第34卷 第11期 2011年11月
计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS
Vol. 34 No. 11 Nov. 2011

程序执行的精确重现技术及其在体系结构模拟中的应用

赵天磊 唐遇星 徐炜遐 付桂涛 齐树波 贾小敏 张民选

(国防科学技术大学计算机学院 长沙 410073)

摘 要 近年来有研究提出利用动态二进制翻译技术(Dynamic Binary Translation,DBT)加速程序代表性模拟点的提取,然而这些研究并未考虑DBT方法对模拟结果准确度的影响.实验发现,对于某些程序,DBT加速方法会带来将近20%的误差.经分析,误差的根源在于程序在DBT执行和模拟执行时执行踪迹有巨大差异,即程序执行踪迹的不可重现性.本文系统地分析了引起程序执行踪迹不可重现的原因,提出了解决方法.实验证实,利用文中提出的方法,可以在不影响性能的情况下,实现程序执行踪迹的精确重现,从而保证DBT方法与传统模拟方法具有完全相同的精确度.

关键词 应用程序行为分析;模拟误差;可重现模拟;执行踪迹;二进制翻译;SimPoint;BBV Profile

中图法分类号 TP314 **DOI号**: 10.3724/SP.J.1016.2011.02073

Exactly Reproducible Program Execution and Its Application in Computer Architecture Simulation

ZHAO Tian-Lei TANG Yu-Xing XU Wei-Xia FU Gui-Tao QI Shu-Bo JIA Xiao-Min ZHANG Min-Xuan

(*School of Computer, National University of Defense Technology, Changsha* 410073)

Abstract Recently, dynamic binary translation (DBT) technique has been proposed to accelerate the extraction of representative simulation points of programs. However, the accuracy implications of the DBT method have not been studied. It is observed that for some programs, the DBT method can incur a nearly 20% error in the simulation results. Careful analysis shows that the error is caused by the mismatch between the execution traces of programs under DBT environment and simulation environment. The cause of the execution trace mismatch is studied carefully. Several methods are proposed to overcome the problem. Experimental results show that with the proposed methods, the execution traces of programs can be matched accurately between different execution environments. Therefore, the accuracy of the DBT method can be guaranteed.

Keywords program behavior analysis; simulation error; reproducible simulation; execution trace; binary translation; SimPoint; BBV profile

1 引 言

周期精确的性能模拟是微处理器体系结构研究的一种重要方法[1],然而模拟器的运行速度非常慢,功能模拟的速度通常只有10MIPS左右,性能模拟的速度只有几百KIPS[2].为了提高模拟速度,研究人员提出了SimPoint方法来提取程序的代表性模

收稿日期:2011-08-29;最终修改稿收到日期:2011-09-19.本课题得到国家自然科学基金(60970036)、教育部博士点基金(20094307120007)资助.**赵天磊**,男,1982年生,博士研究生,主要研究方向为多核处理器存储系统、体系结构模拟技术.E-mail: tlzhao@nudt.edu.cn. **唐遇星**,男,1977年生,博士,副研究员,主要研究方向为微处理器设计与验证、二进制翻译. **徐炜遐**,男,1963年生,硕士,研究员,主要研究领域为巨型计算机系统结构等. **付桂涛**,男,1980年生,博士研究生,主要研究方向为Cache一致性协议. **齐树波**,男,1982年生,博士研究生,主要研究方向为片上互连网络. **贾小敏**,女,1982年生,博士,主要研究方向为高性能片上存储系统. **张民选**,男,1954年生,教授,博士生导师,主要研究领域为高性能微处理器体系结构、超大规模集成电路设计、巨型计算机系统结构等.

拟点，只需对提取的代表性模拟点进行详细模拟就可以获取程序的精确性能参数[1,3-4]. 然而在这种模拟方法中，生成 SimPoint 所需的 BBV（Basic Block Vector）Profile 需要利用功能模拟器将测试程序预先完整执行一遍. 由于功能模拟的速度非常慢，BBV Profile 的提取成了一个新的性能瓶颈. 近年来有研究者提出利用动态二进制翻译（Dynamic Binary Translation，DBT）[5-6] 技术提取应用程序的 BBV Profile[7-9]. 这种方法可以大大加速代表性模拟点的提取，减少整个模拟过程所需的时间，但现有研究都没有评估其对模拟结果准确性的影响.

本文的主要工作有两项. 第一项是评估了 DBT 方法对模拟结果准确度的影响. 通过实验发现 DBT 方法会较为严重的损害模拟结果的准确度，对部分测试程序会带来近 20% 的模拟误差. 经分析，这种误差的根源是程序在 DBT 执行和模拟执行时指令踪迹有巨大差异，即模拟时无法精确重现 DBT 执行时程序的执行踪迹. 进一步分析发现，程序的执行踪迹差异是由不同环境中实验设置上的细微差异引起的. 这与文献[10]的结论类似. 文献[10]分析了体系领域研究中的测量偏倚问题，指出设置实验环境时，看似无关的细微差异都会对实验结果产生重大影响，甚至会改变实验的结论. 然而该文献并没有系统的分析哪些因素可能会对实验结果造成较大影响.

本文的第二项工作是提高 DBT 方法的准确度. 本文首先提出了一个进程执行踪迹模型，根据此模型，从进程初始状态和状态更新过程两方面入手，给出了一种系统方法来确定所有可能影响程序行为的环境因素. 然后针对每种影响因素，分别提出了相应的方法来消除其对执行踪迹的影响. 实验结果证实，文中提出的方法可以有效地消除程序在不同执行环境下的指令踪迹差异，实现程序执行踪迹的精确重现，从而保证 DBT 方法与传统模拟方法具有完全相同的准确性. 本文提出的方法也可用于程序调错等需要精确重现程序执行踪迹的应用领域，或用于消除文献[10]中提出的测量偏倚问题.

本文第 2 节阐述现有方法中存在的问题，分析了 DBT 方法对模拟结果准确性的影响及其内在原因；第 3 节提出解决问题的方法，首先提出了一个进程执行踪迹模型，然后确定了影响进程执行踪迹的可能因素及其规避方法；第 4 节评估所提出的方法；第 5 节总结全文.

