一种 Dalvik 虚拟机的自适应编译系统

凌 明,武建平,冯克环
(东南大学国家专用集成电路系统工程技术研究中心,江苏南京 210096)

摘 要: 在深入分析 Dalvik 虚拟机自适应编译系统的瓶颈之后,针对当阈值计数器自减到零时,系统初始化导致本地代码多次等待的情况,提出一种阈值重置策略;在热点代码密集的应用程序中,针对编译线程因积压大量等待处理任务而导致编译滞后情况,提出一种基于结果反馈的动态自适应阈值改进策略,结合阈值最优化理论,最大限度地挖掘动态编译的性能收益.实验结果显示,基于三星 Galaxy S 平台,经过阈值重置优化后,Dalvik 虚拟机的循环执行效率平均提升 7%,分支跳转、派发指令执行效率平均提升 5%;采用动态自适应阈值优化后,分支跳转、派发指令执行效率平均提升 8%.

关键词: Dalvik 虚拟机; 自适应编译系统; 阈值重置; 动态自适应阈值
中图分类号: TN302 **文献标识码:** A **文章编号:** 0372-2112 (2013) 08-1622-06
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2013.08.027

An Adaptive Compilation System Based on the Dalvik Virtual Machine

LING Ming, WU Jian-ping, FENG Ke-huan
(*National ASIC System Engineering Research Center, Southeast University, Nanjing, Jiangsu* 210096, *China*)

Abstract: After the bottle-neck of the adaptive compile system for Dalvik being analyzed, according to the scenery when the threshold counter decrements to zero, a lot of waiting time will be caused by system initialization, the strategy based on threshold reset is introduced. According to some applications centralized with hot-spot codes, which will caused a great deal of compiling work to be suspended, a strategy of dynamic adaptive threshold based on a result feedback mechanism, is implemented. It could maximize the performance enhanced by the dynamic compilation scheme. Compared to the Dalvik before optimization, the experimental result based on the platform of Galaxy S of Samsung Corporation indicates that the execution efficiency of loop operations is improved by 7% on average after optimized by the threshold reset strategy, the execution efficiency of conditional jump operations and instruction dispatch are improved by 5% on average and 8% assisted by the dynamic adaptive threshold scheme.

Key words: Dalvik; Adaptive compilation system; threshold reset; dynamic adaptive threshold

1 引言

伴随着移动时代的到来,移动智能操作系统迅速崛起.Google 公司推出的 Android 系统以其灵活多变的扩展性和绚丽多彩的图形用户界面迅速成为工业界和学术界的研究热点.为了满足跨平台运行,Android 系统应用程序采用 Java 语言编写.Java 应用程序的通用性和可移植性源于虚拟机的设计与实现[1,2].移动终端设备的处理性能和内存资源受限,传统 Java 虚拟机不能满足智能操作系统需求[3].因此,Google 公司定制开发了一款 Java 虚拟机:Dalvik 虚拟机,为 Android 系统的应用程序提供良好的运行环境和执行引擎[4].

Android 系统推出以来,国内外众多科研单位所针对它进行了多方面的研究工作,但与 Dalvik 虚拟机相关的研究并不多.Dalvik 虚拟机的自适应编译技术早在 Java 虚拟机中已有所涉及,如 Sun 公司的 HotSpot 技术[5].与 HotSpot 相关的研究对 Dalvik 虚拟机的研究有着重要指导意义.韩洪波等人基于 J2ME 平台提出一种采样与计数相结合的,程序运行时的混合型热点信息分析方法,并与运行时信息记录引导的热点预判机制相结合[6].史辉辉等人提出一种高效的基于路径的热点信息收集方法,将一些频繁执行的连续基本块序列合并成具有单入口、多出口的超级块,进而减少因基本块执行结束时所带来的上下文频繁切换[7].黄耀志等人提出一种本地代码在线程间共享的机制,能够有效减少线程间相同代码的重复编译和缓存消耗[8];Igor 等人提出一种编

收稿日期:2012-06-15;修回日期:2013-04-03;责任编辑:马兰英
基金项目:国家科技重大专项(No.2009ZX01031-001-004-3);青蓝工程资助项目

译任务按执行频率高低顺序排队的动态编译策略[9]. Chandra 等人提出一种在线和离线相结合的热点信息采集机制[10].

