第 35 卷 第 5 期 计 算 机 工 程 2009 年 3 月
Vol.35 No.5 Computer Engineering March 2009
·软件技术与数据库· 文章编号：1000—3428(2009)05—0040—04 文献标识码：A 中图分类号：TP311

基于四阶段人工优化的软件流水技术

周国建[1]，吴少刚[1,2]，李祖松[2]，史 岗[2]
(1. 中国石油大学(华东)计算机与通信工程学院，东营 257061；2. 中国科学院计算技术研究所微处理器中心，北京 100080)

摘 要：代码体积是优化存储资源有限的嵌入式系统的重要因素之一。针对该特点，使用 oprofile 性能分析工具，以 EEMBC 基准程序集作为工作负载，提出四阶段人工优化软件流水方法(FPMO)。电信类的自相关程序实验结果表明，FPMO 以 2.04%的代码增量为代价换来 40.678%的性能提升，而单纯的编译器自动优化则以 33.35%的体积膨胀换来 38.33%的性能提升。

关键词：软件流水；循环展开；性能分析；四阶段人工优化

Software Pipelining Technique Based on Four-Phase Manual Optimization

ZHOU Guo-jian[1], WU Shao-gang[1,2], LI Zu-song[2], SHI Gang[2]
(1. College of Computer & Communication Engineering, China University of Petroleum, Dongying 257061;
2. Microchip Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080)

【Abstract】For embedded systems with very limited memory resources, code size becomes one of the most important optimization concerns. Using the oprofile profiling tool, this paper focuses on the Four-Phase Manual Optimization(FPMO) for the software pipelining technique when running the EEMBC benchmark. Experimental result of telecom-autocorrelation shows the FPMO method gets 40.678% performance promotion by increasing 2.04% code size but the pure compiler automatic optimization trades 38.33% performance improvements by 33.35% code size expansion.

【Key words】 software pipelining; loop unrolling; performance analysis; Four-Phase Manual Optimization(FPMO)

1 概述

基于龙芯处理器的嵌入式系统设计，不仅要结合应用需求，评估 CPU 的性能、功耗、面积、成本等关键要素，而且还要考虑软件的可移植性与优化问题。软件优化是时间与空间之间的权衡取舍，软件流水循环展开是提高性能的一种常用方法，通过增加代码的体积以空间资源消耗换取时间效率。

在软件流水中应用循环展开可以降低程序的资源需求和关键路径的长度。循环展开将循环体中的指令复制多份，增加循环体的代码量，减少了循环的重复次数，降低了循环转移的开销。展开后的循环体包含更多的指令，使调度器能更好地选择不相关的指令进行调度，从而提高执行效率。但是展开会引起代码量增长和寄存器需求增大，代码量的增长会导致缓存的性能变差[1]，寄存器需求的增大则有可能使软件流水失败。

本文针对单纯的编译器优化存在的优化后代码体积迅速膨胀的问题，提出四阶段人工优化的软件流水(Four-Phase Manual Optimization, FPMO)方法，兼顾了性能与代码体积膨胀之间的平衡。本实验从标准的嵌入式测试程序 EEMBC 中选取应用来验证 FPMO 方法效果的有效性，具有很好的代表性。

2 相关工作

循环展开通常在软件流水调度之前进行，即根据最优的启动间距(Initiation Interva1, II)来确定展开因子 k。如果在软件流水时无法得到启动间距为 II 的调度，则增加 II 继续调度，这样，原来计算的展开因子 k 可能就不是最优的了。文献[2]提出一种在二维空间(II, k)搜索展开因子的方法，先将二元组(II, k)按吞吐率的降序排列，再依次进行调度。该方法需要重复进行循环展开和模调度，计算复杂度较高。文献[3]提出针对完美多重循环展开因子的选择算法，此算法并没有考虑循环展开对软件流水的影响，只是单纯地从循环展开的角度来计算展开因子。

文献[4]提出基于程序特性的展开因子算法(Unrolling Times Based Program Characteristics, UTBPC)，解决了展开因子的确定问题，同时提出基于展开的软件数据预取优化技术，提高了软件数据预取的效率。

