一种新的自适应翻译单元构造算法

曹宏嘉, 肖 勇, 唐遇星, 邓 昆鸟, 周兴铭
(国防科技大学并行与分布处理国家重点实验室, 湖南长沙 410073)

摘 要: 翻译单元的构造对动态二进制翻译系统的性能有着重要影响. 本文提出一种新的硬件支持下的自适应翻译单元构造算法 ATUC, 动态监测程序执行, 根据程序的执行特性动态自适应调整翻译单元的构造, 提高翻译后代码的执行成功率, 并尽可能提高翻译后代码效率. 引入了硬件的连续提交地址缓冲, 辅助二进制翻译软件进行程序执行特性监测, 降低 profile 开销. SPEC2000 程序模拟结果表明, ATUC 算法对系统性能提高明显. 分析表明 ATUC 具有很低的时间空间开销与硬件支持实现开销.

关键词: 动态二进制翻译; 翻译单元构造; 指令踪迹; 轮廓信息
中图分类号: TP302 文献标识码: A 文章编号: 0372-2112 (2005) 08-1360-05

A Novel Algorithm for Adaptive Translation Unit Construction

CAO Hong-jia, XIAO Yong, TANG Yu-xing, DENG Kun, ZHOU Xing-ming
(*National Key Laboratory of Parallel and Distributed Processing, NUDT, Changsha, Hunan 410073, China*)

Abstract: Construction of translation units deeply affects the performance of binary translation systems. In this paper we propose a novel adaptive translation unit construction (ATUC) algorithm. With hardware support, ATUC monitors the execution of the source architecture programs and dynamically adjusts the translation units according to the execution behavior characteristics of the programs. A hardware buffer is introduced to assist the binary translation software to detect continuously committed addresses. ATUC improves the efficiency of the translated code while guaranteeing the high success execution ratio of the translated code segments. Simulation results of the SPECint2000 benchmark show that ATUC has up to 31% performance advantages over other translation unit construction methods.

Key words: dynamic binary translation; translation unit construction; trace; profile

1 引言

动态二进制翻译(Dynamic Binary Translation, DBT)技术在程序运行过程中将源体系结构的指令翻译成目标体系结构指令执行. 二进制翻译处理器通过在硬件处理内核(Hardware Processing Core, HPC)上运行二进制翻译软件(Binary Translation Software, BTS)实现二进制代码兼容. 这种硬核加软件翻译的技术可以简化硬件设计, 利于减少芯片面积、提高时钟频率、降低功耗和制造成本等, 还可避开处理器硬件的专利限制. DBT 技术有很高的研究价值和应用前景, 特别适合中国的国情.

翻译单元(Translation Unit, TU)是 DBT 系统中进行代码翻译的程序级单位. 不同 TU 在所需 profile 开销、翻译后代码执行成功率以及翻译后代码效率等方面对 DBT 系统的性能有着重要影响. 当前 DBT 系统多采用程序中多次执行的单路径指令序列即 Hot Trace 作为 TU, 以期得到高效目标代码.

本文提出一种硬件支持下的自适应翻译单元构造(Adaptive Translation Unit Construction, ATUC)算法, 动态监测程序执行的行为特性, 并相应调整翻译单元 Trace 的构造以提高系统性能; 引入硬件的连续提交地址检测缓冲对翻译后代码的成功执行进行监测以降低 profile 开销. 详细模拟与分析表明, ATUC 能够以低开销实现对程序执行行为的动态适应, 对 DBT 系统性能提升效果显著.

2 相关工作

研究项目 DAISY[1] 利用 DBT 技术将 PowerPC 指令转换到树状 VLIW 上执行. DAISY 中的 TU 为程序页面, 翻译过多的非热点代码. Aries 系统[2] 将 PA-RISC/HP-UX 的用户级代码动态翻译到 IA-64/HP-UX 执行, 它以基本块为翻译单元, 虽然具有构造简单等优点, 但是其翻译后代码的效率低下. BOA 处理器[3] 中的翻译软件 VMM 以 Hot Trace 作为 TU. 它使用统计分支执行频率的方法来构造 Trace, 因为未考虑分支之间的协相关, 不能总是准确预测热点路径.