2 DBT 方法对模拟准确度的影响

2.1 基于 DBT 加速的模拟流程

基于 DBT 加速的微处理器体系结构性能模拟是利用 DBT 技术代替功能模拟器（ISA Simulator）来收集 SimPoint 方法所需的 BBV Profile，是对 SimPoint 方法的加速. 其流程如图 1 所示，可以分为 3 步：

1. 利用基于功能模拟器或基于 DBT 技术的 BBV Profiler 将测试程序完整执行一遍，在执行过程中收集测试程序的 BBV Profile. BBV Profile 是对程序执行过程的一种表示，记录了每个时间片段内程序所执行的基本块集合；

2. 利用 SimPoint 工具对收集的 BBV Profile 进行分析，提取出一些执行片段. 这些提取出的执行片段代表了程序整个执行过程的行为特征，称为代表性模拟点；

3. 利用性能模拟器（Performance Simulator）对测试程序进行模拟，模拟的过程中只对代表性模拟点进行详细的性能模拟，其它部分使用功能模拟快速跳过. 这一步的关键是准确地定位到第 2 步中所识别出的代表性模拟点.

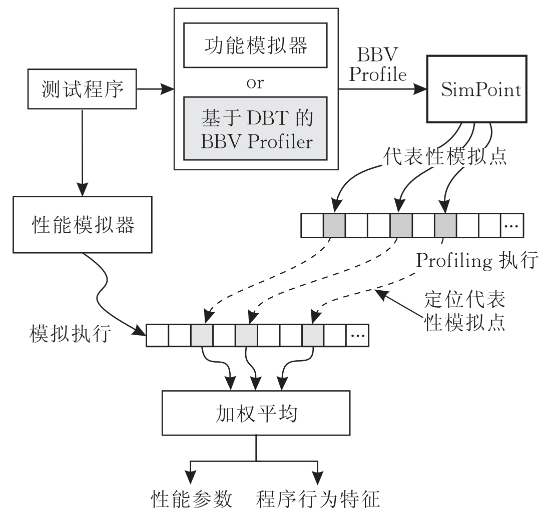


图 1 利用动态二进制翻译技术和 SimPoint 技术加速的模拟流程. 代表性模拟点在模拟执行过程中的准确定位对模拟结果的准确性有重要影响

收集 BBV Profile 时需要将测试程序预先完整执行一遍. 传统上 BBV Profile 收集是利用功能模拟器实现的，然而功能模拟器的速度非常慢，是整个模拟过程的一个重要瓶颈. 对于像 SPEC2006 这样的现代测试程序集，如果使用功能模拟器完整执行的话，将需要数月时间. 基于 DBT 的 BBV Profiler 速度可以达到功能模拟器的 10 倍左右[11]，因此可以大大缩短收集 BBV Profile 所需的时间.

从图 1 中可以看出，在 DBT 加速的模拟流程

中，测试程序一共被执行了两次. 第 1 次是在基于 DBT 的 BBV Profiler 中执行的，用于收集 BBV Profile. 第 2 次是在性能模拟器上执行的，用于得到程序的性能参数或行为特征. 在第 2 次执行过程中，需要准确定位到由第 1 次执行提取出的代表性模拟点. 因此，这种 DBT 加速的模拟流程隐含了一个前提，即程序在两次执行时具有相同或非常接近的执行踪迹.

这个前提成立与否对模拟结果的准确性具有重要影响. 代表性模拟点是根据程序第 1 次执行时的行为特征提取出来的. 如果两次执行时的行为不同，模拟点就不能代表程序在第 2 次执行时的行为特征. 因此利用这些模拟点得出的模拟结果就不能准确反映第 2 次执行时的性能特性. 反过来，由于模拟结果得自程序的第 2 次执行，所以也不能准确反映第 1 次执行时的性能特性. 要对代表性模拟点进行详细模拟，就必须能够在第 2 次模拟执行时准确定位这些模拟点. 如果程序两次执行时的执行踪迹有较大差异的话，就难以在模拟时准确的定位到模拟点，从而导致模拟结果误差.

虽然这个前提对模拟结果的准确性有非常大的影响，然而现有研究都没有对此前提的有效性进行验证. 本文通过实验发现，对于很多程序，基于 DBT 的执行和基于模拟器的执行会有较大的执行踪迹差异，且这种执行踪迹差异会对模拟结果的准确性造成较大影响. 因此在使用 DBT 加速时必须设法消除程序两次执行时的执行踪迹差异.

2.2　模拟点的定位方法

定位模拟点的方法有两种，第 1 种基于指令计数，即利用程序从启动至到达模拟点之间所提交的指令总数定位模拟点，记这种方法为 icount 定位方法. 这种方法要求程序在两次执行中的指令总数差异不能太大，必须远小于模拟点的大小（模拟点大小一般处于 10 M～100 M 条指令之间[3-4,12]）. 否则就会导致基于指令计数的方法在定位模拟点时出现很大的偏差，从而影响模拟结果的准确性.

第 2 种方法利用模拟点边界指令的 PC 以及该指令的执行次数这两个信息来确定模拟点，记这种方法为 marker 定位方法. 这种方法亦要求程序在两次执行间的指令踪迹不能相差太大. 若模拟点边界指令的执行次数在两次执行间有较大差异的话，就会导致模拟点定位出现偏差. 尤为严重的是若第 2 次执行时相应边界指令没有被执行或执行次数不够，就会导致无法定位模拟点. 另外，由于事先并不知道模拟点边界指令的 PC，所以这种方法事实上要求在 BBV Profiling 过程中对所有指令的执行次数都进行统计. 这会给基于 DBT 的 BBV Profiler 带来超过 30% 的性能下降[11]，从而部分抵消了 DBT 方法的加速效果.

2.3　DBT 方法对模拟结果准确度的影响

为了评估 DBT 方法对模拟结果准确度的影响，本文利用 Simics[13] 模拟器和基于 DBT 的 BBV Profile 提取工具，QPoint[9]，分别执行了 SPEC2006 测试程序，并对提取的模拟点进行了模拟. 本节给出了 DBT 方法和功能模拟方法在得到的指令总数、整体性能参数和程序行为特征等方面的差异.