由本文针针对 Dalvik 虚拟机的自适应编译系统的瓶颈提出两种优化方案来提高字节码的执行效率.

2　Dalvik 自适应编译系统

2.1　执行流程简介

Dalvik 虚拟机采用解释执行与即时编译相结合的自适应编译执行技术.虚拟机包含两种工作状态:INTERP_STD 与 INTERP_DBG.图 1 是 Dalvik 虚拟机的执行流程.

首先,虚拟机进入 INTERP_STD 状态.在字节码流的解释执行过程中,通过一个阈值计数系统来统计特定字节码对应的 rPC(Dalvik 虚拟机的模拟通用寄存器)的调用次数.当阈值计数器自减到零时,热点信息采集系统就认定当前的 rPC 为一条潜在的 trace head.研究发现,一个应用程序 80%的执行时间花费在 20%的代码上[11],而 trace 就是指那些调用频率高、执行时间长的热点代码.然后,判断当前的 trace head 是否已编译.如果已编译,则直接使用保存在代码缓存中的本地代码.否则,虚拟机切换到 INTERP_DBG 状态,建立完整的 trace 信息.若干个 trace head 和 trace tail 即可建立一条完整的 trace 信息.最后,trace 信息建立以后,虚拟机会将当前的 trace 加入到编译任务队列中.

2.2　阈值计数系统

从执行流程的分析可以看出,阈值计数系统提供了解释器到即时编译器的入口.

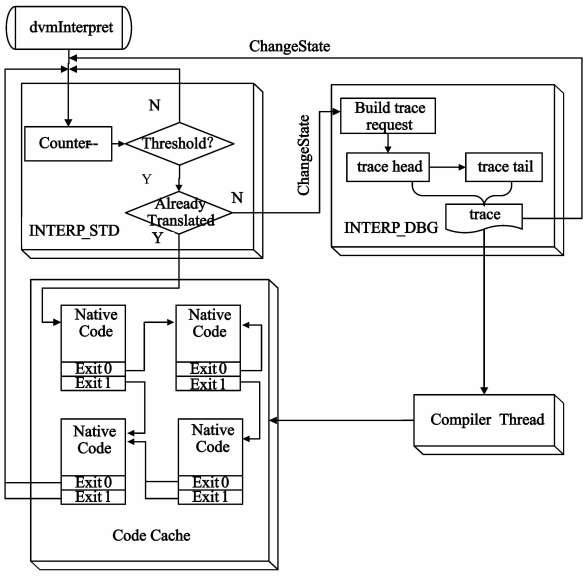

图1　Dalvik虚拟机自适应编译系统的执行流程

图 2 是阈值计数系统的执行流程.pJitProfTable 是一个大小为 2048B 的 hash 表.hash 表中存储的是 rPC 所对应的阈值计数器,如果当前 rPC 对应的字节码指令调用一次,则相应的阈值计数器自减一(armv7-a-neon 架构中阈值计数器的初始值为 40).当某个 rPC 对应的字节码指令调用 40 次后,阈值计数器会自减到零,当前字节码指令可作为热点代码的首条指令,这样解释器即会捕捉到一个 trace head.然后通过函数 dvmJitGetCodeAddr(rPC)查看当前热点是否被编译过,若已编译则直接跳转到相应的本地代码中执行;否则,解释器将切换到调试状态,开始热点确定,建立热点信息并通知编译线程编译当前热点.