动态优化系统 Dynamo[4] 中使用低 profile 开销的 NET

收稿日期: 2004-12-07; 修回日期: 2005-03-28
基金项目: 国家自然科学基金(No. 90307001)

方法预测程序的热点路径,并对预测不准的路径进行重新预测. 但是这种预测并不能保证执行成功率: 文献[5]的模拟结果中, NET 算法所预测路径的执行数占程序中所有路径执行数的比例, 对于程序 gcc 和 go 分别仅有 47.5% 和 55.5%. 对于 DBT 系统这将严重降低系统性能.

一般而言, 检测系统中的 Hot Trace 需要高开销的路径 profile. 文献[6]讨论了边 profile 与路径 profile 的关系, 指出仅由低开销的边 profile 并不总是能准确计算出路径 profile 信息. 文献[7]提出一种硬件的基于路径的 Trace 预测技术; 然而其准确率只有 90%, 且需要很高的 profile 开销, 不适用于 DBT 系统.

3　动态二进制代码翻译

为保证源结构指令执行的原子性以及精确异常等语义, 二进制翻译处理器一般采用 Shadow/Commit 的执行机制[3]: 除工作寄存器之外, 在 HPC 中设置 Shadow 寄存器. 在执行翻译后代码片断(Translated Code Segment, TCS) 之前将 Shadow 寄存器的内容复制到工作寄存器, 执行过程中仅修改工作寄存器. 当 TCS 执行结束, 工作寄存器的内容被提交到 Shadow 寄存器中. 若在 TCS 执行过程中发生异常或者 TCS 的实际执行路径与翻译时选取的路径不一致, 则其执行结果被取消, 系统回退到 TCS 执行之前的状态. 本文称 TCS 实际执行路径与 TU 构造所选取路径一致的情况为 TCS 的成功执行, 不一致的情况为失败执行.

DBT 系统的执行时间由解释执行指令时间、代码翻译开销、翻译后代码执行时间与 profile 开销几部分组成:

$$T_{dbt} = T_{interp} + T_{trans} + T_{exec} + T_{prof} = I_{interp} \times t_{interp} + I_{trans} \times t_{trans} + (V_{cmt} + V_{cnl}) \times t_{vliw} + N_{prof} \times t_{prof},$$

其中 I_{interp} 为解释执行的源结构指令数, t_{interp} 为解释执行一条源结构指令的平均时间; I_{trans} 为翻译的指令数, t_{trans} 为翻译一条指令的平均时间; V_{cmt} 和 V_{cnl} 分别为提交与作废的翻译后代码 VLIW 条数, t_{vliw} 为一条 VLIW 的执行时间; N_{prof} 为 profile 代码执行次数, t_{prof} 为 profile 代码的平均执行时间. 提高 TCS 执行成功率是提高系统性能的关键, 因为 TCS 的失败执行不仅造成执行资源浪费, 而且导致 BTS 返回解释执行; 而源结构指令的解释执行速度比翻译后执行慢一到两个数量级[1].

另一方面, 高性能 DBT 系统中翻译执行指令占主要比例, 从而翻译后代码的效率成为影响系统性能的重要因素. 代码效率定义为翻译执行的源结构指令数与系统中提交的 TCS 中的 VLIW 总数之比: $Eff = INST_{transexec}/VLIW_{cmt}$. 在相同翻译执行指令比例下, TCS 代码效率越高, 系统性能越高.

一般而言, TU 跨越的分支指令越多, 其 TCS 的执行从中间退出的概率就越大, 执行成功率越低; 然而, TU 中包含的指令越多, 翻译中对其进行优化与调度时所能看到的指令窗口就越大, 展现的优化机会越多, 生成目标代码的效率越高.

4　课题的环境与平台

课题组开发了一个动态二进制翻译系统 ULDBT[8]. 系统的源结构为 IA-32 结构的整数指令部分, 目标结构为一个自主设计的四发射 VLIW 处理器. 在 VLIW 的调度上采取了一种简化处理: 只要两条指令间不存在数据相关和控制相关即允许调度到同一 VLIW 中.

ULDBT 中 TCS 执行控制指令如表 1 所示. cnlcc 指令在条件满足时取消 TCS 的执行结果, 并返回 BTS 进行解释执行. cmt 指令提交 TCS 的执行结果, 更新源结构指令指针(Source architecture Instruction Pointer, SIP), 并查找对应的 TCS 执行. cmt 指令有几种形式, 分别进行条件/无条件的提交, 以及由偏移/寄存器间接给出下一指令地址. pcmt 指令除完成 cmt 指令的操作外, 还通过硬件缓冲区进行连续提交地址检测(见第 5.2 节).