尽管目前对编译器的软件流水已有较多编译优化方法和算法，但是编译器自动优化往往造成对一些不是热点的循环也要进行展开操作，导致代码段体积膨胀，限制了其在嵌入式等对体积严格要求领域的应用。本文提出的四阶段人工优化软件流水方法能较好地兼顾程序性能与代码体积。实验选择了 EEMBC 中的电信类测试程序，经过优化后，性能有了 40.678%和 40.350 9%的提高，而代码段大小只增加了 2.04%。

3 四阶段的人工优化软件流水方法

利用 gprof 或 oprofile 工具进行原始程序运行结果统计分析，找出程序的热点(瓶颈)，实际情况可能为：一个非常热

基金项目：国家“863”计划基金资助项目(2007AA01Z114)；国家“863”计划基金资助重点项目“低成本先进计算机单机”(2006AA010201)；国家自然科学基金资助项目(60703017)

作者简介：周国建(1980－)，男，硕士研究生，主研方向：嵌入式系统性能分析优化；吴少刚，副教授、副研究员、博士；李祖松，博士；史 岗，副研究员、博士

收稿日期：2008-07-14 **E-mail：**zhouguojian@ict.ac.cn

点的运行时间大概要占到总的时间 90%以上，或多个均匀热点，热点分布比较平均。图 1 概括了整个优化流程。

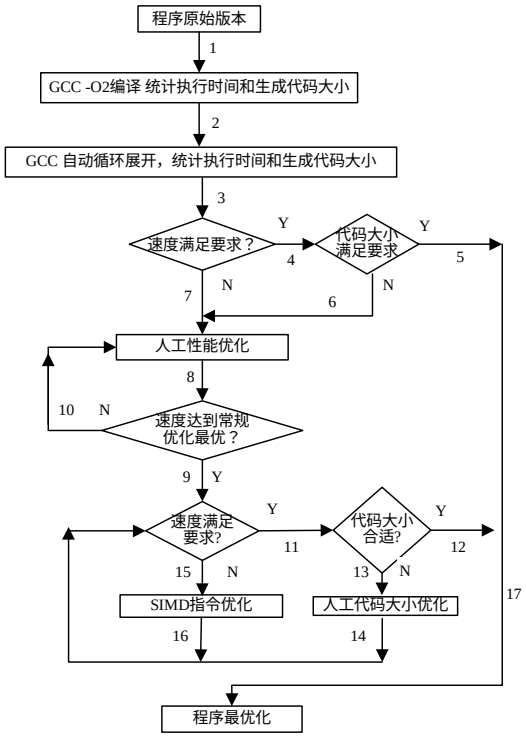

图 1　四阶段人工优化方法流程

(1)第 1 阶段：图 1 中的 1, 2, 3, 4 和 5。利用编译器自动优化选项进行优化分析，如果性能或者代码大小受限，则转入第 2 阶段。

(2)第 2 阶段：图 1 中的 6, 7, 8, 9 和 10。须根据上一阶段的反汇编信息，先进行一步粗粒度(2 的幂次)循环展开，利用折半法快速定位一个热点区间，找到重点优化区间，再针对该区间进行细粒度循环优化，找出一个性能最优值。直接转入第 3 阶段。

(3)第 3 阶段：图 1 中的 11, 12, 13 和 14。如果第 2 阶段求的最优值能满足性能要求，就要进行代码大小判断，如果超出允许范围，可以降低展开次数。再判断，同时如果程序中存在多个分布相对均匀的热点，而第 1 热点展开对代码体积增加太多，就可以从第 2、3 等热点同时选择几个对代码大小影响小的进行展开优化。根据对 EEMBC 测试程序组的统计分析发现，前 6 个热点就占整个程序的运行时间 95%以上，也有一些程序第一热点就能占 99%以上的运行时间，除此之外的非热点对程序的性能影响较小，不用进行展开。因此这一阶段的循环优化一般会在最多 $C_6^1+C_6^2+C_6^3+C_6^4+C_6^5+C_6^6=63$ 种组合测试中找到最优展开因子。如果经过这种优化后运行时间和代码仍然达不到要求，则要跳转到第 4 阶段。

(4)第 4 阶段：图 1 中的 15 和 16，在上述一些简单的手动优化达不到性能和代码大小要求时，就需要分析程序热点的原因和特点，使用 SIMD 指令增加并行处理能力，在有效地提高性能的同时还能降低代码要求，但是这对程序员提出了更高的要求。

