模拟特权指令,不用启动操作系统就能够运行应用程序.我们通过 Prophet+内存分配器管理动态内存.在 Prophet+的动态存储空间管理中,要关注的设计问题如下.

(1)确定执行的线程对动态内存的申请和释放与串行程序中相同,因此它对堆空间影响(如可用空间等)是即时生效的.

(2)推测执行的线程申请动态内存时,需要对其申请的内存块进行记录.因为推测线程任何时刻都有可能因为发生各种依赖违规而被其祖先线程撤销,如果让其对堆空间产生的影响即时生效而不记录该申请操作,那么在该线程被撤销时就没有足够的信息恢复该线程对堆空间产生的影响.解决的方法是登记推测线程申请的每一块动态内存空间,如果该推测线程通过了父线程的验证并顺利提交,只需简单地丢弃该线程推测执行时的登记信息;如果推测线程在申请了动态内存后由于依赖违规被撤销,那么需要将所有登记的内存释放给模拟器的动态内存分配器.

(3)类似的问题也出现在推测线程对动态内存的释放上.解决的方法也是相似的,即推测线程释放的动态内存并不真正释放,而是将它们全部记录下来,在推测线程被验证通过并顺利提交时,才通过内存分配器真正释放记录的全部内存块.

推测线程对动态内存的申请与释放的不同之处在于,申请内存时是要真正为线程分配指定大小的内存空间的,且是即时生效的,因为线程很可能马上要对申请的内存块进行操作(如动态申请一个数据结构后初始化该结构体),而释放时不能马上将内存块还给内存分配器,因为一旦该推测线程被撤销,就说明该线程的执行路径可能是错误的,因此存储在内存块中的数据在将来的某个时候还是要被其他线程使用.

4 测试与评价

本文选择 Olden 基准程序测试集对 Prophet+的功能和性能进行测试.选择 Olden 的原因有 3 个:首先,Olden 具有复杂的数据结构以及对这些数据结构的操作,例如都是对树和链表进行合并、遍历等操作;其次,Olden 有复杂的线程间依赖关系,程序结构大多为递归,这种依赖关系便于测试线程划分算法以及执行模式的有效性;最后,很难通过现有的自动并行编译器从 Olden 中提取线程级的并行性.因此,选取 Olden 测试集具有很强的针对性.

4.1 正确性测试

由于目前还没有商用的支持推测多线程的多核处理器,因此在验证 Prophet+的正确性时我们采用将模拟器对程序的执行结果与 GNU GCC 对程序的执行结果进行对比的方法,如果两者的运行结果完全一致,则证明模拟器是正确的.对 Olden 基准程序集中的所有程序进行测试,模拟器执行结果和 GCC 执行结果完全一致.图 3 和图 4 分别是 GCC 和 Prophet 执行 health 基准程序的结果.

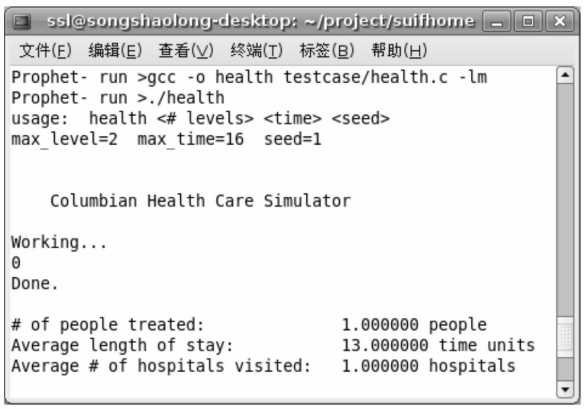

图 3 GCC 执行 health 的结果

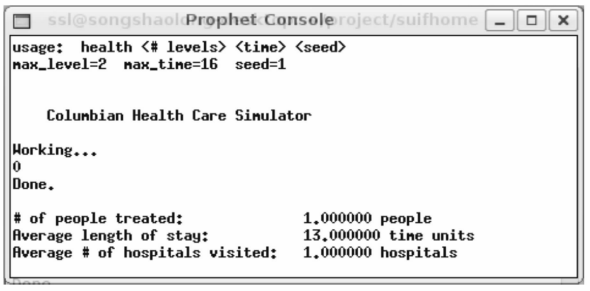

图 4　Prophet 执行 health 的结果

4.2　性能测试

对 Prophet 性能的测试分两个方面：第一个方面是对独立栈性能的测试，测试指标是程序的加速比；第二个方面是对模拟器模拟速度的测试，使用的指标为指令模拟速度和主频模拟速度. 为了得到核的个数对这两个指标的影响，分别对 2 核、4 核和 8 核 3 种配置分别进行了测试.

(1)独立栈性能测试. 在测试栈模型的性能时，我们对独立栈和共享栈的性能做了比较.

图 5 和图 6 是模拟器在 4 核和 8 核时共享栈和独立栈加速比的测试结果对比图. 从图中可以看出，独立栈提高了整个程序的加速比，并且核的个数越多，独立栈的优势越明显. 根据测试数据，4 核时独立栈平均加速比比共享栈高出 1.1%，而在 8 核时独立栈平均要高出 3.9%.