3　阈值计数问题

3.1　多次等待

如上面的分析,函数 common_updateProfile 会统计和维护 Dalvik 虚拟机的阈值计数器.当某个阈值计数器自减到零时,首先查看当前 rPC 所对应的热点代码是否被编译过,若未编译,则切换到 Dalvik 的调试状态去请求编译;否则,它会获取热点代码对应的本地代码地址并跳转到本地代码中执行.

主线程(Main Thread)包含三种执行状态:标准状态解释执行(INTERP_STD)、调试状态解释执行(INTERP_DBG)和本地代码编译执行(Native Code).以 armv7-a-neon 架构为例,阈值计数器的初始值为 40,当主线程的某段热点代码在 INTERP_STD 状态执行 40 次后,阈值计数器自减到零.若系统发现当前的热点代码未编译过,则立即切换到调试状态解释执行并建立 trace 信息.

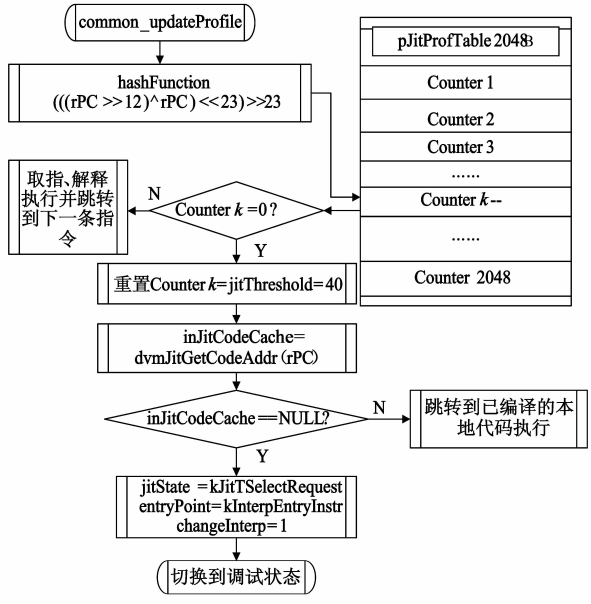

图2　阈值计数系统的执行流程

由于编译时间的不确定性,会发生下面三种情况:

(1)编译时间正好等于 40 次的解释执行时间(如图 3(a)中的 Compiler Thread 1).当阈值计数器第二次自减到零时,发现当前的本地代码已编译,主线程则跳转到本地代码中去编译执行.

(2)编译时间小于 40 次的解释执行时间(如图 3(a)中的 Compiler Thread 2 所示).当阈值计数器自减到零后,不仅切换 Dalvik 执行状态,而且把阈值计数器再次初始化为热点阈值.第一次初始化阈值是为了发现热点,而第二次初始化阈值是为了等待先前发现的热点完成编译.因此,从第一次解释执行热点代码到第一次编译执行热点代码至少需要 2 * gDvm.jitThreshold 次热点代码调用,但并非值得第二次等待阈值计数器自减到零,而多次等待处理并不合理;

(3)编译时间大于 40 次的解释执行时间(如图 3(b)中 Compiler Thread 3 所示).此时,热点代码的编译时间很长,超过了 40 次的解释执行时间.因此,当第二次阈值计数器自减到零时,Dalvik 会发现 rPC 对应的本地代码地址为空,误认为当前的热点代码还未编译,将会切换到调试状态去请求编译.