表 1　TCS 执行控制指令

指令格式	语义
cnlcc cc, creg, offset	若 creg 计算条件 cc 成立则取消执行结果, 转入解释执行; 否则为空操作
cmt offset/treg	提交执行结果, 查找源结构地址(offset + SIP)/treg 所对应的 TCS 并执行
cmtcc cc, creg, offset/treg	同 cmt offset/treg, 但仅在 creg 计算条件 cc 成立时执行
pcmt offset/treg	同 cmt, 但同时进行 CCAD 检测
pcmtcc cc, creg, offset/treg	同 cmtcc, 但同时进行 CCAD 检测

5　自适应翻译单元构造算法

TU 的构造需要兼顾 TCS 执行成功率与代码效率两个因素, 同时不能引入太高 Profile 开销. 自适应翻译单元构造算法(ATUC)采取了优先保证 TCS 执行成功率, 同时尽可能提高代码效率的策略. 算法的整体框架如图 1 所示.

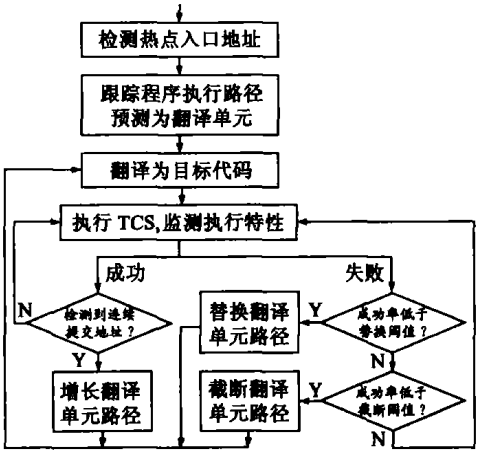


图 1　自适应翻译单元构造算法

5.1　热点路径预测

热点入口地址(Hot Entry Address, HEA)的检测由 BTS 对源结构程序解释执行时为每个潜在的翻译单元入口(Potential Translation Unit Entry, PTUE)维持一个执行计数进行. 当 PTUE 的执行次数到达阈值 THHEA 后, 即认为该地址为热点入口. PTUE 指满足以下条件的指令地址:

(1) 后向分支目标　后向分支经常意味着循环的开始, 而循环是程序中执行热点的主要组成部分.

(2) 不可翻译指令的后继　不可翻译指令将中止 ATUC 中的路径跟踪;将其后继指令地址作为 PTUE,以避免因不可翻译指令的出现导致过多的指令不被翻译.

(3) TCS 的成功执行出口地址　对检测到的 HEA,跟踪其下一次执行的路径并预测为热点路径.理由在于,发现 HEA 意味着程序正在执行一段热点代码,而接下来执行的路径很可能是这些代码中的一部分. BTS 以此路径对应的 Trace 作为从该 HEA 地址开始的初始 TU,翻译为目标代码 TCS. 路径跟踪的结束条件为:

(1) 循环展开次数限制　限制一条 Trace 中循环展开次数至多为 4 次,以避免代码体积膨胀过于严重,同时又能展现一定优化机会.

(2) 指令数目限制　一条 Trace 中最多允许 15 个条件分支和 2 个间接分支.两个间接分支可以允许一个间接调用的函数被完整地内联到 TU 中.

(3) 不可翻译指令.

5.2 程序执行特性监测

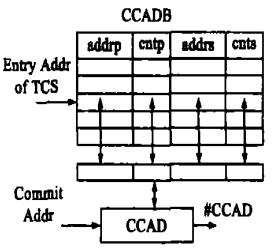
对 TCS 执行成功率的监测在 TCS 的失败执行处理中完成,如图 2 所示.在计算出 TCS 执行成功率 *succFreq* 后,与路径替换阈值 *thPrep* 和路径截断阈值 TH_{PCUT} 比较,并进行相应的路径替换或路径截断.算法为每个 HEA 维持一个路径替换阈值,其初值设为 TH_{PREP},并在每次该 HEA 进行路径替换后减半.这是为了避免从 HEA 开始有多条路径的执行频率相近并且均小于 *THPREP* 而出现频繁为其替换 TU 路径的抖动情况.

```
tcsCanceled(tcs, cancelxferAddr){
  tca->cancel ++
  succFreq= 1-(tcs->cacel/tca->exec);
  if(succFreq< thPrep){
    invalidate(tcs);
    traceExecPath(tcs->enryAddr);
  }else if(succFreq< TH_PCUT){
    invalidate(tcs);
    traceExecPathWithCutAddr(tcs->entry Addr, cancelXferAddr);
  };
}
```