表 1 给出了所有 SPEC2006 测试程序①在 QPoint 上和模拟器 Simics 上分别执行时的动态指令总数. 从表中可以看出，有将近 1/3 (9/28) 的测试程序在两次执行间的指令总数相差超过 100 M 条指令，超过 40% (12/28) 的测试程序在两次执行间的指令总数差异超过 10 M. 由于典型的模拟点大小为 10 M～100 M 条指令，因此这种程度的指令偏差会导致

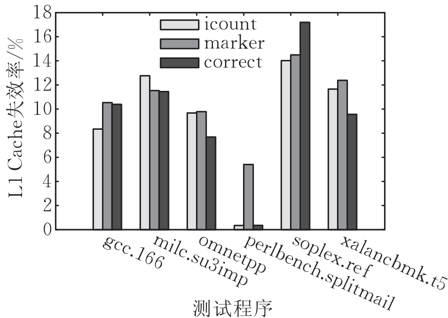
表 1　SPEC2006 测试程序在基于 DBT 的 BBV Profiling 执行时和模拟执行时的动态指令总数及偏差（所有的测试程序执行时均使用 reference 输入集）

测试程序	动态指令总数		
	Simics	QPoint	偏差(Δ)
400.perlbench	762816940717	763456297927	639357210
401.bzip2	535110832014	535110834361	2347
403.gcc	83792871309	84764827875	971956566
410.bwaves	4109187803249	4109188163883	360634
416.gamess	4477015763260	4477015771387	8127
429.mcf	439350891791	439351002443	110652
433.milc	1162698380427	1162829752454	131372027
434.zeusmp	2315700510347	2315700517520	7173
435.gromacs	1754307479669	1754307606233	126564
436.cactusADM	3692374503694	3692376718040	2214346
437.leslie3d	2599825931830	2599826865604	933774
444.namd	3800239248763	3800239368396	119633
445.gobmk	688102677975	688102680766	2791
447.dealII	2069387009051	2069306024811	80984240
450.soplex	448971126111	449160697736	189571625
453.povray	1023653968839	1023652972016	996823
454.calculix	8253521155666	8253934643401	413487735
456.hmmer	1184883498847	1184885134629	1635782
458.sjeng	2954760643001	2954760645136	2135
459.GemsFDTD	2106760402142	2106788414210	28012068
462.libquantum	3340142177868	3340170307670	28129802
464.h264ref	382869988950	382869983200	5750
465.tonto	3759189446470	3758683629473	505816997
470.lbm	1678909037991	1678909084138	46147
471.omnetpp	631607032909	631402465700	204567209
473.astar	882232681082	882232682177	1095
482.sphinx3	3539853733851	3540026244521	172510670
483.xalancbmk	1309798089981	1306095154135	3702935846

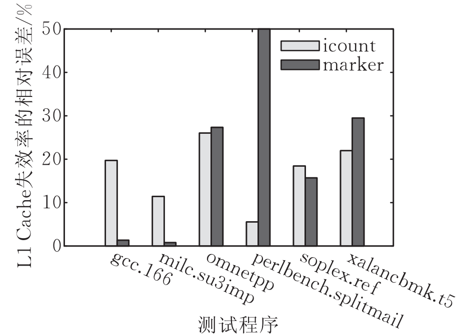
① 481.wrf 执行过程中出错，因此本文没有测试此程序

icount 定位方法无法准确定位代表性模拟点；同时指令总数的差异也反映了程序执行踪迹上的差异，因此 marker 定位方法也很可能失效.

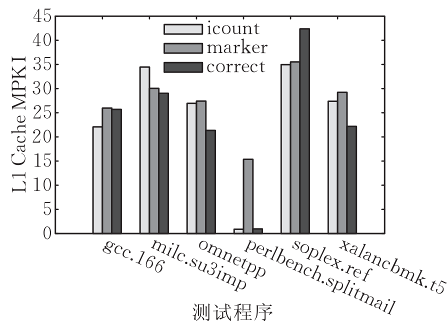
为了评估这些指令偏差对模拟结果的影响，本文分别对 QPoint 和 Simics 提取出的代表性模拟点进行了详细的性能模拟. 对 Simics 生成的模拟点使用 icount 方法定位，对 QPoint 生成的模拟点在模拟时分别使用了 icount 方法和 marker 方法进行定位. 由于模拟全部 SPEC2006 测试程序所需的时间过长，本文选取了总指令数误差大于 100M、且运行时间相对较短的 6 个测试程序：400.perlbench，403.gcc，433.milc，450.soplex，471.omnetpp，483.xalancbmk. 用于比较的参数共有 4 个，分别是 L1 Cache 和 L2 Cache 的失效率和 MPKI（Misses Per Kilo Instructions）. 图 2 给出了 6 个测试程序在分别利用 QPoint 和 Simics 提取出的模拟点时得到的



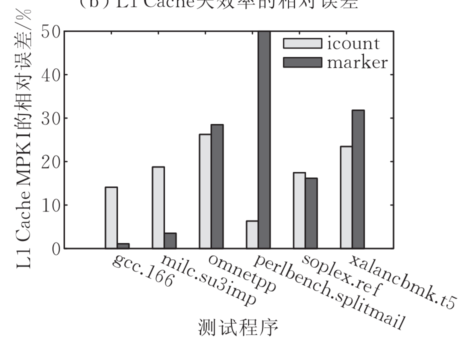
(a) L1 Cache失效率



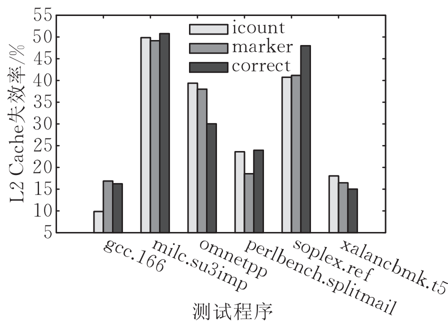
(b) L1 Cache失效率的相对误差



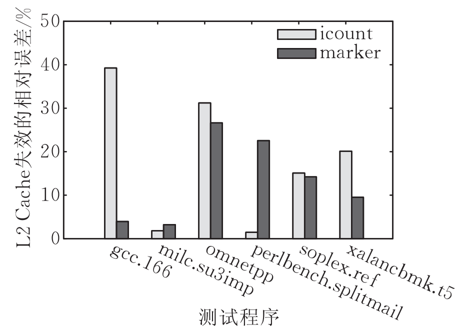
(c) L1 Cache MPKI



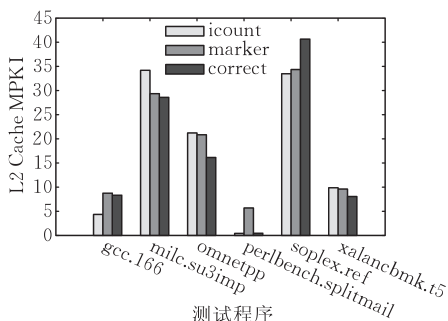
(d) L1 Cache MPK的相对误差



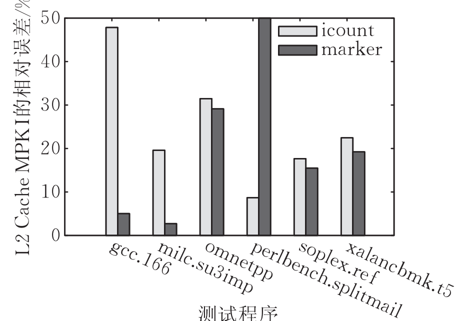
(e) L2 Cache失效率



(f) L2 Cache失效率的相对误差



(g) L2 Cache MPKI



(h) L2 Cache MPKI的相对误差

图 2 对于几个指令数差异较大的 SPEC2006 测试程序，DBT 方法和功能模拟方法在得到的 L1 Cache 失效率、MKPI，L2 Cache 失效率、MPKI 参数上的绝对值和相对误差. DBT 方法使用了 icount 和 marker 两种不同的模拟点定位方法