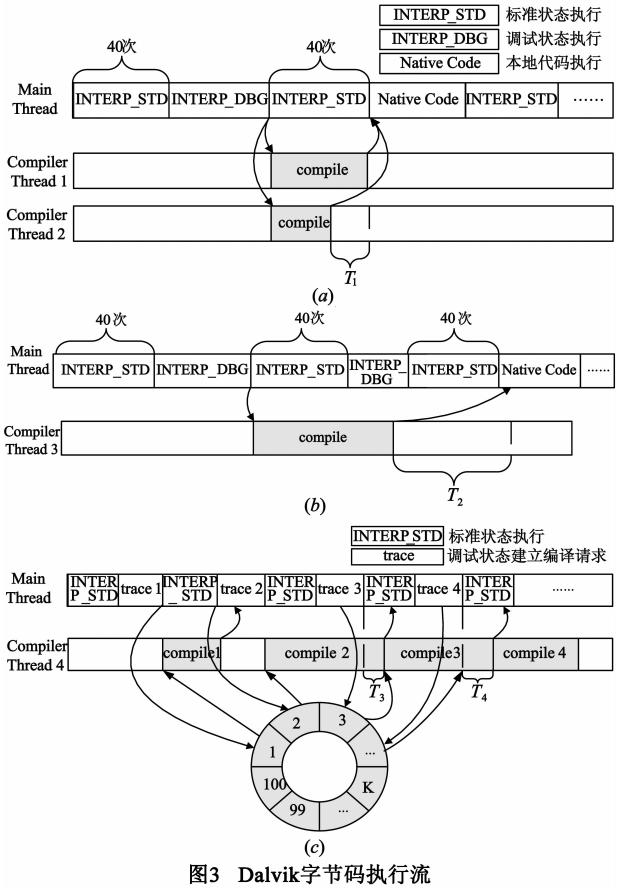

图3 Dalvik字节码执行流

综上,第一种情况属于小概率事件,而第二种和第三种情况发生的概率很大,即无论编译时间大于或小于 40 次解释执行时间,这种阈值计数器自减到零后立即初始化并不合理.当热点代码大量存在时,(a * T_1 + b * T_2)的时间将会很大,因此,这种决策对整体性能会产生很大影响(图 3(a)为第二种情况发生的次数,图 3-(b)为第三种情况发生的次数).

3.2 编译滞后

当程序中含有大量 trace 时,编译线程中会积压大量的待处理任务.由 Dalvik 虚拟机性能测试工具——CaffeineMark 的 logcat 信息可看出,变量 compilerQueue-Length 为循环队列中非空任务的个数.在 CaffeineMark 运行过程中,等待编译线程编译的任务个数一度达到了 65 个(最大长度为 100),即早期热点代码因不能被及时处理而积压,势必会降低自适应编译系统的响应速度.

下面以图 3(c)中 Dalvik 字节码执行流来分析编译滞后过程.主线程 Main Thread 执行字节码时不断发现新的热点代码,并建立 trace 信息.一方面,Dalvik 在解释执行过程中发现 trace 1,trace 2…添加到任务队列中,从而不断向循环队列中添加编译任务;另外,编译线程从任务队列中获取任务进行编译.考虑以下情况,trace 2 在编译过程中,解释执行引擎发现了另一个热点代码 trace 3,此时,trace 3 必须等待 trace 2 编译完成才能被编译线程处理.随着时间的推进,队列中会积压更多待处理的编译任务.上面的 logcat 信息展示了 CaffeineMark 在最坏的情况下,有 65 个编译任务等待编译线程处理.图中 T_3、T_4 分别为为 trace 3、trace4 的等待编译时间,若队列中等待任务有 K 个,那么总体等待的时间 $T = \sum T_K$.K值越大,等待时间越长,性能影响越严重.

由上面的分析可以看出,编译滞后问题主要由某些应用程序的热点多且比较集中引起,导致编译线程中积压大量待处理任务.理想状况是,每一次发现的 trace 均能被及时编译,与程序本身无关,自适应编译系统能使编译随着时间的推进而进行.

4 改进策略

4.1 阈值重置策略

通过以上分析可以发现,Dalvik 虚拟机的热点信息采集机制有待改进.如 3.1 节中所述,多次等待问题主要由于阈值计数器自减到零之后立即初始化所导致.由于编译时间的不确定性,导致很早被发现的热点代码得不到及时编译执行.多次等待时间可能是一个或者多个阈值自减到零的解释执行时间.多次等待时间累积越长,对执行性能的影响越大.