4　实验平台

4.1　测试环境

本实验选用主频为 660 MHz 的龙芯 2E 处理器，它的一级 Cache 由 64 KB 的指令 Cache 和 64 KB 的数据 Cache 组成，片上二级 Cache 大小为 512 KB，均采用 4 路组相联的结构。操作系统为 Debian Linux(内核版本为 2.6.18)，编译器为 gcc 4.1.2，编译缺省优化选项统一为-O2，自动循环展开优化选项为-funroll-loops。

4.2　嵌入式测试程序 EEMBC

EEMBC 选择的基准程序是实际嵌入式应用中的真实算法。EEMBC 一般使用 2 种测评方法 标准型的(out-of-the-box)和全定制型的(full-fury)。标准型方法允许用户使用各类代码编译器选项设置，但不允许改变测试程序的源代码，主要目的是测试处理器和编译器优化的能力。与之相反，全定制型方法则允许修改测试程序的代码，用户可以从性能最优的角度考虑，使用汇编、专用库函数、硬件加速程序等各种可能的有效手段，目的是测试该处理器的最大性能指标。

实验选择了 EEMBC 电信类的一个自相关程序作为研究对象。其核心算法由 2 个嵌套循环构成并且实际的算法是由内层循环执行。这个测试程序能够揭示 CPU 在嵌套循环的过程中执行乘法、参数移位和加法操作的能力。

4.3　性能分析工具 Oprofile

在程序运行时，处理器内部会产生各种事件，如 Cache 是否命中、TLB 地址转换是否成功等。这些事件是程序行为在处理器内部的体现，对程序性能有直接的影响[5]。如果能统计程序的这些性能数据，并进行深入分析，将对程序的性能分析有着重要的意义。

龙芯 2 号内部集成了 2 个性能计数器，通过软件可以设置计数初值以及计数的条件，当计数溢出时，将触发一个中断。根据统计规律，程序运行过程中发生的事件越多，则触发的中断也越多。Oprofile 正是利用了这一基本规律，对程序进行采样分析。

5　性能优化与分析

5.1　编译器自动优化

通过使用 Oprofile 的分析工具 Opreport 可以准确分析整个程序的运行时间分布，实验结果如下：

```
CPU: ICT GODSON2, speed 661.725 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Cycles outside of haltstate) with a unit mask of 0x00 (No unit mask) count 30000
samples        %          symbol name
17734        99.9944      fxpAutoCorrelation
1             0.0056      t_run_test
```

从上述对执行时间的分布情况可见，99.994 4%的程序都被函数 *fxpAutoCorrelation* 消耗了，其主要操作就是内循环的 2 次取数和偏移操作。最简单有效的方法就是循环展开，让编译器针对处理器的具体结构进行一些指令级的优化工作。

经过 gcc 自动循环优化(-funroll-loops)性能得到了大幅提升，数据集 2 提高了 38.33%，数据集 3 提高了 37.93%，由于数据集 1 样本太小，运行时间较短，没有太大的性能变化。利用反汇编工具得到程序的热点展开了 15 次。代码段长度增长了 33.35%，这对于硬件资源敏感的嵌入式系统是不能容忍的。因此，需要进行下一阶段的粗粒度的人工优化。

5.2　人工优化

使用 Oprofile 分析工具 Opannotate 可以得到程序各条语句所执行的精确时间(为了兼顾采样准确性和效率，实验选取了 CPU 每 30 000 个时钟周期进行一次采样)：

```
(1)                    : /* Compute AutoCorrelation */
(2)      24    0.1353 : for (lag = 0; lag < NumberOfLags; lag++) {
```

```
(3)                 :Accumulator = 0;
(4)                 :LastIndex = DataSize - lag;
(5)                 :LastIndex_z = LastIndex - LastIndex % STEP_SIZE;
(6)   8011   45.1706 : for (i = 0; i < LastIndex_z; i++) {
(7)   9699   54.6885: Accumulator += ((e_s32) InputData[i] * (e_s32) InputData[i+lag]) >> Scale;
(8)                 : }
```