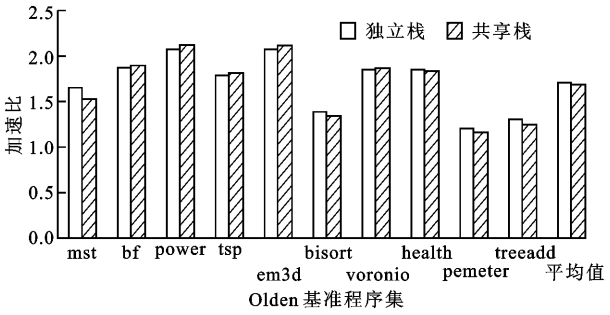

图 5　4 核时独立栈与共享栈加速比对比

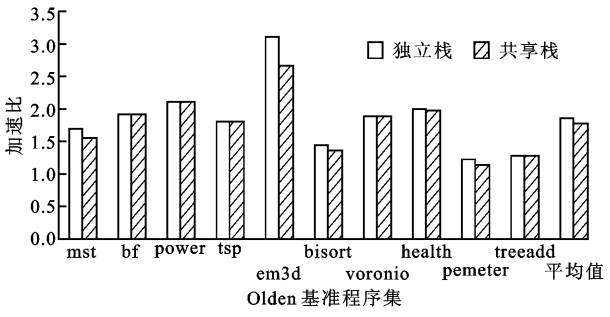

图 6　8 核时独立栈与共享栈加速比对比

(2)图 7 和图 8 是 Prophet+模拟器在不同核数量时的主频模拟速度和指令模拟速度对比.

由图 7 可见，主频模拟速度主要与模拟器模拟的体系结构有关. 当核的个数增多时主频模拟速度呈下降趋势，并且下降的速度随着核个数的增多逐渐变缓慢. 最快的主频模拟速度出现在两核的 Power 程序上，速度达到了 38×10^3 次 · s^{-1}.

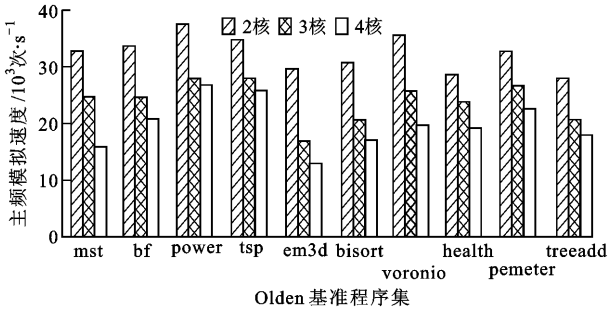

图 7　不同核数的主频模拟速度对比

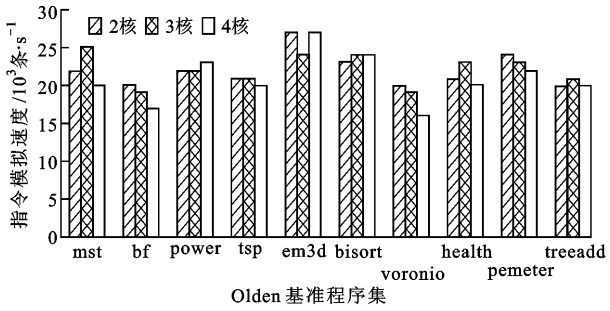

图 8　不同核数的指令模拟速度对比

由图 8 可见，指令模拟速度对核的个数不是很敏感，但是不同程序上的指令模拟速度存在较大的差异. 因此，指令模拟速度主要与模拟器模拟的应用程序有关. 最高的指令模拟速度出现在 2 核与 8 核的 em3d 程序上，达到 27×10^3 条 · s^{-1}. 这是因为 em3d 程序指令之间的数据依赖比较小，因此指令的模拟速度相对比较快. 从这个意义上来说，指令模拟速度实际上与应用程序的 IPC 密切相关，即程序的 IPC 越高，指令模拟速度越快.

5　结　论

本文提出了一种支持 SpMT 的超标量流水线结构和一种基于独立栈的运行时内存空间管理方法，得到的结论如下.

分析设计了加入 SpMT 后超标量流水线，实验结果证明这种方法是有效的. 独立栈方法能够克服共享栈模型中由加锁机制导致的冲突数目与核数目正相关问题，核的个数越多，独立栈提高程序效率的优势越明显. 将线程申请或释放动态内存的动作延迟到该线程拥有确定执行权限时完成，保证了动态内存管理的正确性，并提高了效率. 主频模拟速度主要由目标机的体系结构决定，而指令模拟速度主要与目标程序相关.

参考文献：

[1] BRIAN A, RUDOLF E. Application of automatic parallelization to modern challenges of scientific computing industries[C]// Proceedings of the 37th International Conference on Parallel Processing. Piscataway, NJ, USA: IEEE, 2008:279-286.

[2] ARMSTRONG B, EIGENMANN R. Challenges in the automatic parallelization of large-scale computational applications[C]// Proceedings of SPIE/ITCOM 2001. Bellingham, WA, USA: SPIE, 2001:50-60.

[3] TIAN C, FENG M, NAGARAJAN V, et al. Copy or discard execution model for speculative parallelization on multicores[C]. Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture. Piscataway, NJ, USA: IEEE, 2008, 330-341.

[4] OHSAWA T, TAKAGI M, KAWAHARA S, et al. Pinot: speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities[C]// Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. Piscataway, NJ, USA: IEEE, 2005, 81-92.

[5] MADRILES C, QUINONES C, SANCHEZ J, et al. Mitosis: a speculative multithreaded processor based on precomputation slices [J]. IEEE Transactions on Parallel and Distributed Systems, 2008, 19(7): 914-925.

[6] DONG Zhaoyu, ZHAO Yinliang, WEI Yuanke, et al. Prophet: a speculative multi-threading execution model with architectural support based on CMP[C]// Proceedings of the 2009 International Conference on Scalable Computing and Communications. Piscataway, NJ, USA: IEEE, 2009, 103-108.

[7] ZIER D, LEE B. Performance evaluation of dynamic speculative multithreading with the cascadia architecture[J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(1): 47-59.

[8] SWEETMAN D. See MIPS run[M]. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2007.

[9] GAEKE B R. VMIPS Project [EB/OL]. [2008-01-24]. http://www.dgate.org/VMIPS.

（编辑 武红江）