如图 4,为了解决本地代码的多次等待问题,本文提出一种阈值重置改进策略——当且仅当本地代码地

址不为空时,才进行阈值初始化.

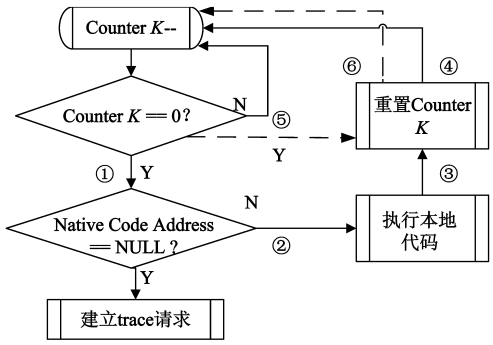

图4　阈值重置系统

通常,当 rPC 所对应的阈值计数器自减到零时,会立即重置当前计数器为初始阈值(如图中的虚线箭头所示,执行路径为⑤⑥).改进后的阈值重置系统,在阈值计数器自减到零后并不立即重置阈值,而是先判断本地代码地址是否为空,即上一次添加的编译任务是否已完成编译;若完成,则直接跳转到本地代码片段执行并重置计数器为初始阈值(如图中的实线箭头所示,执行路径为①②③④).

4.2　动态自适应阈值策略

如 3.2 节所述,编译滞后问题会影响自适应编译系统的响应速度.程序运行过程中,发现的热点密集时,则会增加编译线程的压力,导致编译任务被大量积压,热点代码对应的本地代码不能及时提供给主线程,从而影响自适应编译系统的执行性能.为了避免这种情况,本文基于闭环控制思想提出一种基于结果反馈的改进策略.如图 5 所示,通过编译线程的任务个数反馈来调节阈值:

(1)如果程序中热点代码较多,超过一定的额度即编译线程中有任务积压.通过反馈系统调节阈值,使得热点的选择更具竞争性,从而解决编译滞后的问题;(2)随着阈值的提高,代码能被编译的门槛越来越高.虽然编译线程会变得"清闲",但能被发现的热点太少,自适应编译系统不能获取动态编译的最大性能收益,

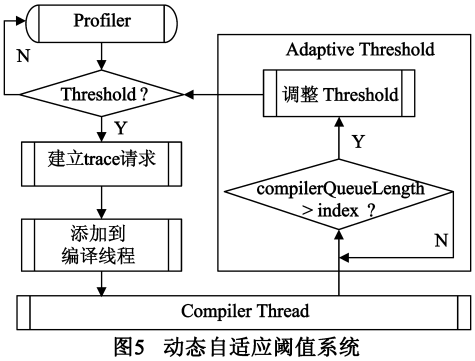

图5　动态自适应阈值系统

这违背了阈值最优理论(每个应用程序对应一个阈值最优点[12]).

图 6 是在图 5 基础上的改进方案.虽然热点采集系统无法预测程序的动态特性和未来趋势,但可通过程序的执行历史信息来确定阈值大小.一方面,当任务队列长度高于阈值时,提高阈值,使热点的选择更加"苛刻",从而有效减少队列的长度,避免编译滞后情况;另一方面,当任务队列长度低于阈值时,降低阈值,尽量发挥动态编译所带来的性能提升,在程序运行过程中确定阈值的最优值.阈值必须有一个最低限,如果系统认为编译线程一直空闲,就去降低阈值而导致很多冷代码片段被编译,最终,不仅不能提升整体执行性能,甚至反而导致性能下降和代码缓存的溢出.