L1 Cache 失效率、MPKI，L2 Cache 失效率以及 MPKI 的模拟结果及相对误差. 其中 icount 标记的是利用 icount 方法定位 QPoint 提取的模拟点得到的模拟结果；marker 标记的是利用 marker 方法定位 QPoint 提取的模拟点得到的模拟结果；correct 标记的是由 Simics 提取的模拟点所对应的模拟结果. 从实验结果可以看出，不管是采用 icount 定位方法还是 marker 定位方法，相对于功能模拟器提取的模拟点，由 DBT 方法提取的模拟点得到的模拟结果都有较大的偏差. 使用 icount 定位方法时，对 L1 失效率、L1 MPKI、L2 失效率、L2 MPKI 四个指标，QPoint 相对于 Simics 的平均相对误差分别为 17%、17%、18%、24%；使用 marker 定位方法时，平均相对误差分别为 2.45x、2.66x、13%、2.01x.

SimPoint 方法提取的代表性模拟点不仅可用于分析程序的整体性能参数，还可以用于分析程序的典型行为特征. 代表性模拟点反映了程序执行过程中的主导执行阶段，程序在这些执行阶段的行为就代表了程序的典型行为特征. 因此，模拟点的定位误差不光会造成程序性能参数的估计误差，还会导致程序典型行为特征分析上出现重大偏差. 本文评估了 403.gcc 和 450.soplex 两个程序在各代表性模拟点处的 L2 MRC（Miss Ratio Curve）特性，即当 Cache 容量增加时失效率的变化曲线. L2 MRC 反映了程序的 L2 行为特性，程序在所有代表性模拟点处的 MRC 曲线就反映了程序的典型 L2 行为特性. 3 个模拟点集合，icount，marker 和 correct 分别代表不同方法获取的模拟点. icount 表示用 icount 定位方法得到的模拟点集合，marker 表示用 marker 定位方法得到的模拟点集合，correct 表示用功能模拟得到的模拟点集合. 图 3 给出了 403.gcc 和 450.soplex 程序在所有模拟点集合的 L2 MRC 曲线. 从其中可以看出，不管是 icount 定位方法还是 marker 定位方法，在提取程序的典型行为特征时，都存在丢失正确特征和多出错误特征的问题.

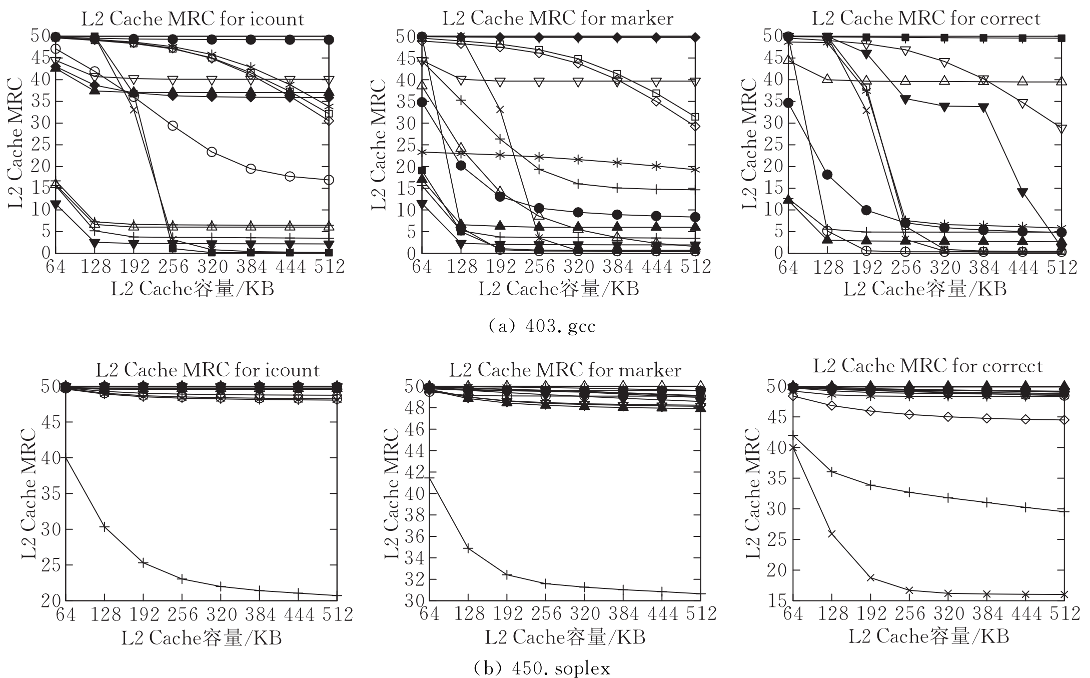


图 3　两个测试程序 403.gcc 和 450.soplex 在不同的模拟方法下所得到的 L2 Cache MRC 曲线. 每条曲线表示一个模拟点处的 L2 MRC 特征，所有曲线的集合代表了一个测试程序的 L2 总体访问特征

3　提高 DBT 方法的准确度

根据第 2 节的分析，导致 DBT 方法有较大误差的根本原因是代表性模拟点的定位偏差. 不管是 icount 定位方法还是 marker 定位方法，其模拟点定位精度都取决于程序在基于 DBT 的 BBV Profiling 执行时和模拟执行时指令踪迹的差异程度. 因此提高 DBT 方法准确度的关键是保证程序在两次执行时具有完全相同或相近的指令踪迹. 本节对进程的执行过程进行分析，以确定所有可能影响进程执行踪迹的因素，并提出相应的规避方法.