图 2　TCS 失败执行处理

对 TCS 提交地址偏向性的监测由 BTS 与硬件协作完成. BTS 选择满足以下条件的"短"TU 进行偏向性监测: ①长度小于系统中 TU 平均长度; ②代码效率低于系统中 TU 的平均代码效率; ③不是以不可翻译指令结束.对这些 TU,翻译时采用 pcmt 替代 cmt 作为提交指令.该指令执行时通过 HPC 中硬件的连续提交地址检测缓冲(Continuous Commit Address Detection Buffer, CCADB)进行提交地址偏向性检测,并在检测到偏向性条件满足时触发异常,通知 BTS 为相应的 HEA 增长 TU.

CCADB 的结构如图 3(*a*)所示,它实现如图 3(*b*)所示的连续提交地址检测算法.对需要进行检测的每个 HEA, CCADB 维持两个候选的连续提交地址及计数.算法中的参数 *I* 与 *D* 控制提交地址偏向性阈值,即 TCS 成功执行时提交的下一指令地址与主候选地址(addrp)匹配的比例.设在 *N* 次连续 TCS 成功执行中有 *X* 次提交地址与主候选地址匹配,要使主计数值增加,应该有: $X\times I-(N-X)\times D>0$,即 $TH_{PGROW}=X/N>D/(I+D)$.阈值 TH_{CCAD} 与参数 *I* 控制了检测阶段的长度:主计数值至少要经过 TH_{CCAD}/I 次增加才会达到 TH_{CCAD},即相应 TCS 至少要在 TH_{CCAD}/I 次成功执行中偏向性达到 TH_{PGROW} 才会触发异常.




```
CCAD(headAddr, cmtaddr){
  entry= pCCADB+ INDEX(headAddr)
  if(entry->addrp== cmtAddr{
    entry->cntp += i;
  }else{
    entry->cntp -= D;
    if(entry->addrs== cmtAddr){
      entry->cnts ++;
    }else{
      entry->addrs= cmtAddr;
      entry->cnts= 1;
    };
  if(entry->cntp>= TH_CCAD){
    raiseException(CCAD);
  }else if(entry->cnts> entry->cntp){
    entry->addrp= entry->addrs;
    entry->cntp= entry->cnts;
  };
}
```

(*b*)连续提交地址检测算法

图 3　连续提交地址检测缓冲

路径替换与增长的实现为:作废当前 TCS,跟踪从相应 HEA 起始的代码的下一次执行路径作为新 TU 中 Trace 的路径.路径截断的实现与之类似,但在跟踪程序执行时增加一个路径结束条件:源结构指令地址与导致本次 TCS 失败执行的分支指令地址(*cancelXferAddr*)相同.这样所得路径将是原 TU 中从 HEA 到导致失败执行的分支指令之间的部分,亦即原先预测的热点路径与本次失败执行路径的公共部分.

6 性能模拟与分析

我们在 ULDBT 系统中对 ATUC 算法进行了详细模拟与性能比较.模拟采用 SPECint2000 中的 7 个基准测试程序进行,输入数据使用 SPEC 的 test 输入数据集.基准程序的编译使用优化开关 -O2,以及编译选项 -msoft-float 以避免生成浮点代码.在模拟中,使用翻译缓存大小为 16M 字节,热点入口阈值 TH_{HEA} 为 16,路径替换的执行成功率初始阈值 TH_{PREP} 为 0.2,路径截断的执行成功率阈值 TH_{PCUT} 为 0.8;连续提交地址检测阶段长度 TH_{CCAD} 为 32,路径增长偏向性阈值 TH_{PGROW} 为 0.8.

6.1 DBT 系统性能

我们比较了 NET、NETR、ATUC 和 BLOCK 四种 TU 构造算法的性能.NET 指为每个 HEA 采用 NET 算法预测热点路径作

为TU. NETR(NET with Retranslation) 在 NET 算法基础上, 检测到TCS 执行失败频率过高时通过 NET 重新预测热点路径并重新翻译. 这与 Dynamo[4] 中的算法类似, 只是 NETR 为每个 HEA 进行重翻译判断, 更为精细. 实验中 NETR 使用与 ATUC 相同的路径替换阈值. BLOCK 指以单个基本块为 TU.