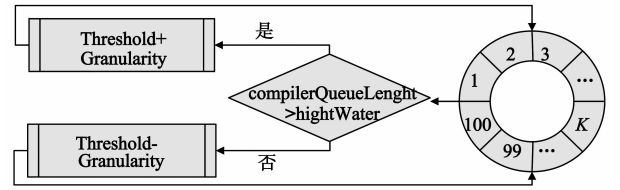

图6　阈值最优调节系统

5　实验与分析

本文的硬件测试平台选用三星公司的 Galaxy S,它采用三星的 S5PC110 为核心处理器.该处理器采用 ARM 公司的 Cortex-A8 架构,主频 1GHz,45nm 制造工艺.实验测试中,分别从阈值重置和动态自适应阈值两个方面来进行性能测试和分析.最终测试数据为 CaffeineMark 运行五次的平均值.

软件测试平台选用 Android 2.3(Gingerbread).测试程序选用虚拟机专用测试软件 CaffeineMark.

表 1　阈值重置改进前后 CaffeineMark 在 Galaxy S 平台测试结果

CaffeineMark	优化前	优化后	提升性能比
Sieve	5679	6010	5.83%
Loop	11940	12814	7.32%
Logic	10055	10593	5.35%
String	3716	3713	-0.08%
Float	9120	9543	4.64%
Method	4126	4141	0.36%
total	6814	7073	3.80%

从表 1 中的数据可以看出:

(1)Loop 性能提升效果最为显著.主要原因是:循环中热点代码较多且调用频繁,多次等待问题产生的时间差 T_1 累积和很大.经过阈值重置改进之后,编译好的本地代码能立即执行.

(2)Sieve、Logic 和 Float 性能提升效果比较明显.

Sieve 和 Logic 中热点代码由于查找、分支指令跳转和指令派发使得其解释执行 40 次的时间,明显大于编译时间,导致多次等待中第二种情况的时间差 T_1 较大,因此,多次等待问题的消除对其性能的影响较为明显.

(3)String 和 Method 性能基本无提升.String 主要测试字符串处理速度,Method 主要测试递归方法调用速度.这两项测试与动态编译器的具体实现细节和优化策略相关,所以性能基本没有提升.

表 2 是 CaffeineMark 在 Galaxy S 平台上,经过动态自适应阈值改进之后的测试结果,可以看出:

(1)Sieve 和 Logic 性能提升效果较为明显.这是因为 Sieve 和 Logic 中的查找、分支指令跳转和指令派发正好是热点信息采集系统所重点统计的指令对象,但这些指令的执行频率却没有 Loop 中那么高,性能对于阈值的调节变化很敏感,通过动态采集之前的信息来找到阈值最优点,从而最大程度获取动态编译所带来的性能收益.

(2)Loop 测试结果在优化前后变化不大.主要因为循环是热点信息采集系统关注的指令,但当循环的次数较多时(远远大于阈值的调节范围),不论阈值在一定范围内如何变化,对于循环中热点代码的采集都没有影响.

(3)String 和 Method 的性能基本不变.这与上面阈值重置优化前后的测试结果基本一致.测试中出现的一些随机性和偶然性导致其测试结果出现微小的波动,甚至降低,但并不影响整体优化效果.

表 2 动态自适应阈值改进前后测试结果

CaffeineMark	优化前	优化后	提升性能比
Sieve	5679	6101	7.43%
Loop	11940	11952	0.10%
Logic	10055	10923	8.63%
String	3716	3709	-0.19%
Float	9120	9146	0.29%
Method	4126	4103	-0.56%
total	6814	7039	3.30%

6 结论

本文在分析 Dalvik 虚拟机自适应编译系统的执行流程和阈值计数器的实现机制之后,为了提高 Dalvik 字节码的执行效率,提出两种改进策略来解决阈值计数系统所存在的问题.针对当阈值计数器导致的本地代码多次等待的情况,提出一种阈值重置策略;针对在热点代码密集的应用程序中,编译线程由于大量等待处理任务的积压而导致编译滞后情况,提出一种基于结果反馈的动态自适应阈值改进策略,有效缓解编译压力,并结合阈值最优化理论,最大限度地挖掘动态编译所带来的性能提升.实验结果显示:经过阈值重置优化之后,Dalvik 虚拟机的循环执行效率、分支跳转指令与派发指令的执行效率均明显提升,Dalvik 虚拟机性能得到显著改善.