3.1 影响程序执行踪迹的可能因素

理想情况下，对于相同的程序和相同的输入，在基于 DBT 的 BBV Profiler 和功能模拟器两种环境下执行时，指令踪迹应该是相同的. 然而实验显示，程序的指令踪迹在两种环境下有较大差异，表现为总提交指令数上的巨大差异，见表 1. 为了分析进程执行踪迹偏差的来源，下面对进程的执行过程进行深入分析.

程序的执行踪迹取决于执行过程中分支指令的取向，而分支指令的取向又决定于分支指令执行时进程的状态，因此程序的执行踪迹最终取决于进程的状态变迁. 一个程序的执行过程是按照程序中的指令对进程的状态进行更新的过程，可以表示为式(1)所示的一个序列.

$$S_0 \xrightarrow{I_0} S_1 \xrightarrow{I_1} S_2 \xrightarrow{I_2} \cdots S_k \xrightarrow{I_k} \cdots S_n \qquad (1)$$

其中 S_k 表示进程的状态，S_0 表示进程被创建后的初始状态；I_k 表示进程所执行的指令序列. 指令是对进程状态的更新，可以表示成一个映射 I：$S \times E \rightarrow S$，其中 E 表示进程的执行环境，比如系统时间、处理器速度、操作系统版本等. 由于某些指令的行为受执行环境的影响，因此指令的语义中必须包含环境部分. 对于行为不受执行环境影响的指令，可以简单地表示为 I：$S \rightarrow S$.

进程的状态可以分为两类，局部状态和全局状态. 局部状态指可见范围很小或生存时间很短的状态，比如临时变量、ABI(Application Binary Interface)中定义的 Scratch 寄存器等. 全局状态指可见范围很大、生存时间很长的状态，比如全局变量、程序的内存分配器状态、PC/SP 等起控制作用的寄存器等. 局部状态由于时空作用范围很小，通常不会对程序的执行踪迹造成大的影响，而全局状态则会持续的影响程序的执行踪迹，因此可能会有较大影响.

虽然从理论上说，只要进程的全局状态在两次执行间保持一致，就可以保证进程的执行踪迹基本一致. 然而在实际模拟过程中，区分进程的局部状态和全局状态是一件非常困难的事情，并且有些情况下局部状态也会影响到全局状态. 因此本文采用了一种较为保守的方法，即要求进程在两次执行过程中的所有状态都保持一致，以保证进程的执行踪迹完全相同. 由式(1)可以看出，如果进程在两次执行时的初始状态相同，并且执行过程中每条指令对进程状态的更新相同，就可以保证进程的状态始终保持一致.

本节从进程初始状态和状态更新过程两个方面，分析了同一个程序分别在 QPoint 和 Simics 两种环境下执行时，有哪些因素可能会导致进程状态出现差异以及这些状态差异如何影响进程的指令踪迹，并提出了避免产生这些差异的方法.

3.2 进程初始状态

进程的初始状态是由执行环境的应用程序加载器(loader)设定的，可以分为寄存器初始状态和内存初始状态两部分. 真实的操作系统环境中，加载器位于操作系统内核中. 在用户级模拟器和二进制翻译器中，由于没有操作系统，所以一般都会实现一个自有的加载器. QPoint 是一个用户级的二进制翻译器，因此加载程序时使用的是自有的加载器. 而 Simics 是一个全系统模拟器，上面运行有一个完整的操作系统，因此使用的是操作系统内核的加载器. 本节以 QPoint 和 Linux 内核为例，分析了 SPARC 体系结构下不同加载器对进程初始状态的影响以及这些差异对进程执行踪迹的影响.

3.2.1 寄存器初始状态

根据 SPARC ABI 规范[14]，进程被创建之后，PC 寄存器指向进程的入口地址，SP 寄存器指向栈顶，其它寄存器中是预定义的常量. 因为可执行程序是不可重定位的，在任何执行环境中都会被加载到固定地址，因此程序的入口地址是固定的，即 PC 寄存器初值没有差异.

然而，QPoint 和 Simics 两个环境中，进程的 SP 寄存器初值并不相同. 有两个原因，第 1 个原因是堆栈区域的位置不同. 因为 SPARC ABI 中没有规定堆栈区域的位置，所以不同的加载器可能分配不同的存储区域作为堆栈区. 为了提高安全性，Linux 内核会对堆栈区域的位置进行随机化处理，导致程序每次运行时，堆栈的位置都不相同. 第 2 个原因是堆栈中存储的初始数据量不同，因此即使堆栈区域的位置相同，SP 寄存器的初值也可能不同.

SP 寄存器值的差异会影响所有在堆栈中分配的对象的地址，有些程序会使用对象的地址参与计算，甚至利用计算结果控制程序的执行流程. 因此，SP 寄存器的差异会导致进程的执行踪迹出现差异.

消除 SP 寄存器初值差异的办法包括两点：第一是禁用 Linux 内核的地址空间随机化机制①，保证每次运行时堆栈起始位置固定. 第二是修改

① 可以通过向/proc/sys/kernel/randomize_va_space 文件写入 0 实现

QPoint 所创建的堆栈区域的位置，保证与 Linux 内核所分配的堆栈位置相同. 通过这两个方法，就可以保证 SP 寄存器的初值相同，从而保证两种执行环境下进程的寄存器初始状态完全相同.

3.2.2　内存初始状态

Linux 系统中，一个进程的内存映像如图 4 所示. 由于可执行程序是不可重定位的，所以.text/.data/.bss/heap 段的位置和内容都是固定的. 由于使用的是静态链接的程序，所以 mmap 区域初始内容为空. 因此内存初始状态中可能存在差异的只有栈区域.

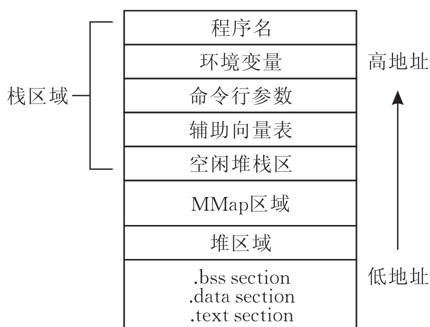


图 4　Linux 系统中进程的内存布局

栈区域的内容包括程序执行时的命令行、环境变量、加载器所设置的辅助向量表等参数. 这些参数的差异会在两个方面引起进程执行流程的差异. 一方面，会直接影响到读取这些参数的函数的执行踪迹，比如环境变量的个数和顺序会影响 getenv 库函数的指令踪迹，而辅助向量的个数和顺序会影响 C 运行库在初始化过程中的行为. 另一方面，这些参数的不同会引起 SP 寄存器初值的差异，继而影响堆栈中分配的对象的地址，最终影响程序的执行踪迹.