经过分析发现，程序 99.859%以上的时间被标号为(6)~(8)行这个内循环所消耗了。可重点对内循环进行人工性能优化。

在进行粗粒度(2 的幂次)循环展开得到的执行时间(程序实际测量 5 000 次的期望)如图 2 所示。

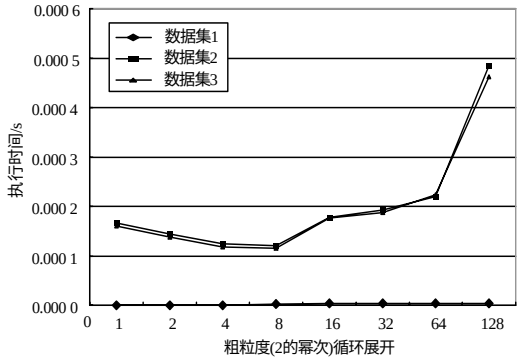

图 2　粗粒度循环展开次数与平均运行时间关系

由图 2 可以看出，在循环展开的过程中，运行时间呈现抛物线形状。在测试过程在 8 次循环展开时出现了一个局部最优点。尽管 15 次落在了[8, 16]区间内，但是从图中曲线走势可以看到[4, 8]区间也有可能出现全局最优点，因此，要对[4, 16]区间进行细粒度展开，如图 3 所示。

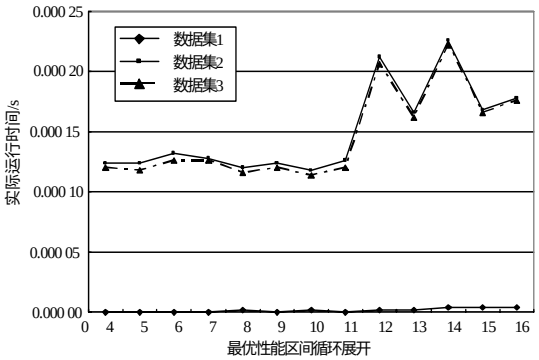

图 3　细粒度循环展开次数和平均运行时间关系

在图 3 中，最优点出现在 10 次展开循环时，结果与预期的最优点 15 次左右出入较大。因为龙芯 2 号处理器的 Cache Line 为 32 Byte，而程序的输入数据集 InputData[]是短整形的(2 Byte)，每次内循环要从 InputData 中取的 2 个数中间还相差一个偏移因子 *lag*，所以程序大部分时间(当 *lag*≥16 时)是跨多个 Cache Line 进行取数操作，按照理论值应该是 16 次能比较充分利用 Cache 里的数据，这与编译器自动循环展开优化的 15 次基本吻合。但图 3 的实际运行结果显示手动展开 16 次速度反而降了 7.2%~10%，因此，须考虑程序的结构和编译器优化后的汇编代码之间的关系。

对该程序的最热点进行指令级分析，找到各条指令对应的 CPU 时间消耗：

```
                    :for (i = 0; i < LastIndex_z; i++) {
                    :401514:    blez  t4,401570  <fxpAutoCorrelation+0x80>
                    :401518:    move  t3,zero
                    :40151c:    move  t1,a0
        3   0.0169 :401520:    move  t0,t6
                    :401524:    move  t2,zero
                    :Accumulator += ((e_s32) InputData[i] * (e_s32) InputData[i+lag]) >> Scale;
     7309  41.2123 :401528:    lh    v0,0(t1)
                    :40152c:    lh    v1,0(t0)
     1983  11.1813 :401530:    addiu t2,t2,1
                    :401534:    mult  v0,v1
     5987  33.7581 :401538:    addiu t1,t1,2
                    :40153c:    addiu t0,t0,2
     2390  13.4762 :401540:    mflo  v0
                    :401544:    srav  v0,v0,t5
                    :401548:    bne t2,t4,401528 <fxpAutoCorrelation+0x38>
                    :40154c:    addu  t3,t3,v0
       38   0.2143 :401550:    sra   v0,t3,0x10
                    :401554:    addiu t7,t7,1
```

上述汇编代码经过 GCC 编译器，根据龙芯 2 号的结构特征的优化形成，代码里有 4 条热点指令，分别是 401528:lh；401530:addiu；401538:addiu；401540:mflo。其中，第 1 个取半字操作(lh)占用时间长(41.2%) 是因为循环每次都要取回来一个新的 Cache line，访存的等待时间长。而 401530 处的无符号加操作(addiu)占用的时间长，主要是因为 Reorder 队列满了，而前面的 2 条取半字操作等待时间长，所以提交(Commit)等待时间也随之增加。401538 处的无符号加操作是因为前面的乘操作的时间较长，而且与开始的那条 lh 指令存在 WAR 相关，导致 Commit 等待时间长。401540 处的将整数乘法单元结果移到通用目的寄存器的操作(mflo)也是因为等待前面的乘法操作，且占了较长的时间。