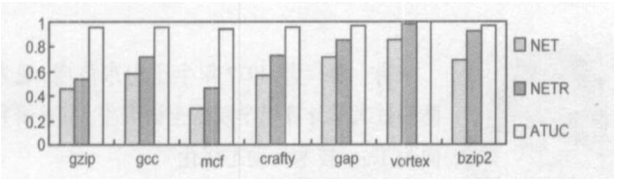


图 4　TCS执行成功率

图4 给出TU 构造算法对 TCS 执行成功率的影响. 图中未示出 BLOCK 情况, 因为其TCS 中不存在失败执行出口. 对于除 vortex 之外的基准程序, ATUC 相比 NET 和 NETR 在 TCS 执行成功率上都有显著提高. 对于所有的程序, 在 ATUC 算法下的执行成功率均超过 94%; 而在 NET 和 NETR 算法下程序 mcf、gzip、gcc、crafty 中的 TCS 执行成功率都低于 70%.

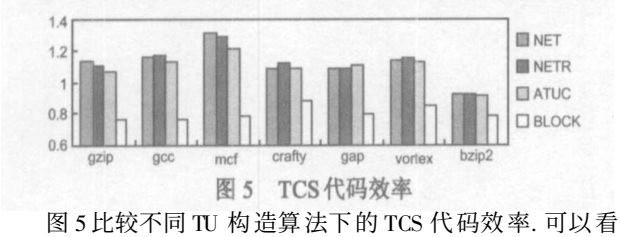


图 5　TCS代码效率

图 5 比较不同 TU 构造算法下的 TCS 代码效率. 可以看出, 不同的程序中 ATUC 算法对代码效率的影响不同. 与 NET 算法相比, mcf 中代码效率降低 8%; 对 gzip, gcc, crafty, vortex, bzip2, 下降在 1% 到 4% 之间; 而对程序 gap, 代码效率有 1% 的提升. 对此的解释是, ATUC 通过缩短 TU 提高 TCS 执行成功率, 同时减少了代码优化机会, 引起代码效率略微降低; 但在 ATUC 算法下 BTS 所检测到的热点入口与 NET 算法下并不相同, 因此会出现代码效率与 TU 长度不成正比的情况. 而 BLOCK 算法的代码效率则明显偏低, 相比 ATUC 算法下降在 14% (bzip2) 到 37% (gcc) 之间.

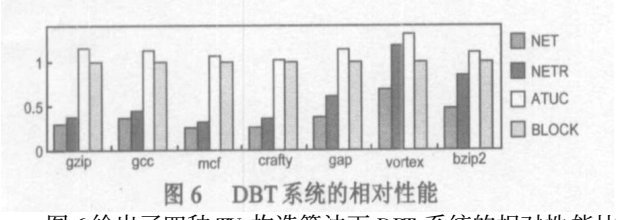


图 6　DBT系统的相对性能

图 6 给出了四种 TU 构造算法下 DBT 系统的相对性能比较, 以 BLOCK 算法的性能为参考. DBT 系统的执行时间由第 3 节中的公式给出. 参考文[3]中的数据, 取 t_{interp}为 30 个时钟周期, t_{trans}为 500 个周期; t_{vliw}为 1.2 个周期, t_{prof}为 20 个周期. 可以看出, ATUC 算法具有明显的性能优势. 相比 BLOCK 算法, 性能提升平均为 11.9%; 最高达 31% (vortex). 这表明 BLOCK 算法低下的代码效率对系统性能的影响严重. 另外, NET 和 NETR 算法下系统的性能明显低很多. 这也说明, 适用于动态优化系统的NET 算法并不适应于 DBT 系统, 因为在动态优化系统中, 未被优化执行的指令可以通过本机硬件直接执行而不需引入解释执行的高开销.

6.2　Profile 开销

ATUC 中 profile 时间开销有两部分. 在TCS 失败执行的处理中, 需要计算 TCS 的执行失败频率, 并检测路径替换与路径截断条件. 图 4 表明 ATUC 中TCS 的失败执行率很低, 从而降低了TCS 失败执行处理过程的调用次数与相应的开销. 对于成功执行, "长"TU 的TCS 不引入 profile 开销; "短"TU 的连续提交地址偏向检测由硬件 CCADB 完成. CCADB 进行的操作可与系统其它部分并行, 不会引起 HPC 中指令执行的延迟, 可以忽略其 profile 时间开销.