通过将堆栈区的初始内容设置为完全相同，就可以消除内存初始状态的差异，方法是在配置运行环境时，进行如下的设置：

(1) 保证程序运行环境中环境变量的个数、取值和排列次序相同；

(2) 修改 QPoint 中的加载器，使之提供与 Linux 内核完全相同的辅助向量表；

(3) 运行程序时使用完全相同的命令行，这意味着不能在测试程序的命令行上调用 QPoint，而必须使用操作系统的 binfmt_misc 机制来自动调用 QPoint.

3.3　状态更新过程

即使两个进程的初始状态完全相同，其执行踪迹也可能因所执行的指令而产生差异. 可以将进程所执行的指令分为两类. 一类指令在每次执行时对进程状态的更新都是确定的，不依赖于进程的外部执行环境，用户级指令集中的大部分指令都属于此类. 另一类指令对进程状态的更新依赖于进程的外部执行环境. 此类指令主要用于申请资源或与外界交互，包括系统调用指令、引发异常的指令、访问外设的指令、访问硬件计数器的指令等. 因为第 1 类指令对进程状态的更新是确定的，因此不会给程序的执行踪迹带来差异. 第 2 类指令执行时会受到外部环境的影响，因此在不同的执行环境下，可能会对程序的执行踪迹有不同的影响. 本小节详细分析了影响第 2 类指令的外部环境因素，并提出了避免这些因素影响的方法.

3.3.1　操作系统内核的影响

对于运行在 QPoint 上的进程来说，QPoint 虚拟了一个操作系统内核，用于处理进程的所有系统调用请求和触发的异常事件. 而对运行在 Simics 上的进程来说，其系统调用请求和异常事件是由真实的 Linux 内核处理的. QPoint 虚拟出的内核和真实 Linux 内核之间的行为差异会导致在两个环境下执行的进程的状态出现差异.

虚拟地址空间布局. Linux 系统中，进程请求的大块动态内存都被分配在 mmap 空间，如果 mmap 地址空间布局不同的话，进程中很多动态对象的地址就会存在差异. 对于使用对象的地址参与计算、并利用计算结果控制程序执行流程的应用程序来说，这种对象地址的差异可能会引起执行踪迹上的巨大差异. 另外，由于虚拟地址空间布局是一个全局状态，会影响后续的整个执行过程，因此消除虚拟地址空间布局差异对保证执行踪迹的一致性具有重要意义.

QPoint 和 Simics 中进程虚拟地址空间布局上的差异可能由两个因素引起：

(1) QPoint 和 Linux 内核中实现的 mmap 空间分配算法不同. QPoint 中从低地址向高地址进行空间分配，而 Linux 内核从高地址向低地址进行空间分配.

(2) Linux 内核中不仅对栈区域进行随机化处理，也会对 mmap 区域进行随机化处理，因此每次执行时 mmap 空间的起始地址会有一个随机的偏移，进而导致进程的虚拟地址空间布局在每次运行时都不相同.

针对以上两点，分别采用如下方法就可以保证 QPoint 和 Simics 中进程的虚拟地址空间布局完全一致. (1) 修改 QPoint 中的 mmap 空间分配算法，

采用自顶向下的方法来分配 mmap 空间；(2)禁用 Linux 内核的虚拟地址空间随机化机制，方法与禁用栈区域随机化相同.

I/O 缓冲块大小. 库函数在进行文件 I/O 时会对 I/O 数据进行缓冲，只有当一个缓冲块被写满或读完之后，才会进行真正的磁盘 I/O. I/O 缓冲块的大小是与执行环境相关的. 如果不同执行环境下的缓冲块大小不同，一方面会引起库函数中 I/O 缓冲部分的执行流程差异；另一方面，由于缓冲块是从系统堆内存中分配的，不同的块大小会导致堆内存分配算法的状态差异. 堆内存状态是一个全局状态，因此这种差异会影响到后续的所有堆内存分配释放操作中，导致指令踪迹出现巨大差异.

在 QPoint 环境中，进程的 I/O 缓冲块大小是由 QPoint 设定的. 而 Simics 环境中，进程的 I/O 缓冲块大小是由 Linux 内核设定的，这两者可能存在差异. 消除 I/O 缓冲块大小差异的方法是修改 QPoint 中的 Stat 系统调用模拟机制，使其返回与 Linux 内核相同的 I/O 缓冲块大小.

可读的进程信息. 一些可读的进程信息比如进程 ID、用户 ID 等，是依赖于进程的执行环境的. 如果进程的行为受这些信息影响，就会导致不同环境下指令踪迹的差异. 在设置实验环境时，通过在不同的执行环境中使用相同的用户 ID，可以消除用户 ID 的差异. 通过在 QPoint 中截获 getpid 等获取进程信息的系统调用，可以保证在不同的执行环境下具有一致的进程信息.

3.3.2 寄存器窗口溢出的影响

SPARC 体系结构中的寄存器窗口机制会引起进程指令踪迹的内在不确定性. SPARC ABI 中规定操作系统内核和用户进程共享寄存器窗口资源. 如果在系统调用或异常事件的处理过程中发生寄存器窗口溢出，溢出的寄存器窗口就会被保存到进程的用户堆栈中. 这会导致进程堆栈的内容发生意外的改变，是系统调用或异常指令的副作用. 在 QPoint 中，由于 OS 是虚拟出来的，不和用户进程共享寄存器资源，因此虚拟 OS 中不会将寄存器窗口溢出到用户进程的堆栈，也就不会引起进程用户堆栈的状态改变. 因此，处理完系统调用或异常事件后，QPoint 和 Simics 中进程的堆栈状态可能存在差异.

某些程序的执行流程会受到用户堆栈的这种状态差异的影响，从而导致进程指令踪迹的差异. 由于并不是每次进入内核都会改变进程用户堆栈的内容，而是取决于进入内核之后是否发生了寄存器窗口溢出，因此这种情况无法被准确识别，没有一种简单的办法避免内核代码中的寄存器窗口溢出对进程指令踪迹的影响.