由图 4 可见，10 次循环展开大部分事件数目都有明显的减少，像指令 Cache 缺失率、分支指令数目、跳转指令误预测率、TLB 重填及读内存等延迟大的事件都比较少，达到了整体性能最优。

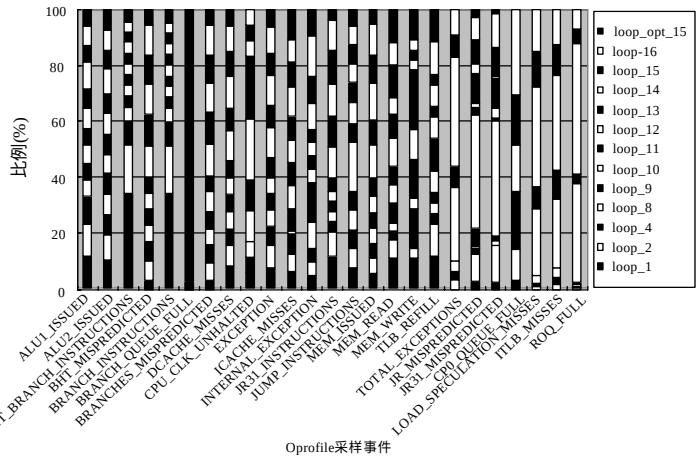

图 4　细粒度展开情况下各种采样事件百分比堆积柱状图

5.3　实验结果分析

当展开次数少于 10 次时，分支指令的执行次数较多，开销较大，同时也没有充分利用每次取到的 Cache Line 里面的

数据，编译器的优化效果不明显，导致指令级并行性没有充分发挥出来。

利用 opannotate 分析发现当手动展开次数超过 10，会增加堆栈操作的指令，增加处理器访存的压力。自相关程序主要是对定点数进行操作，龙芯 2 号有 32 个通用寄存器，但函数调用过程中可用的只有 26 个，展开 1 次要取 2 个半字，因此，在进行 11 次循环展开时需要 22 个通用寄存器保存内存取回的数只剩下 4 个通用寄存器可以保存 4 条运算指令的中间结果，为了优化调度，gcc 编译器把一些取回来的数，再压入堆栈中，直到运行到这条指令时再从栈中取回放到寄存器，原本只要对内存进行一次访问就能取到数，“优化”变成了 2 次读内存操作和 1 次写内存操作，由于没有足够的寄存器，许多中间结果还是需要多增加 1 次压栈和 1 次出栈操作。手动循环展开的次数超过 10 次后，展开得越多，性能下降得就越厉害。循环展开 10 次，虽然只利用一个 Cache Line 的 5/8，但是它正好能够提供 20 个通用寄存器保存这些值，剩下的 6 个寄存器可以保存中间结果，既发挥了指令的并行性，同时也使分支指令数目降到很低的程度，因此，获得了最优的性能。

5.4 进一步的优化研究

本程序存在一个非常热点(>99%)，优化相对简单，在经过了粗粒度循环展开和细粒度循环优化后，性能已经提升了 4 成多，代码段体积只增加了 2%，已经能够满足电信类的嵌入式应用要求了。如果程序再变复杂一些，存在多热点的情况，这时 Oprofile 提供的信息可以更好地选择热点进行优化。如果这些还是不能达到要求，最后需要采用处理器提供的 SIMD 指令来加速优化。而自相关程序中是计算乘、移位之后再进行累加法操作，龙芯 2 号目前提供的 SIMD 指令不具备该操作功能，因此，该程序不适合 SIMD 指令优化。假如程序算法稍微变化一下，核心循环只是一个乘累加运算，就可以充分利用额外的 2 套浮点部件和浮点寄存器堆，性能会在目前优化的基础上翻倍，达到原来的 3 倍左右，而且代码量会减少。