参考文献

[1] 敖琪,蔡嵩松,王剑.基于硬件 cache 锁机制的 Java 虚拟机即时编译器优化[J].计算机研究与发展,2012,49(z1):185 - 190.
Ao Qi,Cai Song-song,Wang Jian.JVM JIT compiler optimization based on cache locking mechanism[J].Journal of Computer Research and Development,2012,49(z1):185 - 190.(in Chinese)

[2] 桂先洲,黄卫东.实时编程语言 RTS/Java 设计[J].电子学报,2002,30(2):153 - 157.
Gui Xian-zhou,Huang Wei-dong.Design of real-time programming language RTS/Java[J].Acta Electronica Sinica,2002,30(2):153 - 157.(in Chinese)

[3] 王文鼐,赵生妹.基于 JVM 的分布计算在网络仿真中的应用研究[J].电子学报,2001,29(6):804 - 807.
Wang Wen-nai,Zhao Sheng-mei.On networks simulating by JVM-based distributed processing[J].Acta Electronica Sinica,2001,29(6):804 - 087.(in Chinese)

[4] Chien-Wei C,Chun-Yu L,Chung-Ta K,et al.Implementation of JVM tool interface on dalvik virtual machine[A].2010 International Symposium on VLSI Design Automation and Test(VLSI-DAT)[C].Chutung,Hsinchu,Taiwan,2010.143 - 146.

[5] Yan W,Jin-jing Z,Hua C,et al.HotSpotInsight:a java application introspection platform based on JVM[A].2011 1st International Conference on Instrumentation,Measurement,Computer,Communication and Control[C].Beijing,China,2011.843 - 847.

[6] 韩洪波,倪宏,韩锐,等.一种混合型运行时信息分析方法[J].微计算机应用,2009,29(9):1 - 2.
Han Hong-bo,Ni Hong,Han Rui,et al.A mixed runtime information analysis methed [J].Microcomputer Applications,2009,29(9):1 - 2.(in Chinese)

[7] 史辉辉,管海兵,梁阿磊.动态二进制翻译中热路径优化的软件实现[J].计算机工程,2007,33(23):78 - 83.
Shi Hui-hui,Guan Hai-bing,Liang A-lei.Hot path optimization in software dynamic binary translation[J].Computer Engineering,2007,33(23):78 - 83.(in Chinese)

[8] Huang Yao-Chih,Chen Yu-Sheng,Yang Wuu,et al.File-based sharing for dynamically compiled code on dalvik virtual machine[J].ICS,2010,16(18):489 - 494.

[9] Igor Böhm, Tobias J K Edler von Koch, Stephen Kyle, et al. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator [J]. ACM, 2008, 11(6): 7 – 10.

[10] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance [J]. Code Generation and Optimization, 2003, 11(9): 69 – 78.

[11] Andreas Gal, Brendan Eich, Mike Shaver, et al. Trace-based Just-in-time type specialization for dynamic languages [J]. ACM, 2009, 19(8): 1 – 3.

[12] Evelyn Duesterwald, Vasanth Bala. Software profiling for hot path prediction: less is more [J]. Hewlett-Packard Labs, 2000, 19(6): 1 – 4.

作者简介

凌　明　男,1972 年出生,江苏淮阴市人,博士,副教授,硕士生导师,主要研究方向为嵌入式系统设计学,SoC 存储子系统.

E-mail: trio@seu.eud.cn

武建平(通信作者)　男,1977 年出生,陕西蒲城县人,博士生,主要研究方向为 SoC 存储子系统优化.

E-mail: herofly2005@163.com