3.3.3 系统时间的影响

大部分使用了系统时间的程序的执行流程都会受系统当前时间的影响. 因此如果不同运行环境中的系统时间不同的话，就会导致此类程序的指令踪迹出现变化. 即使一个程序的计算逻辑没有明显地依赖于系统时间，但是只要程序试图获取系统时间，就可能引起指令踪迹的变动. 原因是由于系统中通常只维护一个秒时间和微秒时间，获取其它形式的时间时就必须进行转换. 转换过程的指令踪迹受 3 个因素的影响：

(1)系统当前日期. 转换秒时间到日期时有一个循环相减的过程，如果系统日期不同，循环次数就会不同；

(2)系统时区设置. 不同的时区可能会引起日期的不同，此外是否支持夏令时等也会影响转换过程的指令踪迹；

(3)是否需要将时间转换成字符串. 将时间转换成字符串的时候，有一个将整数转换成字符串的操作，这个操作的指令踪迹依赖于整数的数量级. 由于时间中的微秒、秒部分都很容易在两次运行中出现数量级上的差异，所以转换时间到字符串的时候很容易出现指令踪迹的差异. 通过在不同的执行环境下设置相同的时区，并且在运行程序时设置相同的系统时间，可以消除上述前两点因素的影响. 但是由于第 3 个因素依赖于进程时间在微秒级上的精确匹配，除非在第 1 次执行的时候记录所有时间相关系统调用的返回值，下一次执行的时候重放，否则无法保证两次执行的指令踪迹完全相同.

此外，有些程序的行为受程序执行速度的影响，比如"每隔固定时间间隔进行一次处理"这类行为. 对于这样的程序，即使在同一个平台上运行多次，也无法保证它每次的执行踪迹都是相同的，在不同的平台上就更无法保证了.

3.3.4 其它环境因素

进程的运行目录. 有些程序的执行踪迹会受到进程当前目录的影响. 比如使用 getcwd 系统调用获取当前目录，然后利用当前目录构造数据文件的完整路径，并且在堆中为其分配存储空间. 因此当前运行目录的不同会引起堆状态的差异，堆状态的差异继而导致后续所有堆内存分配操作的指令踪迹都受

到影响.

文件系统状态. 有些程序的指令踪迹会受到文件系统的当前状态的影响. 比如一些 Fortran 程序在创建输出文件之前会检测该文件是否已经存在，如果已经存在的话就先将其删除. 也有一些程序会扫描当前运行目录下的所有文件. 对于这类程序，如果当前运行目录下的文件不同的话，就会导致进程指令踪迹的差异.

外部事件. 有些程序的行为依赖于程序接收到的 I/O 事件，如果两次执行时 I/O 事件到达的顺序不同，或者每次到达的数据量不同，也会导致程序指令踪迹的差异. 比如使用类似 select 等机制等待 I/O 事件的程序就是这种类型. 因为这类程序的内在行为依赖于外部随机因素，所以无法保证两次执行的指令踪迹匹配.

随机数. 有些程序中会使用到随机数. 如果程序中使用的是伪随机数，只要随机数种子相同，就可以保证随机数序列相同. 但是有些程序使用一些不确定性的数据作为随机数种子，比如当前时间、系统中断事件（urandom）等，对于这类程序，很难在两次运行中重现随机数序列，也就无法保证指令踪迹的一致性.

4　评　估

为了验证第 3 节中提出的避免执行踪迹差异的几种方法的有效性，本文将提出的各种方法分别应用到 QPoint 上，然后运行了 SPEC2006 中的所有测试程序，并与这些程序在 Simics 模拟器上执行时的总提交指令数进行了比较，结果如表 2 所示.

表 2　应用了消除程序指令踪迹差异的方法后，SPEC2006 测试程序在 QPoint 和 Simics 下分别执行时的总提交指令数差异

测试程序	Δ_0	Δ_1	Δ_2	Δ_3
400.perlbench	639357210	669785193	33260879	0
401.bzip2	2347	47	14	0
403.gcc	971956566	841796416	22293	0
410.bwaves	360634	353459	14	0
416.gamess	8127	1270	2602	0
429.mcf	110652	109552	88261	0
433.milc	131372027	134305964	19	5
434.zeusmp	7173	314	14	0
435.gromacs	126564	56223	59207	83
436.cactusADM	2214346	2220001	14	0
437.leslie3d	933774	2777321	4399	12
444.namd	119633	117030	117030	0
445.gobmk	2791	376	376	0
447.dealII	80984240	1853802	14	0
450.soplex	189571625	178225830	14	0
453.povray	996823	171847	1106492	0
454.calculix	413487735	413190921	924	0
456.hmmer	1635782	1633486	1633486	0
458.sjeng	2135	14	14	0
459.GemsFDTD	28012068	27877521	14	0
462.libquantum	28129802	14	14	0
464.h264ref	5750	332	332	0
465.tonto	505816997	821760286	402477578	345
470.lbm	46147	43996	43996	0
471.omnetpp	204567209	17195	17195	9174
473.astar	1095	14	14	0
482.sphinx3	172510670	285205	285205	0
483.xalancbmk	3702935846	1163328	1163330	0

表 2 中的 Δ_0 表示未采用本文中方法时的总提交指令数偏差，Δ_1 表示消除进程的初始状态差异后的指令偏差，Δ_2 表示消除进程的初始状态差异和操作系统内核影响之后的指令偏差，Δ_3 表示使用本文中提出的全部方法之后的指令偏差. 可以看出，应用了本文中提出的方法之后，对于 28 个测试程序中的 23 个，在 QPoint 上执行与在 Simics 模拟器中执行时得到的总提交指令数完全相同，另外的 5 个只有细微的指令总数差异. 相对于 10 M～100 M 的模拟点大小，这种量级的差异对模拟点定位带来的影响可以忽略不计.

表 2 中的实验结果说明，对于影响指令踪迹的大部分因素，都可以利用本文中提出的方法消除其影响. 但是也有一些因素的影响是无法避免的，比如运行速度依赖、外部事件依赖等. 体系结构的某些特性也可能会引起一些不确定性，比如 SPARC 平台上的寄存器窗口溢出机制. 受这些因素影响的程序，其内在特性本身就是不确定的，即使在完全相同的环境中连续运行两次也可能出现执行踪迹的差异. 但是这些因素往往只影响程序的局部状态，不会导致执行踪迹出现较大的差异. 对于表 2 中指令总数存在差异的 5 个测试程序，经分析后确认都是由这些不可消除因素引起的. 其中 433. milc 和 471. omnetpp 测试程序的总指令数差异是由 SPARC 体系结构的寄存器窗口溢出机制引起的；435. gromacs，437. leslie3d 和 465. tonto 测试程序的总指令数差异是由时间日期因素引起的.