在空间开销上, ATUC 对每段 TCS 维持执行次数和失败次数两个计数. 在实现中当 TCS 的执行次数达到 128 时将这两个计数减半以降低历史信息的影响, 这两个计数各需一个字节. 另外, 对每段TCS 除入口地址之外还需要保存其 TU 的路径信息, 包括条件分支结果历史 16 位, 以及两个间接分支目标各 32 位. 因此, ATUC 算法对系统的空间开销仅为每段 TCS 需要 1 字节× 2+ 2 字节+ 4 字节× 2= 12 个字节.

ATUC 所需的硬件支持开销与 CCADB 的大小及组织相关. 在 CCADB 的实现中, 候选提交地址可以用相对于 TU 入口地址的偏移来表示. 在二进制翻译处理器中为处理自修改代码每个TU 都在同一物理内存页面之内[9], 因此偏移可以用 12 位表示. 主计数占用 8 位, 辅计数占用 4 位. 故每项需要 12× 2+ 8+ 4= 36 位即可. 实验系统中采用了 1K 项的 CCADB, 所需硬件开销仅为 4.5K 字节. 这些硬件实现开销相对于目前的电子设计工艺而言是很低的.

7　结束语

翻译单元的构造对动态二进制翻译系统的性能有着重要影响. 本文提出的自适应翻译单元构造算法 ATUC 在硬件 CCADB 的支持下实现动态适应程序执行行为, 保证翻译后代码的执行成功率与代码效率, 同时仅引入很低的 profile 开销. 实验模拟表明 ATUC 是一种有效的翻译单元构造算法. 下一步的工作包括对 ATUC 中各阈值对系统性能的影响作进一步的研究分析, 以及考虑二进制翻译过程中更多的优化因素对代码效率的影响.

参考文献:

[1] K Ebcioglu, E R Altman. Daisy: dynamic compilation for 100% architectural compatibility[A]. The 24th annual ISCA[C]. New York: ACM Press, 1997. 26- 37.

[2] C Zheng, C Tompson. PA-RISC to IA-64: Transparent execution, no recompilation[J]. Computer, 2000, 33(3): 47- 52.

[3] E Altman, M Gschwind, et al. BOA: The architecture of a binary translation processor[R]. New York: IBM Research Center, 2000.

[4] V Bala, E Duesterwald, S Banerjia. Dynamo: A transparent dynamic optimization system[A]. The ACM SIGPLAN 2000 Conference on PLDI[C]. New York: ACM Press, 2000. 1- 12.

[5] E Duesterwald, Vasanth Bala. Software profile for hot path prediction: less is more[A]. The 9th International Conference on ASPLOS[C]. New York: ACM Press, 2000. 202- 211.

[6] T Ball, P Mataga, M Sagiv. Edge profile versus path profile: The show-

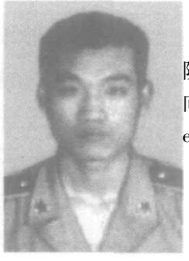
down[A]. The 25th ACM SIGPLAN-SIGACT Symposium on PPL[C]. New York: ACM Press, 1998. 134-148.

[7] Q Jacobson, E Rotenberg, J E Smith. Path-based next trace prediction [A]. The 30th Annual ACM/IEEE International Symposium on Microarchitecture[C]. Washington: IEEE Computer Society, 1997. 14-23.

[8] 曹宏嘉, 俞磊, 等. 一个用户级动态二进制翻译系统的设计与实现[J]. 计算机工程与科学, 2004, 26(8): 79-82.

[9] E J Kelly, R F Cmelik, M J Wing. Translated memory protection apparatus for an advanced microprocessor[P]. US patent: US006199152B1, 2001-03-06.

作者简介:



曹宏嘉 男, 1977 年生于河北无极, 现为国防科技大学计算机学院博士研究生, 主要研究方向为动态二进制翻译与优化. E-mail: hjcao@nudt.edu.cn.

唐遇星 男, 1977 年生于云南昆明, 现为国防科技大学计算机学院博士研究生, 主要研究方向为微处理器体系结构与编译优化技术.

肖 勇 男, 1977 年生于山东济南, 现为国防科技大学计算机学院博士研究生, 主要研究方向为 Trace 技术与动态优化.

邓 鹏 男, 1976 年生于湖南长沙, 博士, 现为国防科技大学副研究员, 主要研究领域包括体系结构与编译技术.

周兴铭 男, 1938 年生于上海, 中科院院士, 教授, 博士生导师, 主要研究领域包括高性能计算机体系结构, 微处理器, 移动计算等.