6 结束语

循环展开是软件流水的一种重要手段，在编译器中已经得到很好的实现，但是编译器的智能有限，当添加了循环优化选项后会把所有的循环按它认为最优的方式展开，很多非热点循环也被展开了，结果性能的提升不明显，但却使代码段的体积迅速膨胀，会对那些资源敏感的嵌入式应用程序的优化产生更大的副作用。本文提出的四阶段人工优化方法，在有效提高性能的同时，尽可能地把代码段的膨胀效应考虑进优化的整个过程中。在 FPMO 的第 3 阶段中，如出现均匀热点(平均占据 10%的时间)超过 10 时，就要从至少 $C_{10}^{1}+C_{10}^{2}+C_{10}^{3}+C_{10}^{4}+C_{10}^{5}+C_{10}^{6}+C_{10}^{7}+C_{10}^{8}+C_{10}^{9}+C_{10}^{10}=1\ 023$ 种组合测试中找到最优展开因子，这给人工优化带来了沉重负担。因此，下一步的研究方向是改进策略，单独展开每个热点，求出它们的性能体积因子(⊿性能/⊿体积)，将问题转换为背包问题，再利用一些贪婪算法可以很快求解出最优组合。

参考文献

[1] Zhuge Qingfeng, Xiao Bin. Code Size Reduction Technique and Implementation for Software-pipelined DSP Applications[J]. ACM Trans. on Embedded Computing Systems, 2003, 2(4): 590-613.

[2] Sanchez F, Cortadella J, Badia R M. Optimal Exploration of the Unrolling Degree for Software Pipelining[J]. Journal of Systems Architecture: the EUROMICRO Journal, 1999, 45(6): 1-16.

[3] Sarkar V. Optimized Unrolling of Nested Loops[C]//Proc. of the 14th International Conference on Supercomputing. New Mexico, USA: [s. n.], 2000.

[4] 李文龙, 刘 利, 汤志忠. 软件流水中的循环展开优化[J]. 北京航空航天大学学报, 2004, 30(7): 1112-1114.

[5] 张福新. 微处理器性能分析与优化[D]. 北京: 中国科学院研究生院, 2005.

编辑 顾姣健

（上接第 36 页）

参考文献

[1] 刘昕鹏. Ontology 理论研究和应用建模——Ontology 研究综述[EB/OL]. (2004-02-01). http://gis.pku.edu.cn/ Resources/TR/.

[2] 刘升平. OWL Web 本体语言指南[EB/OL]. (2005-04-08). http://www.transwiki.org/cn/owlguide.htm.

[3] Xu Zhuoming, Cao Xiao, Dong Yisheng, et al. Formal Approach and Automated Tool for Translating ER Schemata into OWNL Ontologies[C]//Proc. of Advances in Knowledge Discovery and Data Mining. Berlin, Germany: Springer, 2004: 464-475.

[4] 于长锐, 王洪伟, 蒋 馥. 基于逆向工程的领域本体开发方法[J]. 计算机应用研究, 2006, 23(6): 28-30.

编辑 张 帆

（上接第 39 页）

当需要更新教育机器人主控芯片中一个用户空间的程序时，首先由软件平台产生用户程序代码，然后通过串口与主控芯片中监控程序通信，监控程序利用多用户程序在线编程技术将新的用户程序代码下载到指定的用户空间。

6 结束语

多用户程序在线编程技术是建立在单用户程序在线编程技术的基础上，对单用户程序在线编程技术的一种改进和提高[5]。

本文以 MC9S12DG128 芯片为例，重点阐述多用户程序在线编程中的关键技术，给出一个教育机器人开发平台的应用实例，对嵌入式系统中多用户程序在线编程研究具有一定的借鉴意义。

参考文献

[1] Morton T D. 嵌入式微控制器[M]. 严隽永, 译. 北京: 机械工业出版社, 2005.

[2] Freescale Inc.. Freescale MC9S12DG128 DataSheet[Z]. (2004-04-12). http://www.freescale.com.

[3] 邓宗明, 蒋祺明. 基于 PowerPC 开发板的 Flash 编程方法与实现[J]. 计算机工程, 2004, 30(1): 168-170.

[4] Motorola Inc.. M68HC12&HC12 Microcontrollers CPU12 Reference Manual[Z]. (2002-04-16). http://www.freescale.com.

[5] 王宜怀. 嵌入式应用在线编程开发系统的研制[J]. 计算机工程, 2002, 28(12): 22-24.

编辑 金胡考