从表 2 可以看出，应用了本文中提出的所有方法之后，大部分程序在基于 DBT 的 BBV Profiling 执行和模拟执行时的指令踪迹完全相同. 当程序两次执行间的指令踪迹完全相同的时候，两种方式下得到的 BBV Profile 就完全相同，从而提取出的代表性模拟点也将完全相同. 另一方面，由于执行踪迹完全相同，模拟时的模拟点定位也不会有任何偏差. 因此，应用了本文中提出的方法之后，基于 DBT 加

速的模拟流程能够得到与基于功能模拟的模拟流程完全相同的模拟精度.

5 结 论

本文发现,DBT 加速的模拟方法会带来较大的模拟误差.误差产生的根源在于应用程序在模拟执行时无法精确重现 DBT 执行时的指令踪迹,从而导致模拟点定位出现偏差.本文提出了一个程序执行踪迹模型,以此为基础分析了引起程序指令踪迹不可重现的可能因素,并给出了避免其影响的方法.实验结果显示,提出的方法可以保证程序执行过程的精确重现,从而消除 DBT 加速带来的模拟结果误差.文中提出的方法也可以用于程序调错等需要精确重现程序执行踪迹的应用领域中.

参 考 文 献

[1] Yi J J, Kodakara S V, Sendag R et al. Characterizing and comparing prevailing simulation techniques//Proceedings of the 11th International Symposium on High-Performance Computer Architecture. Toronto, Ontario, Canada, 2005: 266-277

[2] Zhang Fu-Xin, Zhang Long-Bing, Hu Wei-Wu. Sim-Godson: A godson processor simulator based on SimpleScalar. Chinese Journal of Computers, 2007, 30(1): 68-73(in Chinese)
(张福新,章隆兵,胡伟武.基于 SimpleScalar 的龙芯 CPU 模拟器 Sim-Godson.计算机学报,2007,30(1):68-73)

[3] Sherwood T, Perelman E, Hamerly G et al. Automatically characterizing large scale program behavior//Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, California, 2002: 45-57

[4] Hamerly G, Perelman E, Lau J et al. Simpoint 3.0: Faster and more flexible program phase analysis. Journal of Instruction Level Parallelism, 2005, 7(4): 1-28

[5] Gschwind M, Altman E, Sathaye S et al. Dynamic and transparent binary translation. Computer, 2002, 33(3): 54-59

[6] Ebcioglu K, Altman E, Gschwind M et al. Dynamic binary translation and optimization. IEEE Transactions on Computers, 2002, 50(6): 529-548

[7] Patil H, Cohn R, Charney M et al. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation//Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture. Portland, Oregon, 2004: 81-92

[8] Weaver V M, McKee S A. Using dynamic binary instrumentation to generate multi-platform simpoints: methodology and accuracy//Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers. Goteborg, Sweden, 2008: 305-319

[9] Zhao Tian-Lei, Tang Yu-Xing, Qi Shu-Bo et al. Qpoint: Generating simpoint basic block vector profiles efficiently with dynamic binary translation//Proceedings of the 2011 International Conference on Computers, Communications, Control and Automation. Hong Kong, China, 2011: 62-65

[10] Mytkowicz T, Diwan A, Hauswirth M et al. Producing wrong data without doing anything obviously wrong!//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, DC, USA, 2009: 265-276

[11] Zhao Tian-Lei, Jiang Jiang, Fu Gui-Tao et al. Accelerating the extraction of representative behaviors of programs with dynamic binary translation//Proceedings of the 13th International Conference on High Performance Computing and Communications. Banff, Canada, 2011: 235-243

[12] Perelman E, Hamerly G, Calder B. Picking statistically valid and early simulation points//Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, 2003: 244-254

[13] Magnusson P S, Christensson M, Eskilson J et al. Simics: A full system simulation platform. Computer, 2002, 35(2): 50-58

[14] Sun Microsystems. SYSTEM V Application Binary Interface, SPARC processor supplement. 3rd Edition. California, USA: The Santa Cruz Operation, Inc., 1996

ZHAO Tian-Lei, born in 1982, Ph.D. candidate. His research interests include memory system optimization for multi-core processors and computer architecture simulation.

TANG Yu-Xing, born in 1977, Ph.D., associate professor. His research interests include micro-processor design and verification and binary translation.

XU Wei-Xia, born in 1963, M.S., professor. His research interests include high performance mainframe computer architecture.

FU Gui-Tao, born in 1980, Ph.D. candidate. His research interests include cache coherency protocols.

QI Shu-Bo, born in 1982, Ph.D. candidate. His research interests include network on chip for multi-core processor.

JIA Xiao-Min, born in 1982, Ph. D.. Her research interests include high performance on chip memory systems.

ZHANG Min-Xuan, born in 1954, M. S., professor. His research interests include high performance micro-processor architecture, VLSI design and high performance mainframe computer architecture.

Background

Simulation is very important to computer architecture researches. Unfortunately, the speed of simulation is very slow: hundreds of KIPS for detailed performance simulation. For modern benchmark suites, e.g. SPEC2006, the time required for full simulation is usually several years. As computer architecture research steps into multi-core era, this problem gets more and more severe. To solve this problem, researchers have proposed the SimPoint methodology to reduce simulation time. However, generating BBV (Basic Block Vector) profiles for the SimPoint tool requires the benchmark to be executed on a functional simulator. As the speed of functional simulation is typically 10MIPS, this implies that 3 months is needed to generate BBV profiles for all the SPEC2006 benchmarks. As a result, the prevalence of the SimPoint methodology is severely affected by the difficulties of the generation of BBV profiles. Although the SimPoint methodology has been proposed for almost 10 years, there are still a lot of researches using the simple fast-forward X, warm-up Y and simulate Z method which has already been shown to be quite inaccurate by Yi et al. (Joshua J. Yi 2005).

Several recent works propose solving the problem of BBV profile generation with the help of dynamic binary translation (DBT) technology. However, although the DBT method can greatly reduce the time required for BBV profile generation, the accuracy implication of this method is not studied. This paper studies the accuracy implication of the DBT method and observes that the DBT method can incurs an error of 20% for several SPEC2006 benchmarks. The underlying causes of the errors are carefully analyzed. The causes are the mismatch of execution traces of programs under different execution environments. To assist the execution trace analysis, an execution model of processes is proposed. Several affecting factors are identified. Appropriate methods are proposed respectively. Experimental results show that with the proposed methods, the accuracy of the DBT method can be guaranteed the same as the traditional functional simulation based method.

The accuracy problem of the DBT method is first studied by this work. This work completes previous works on DBT accelerated SimPoint methodology. With the contribution of this work, the DBT method can be both fast and accurate. We believe this work, combined with previous works, can greatly improve the efficiency and quality of computer architecture researches.