一种面向嵌入式系统总线的低功耗优化方法

葛红美[1,2] 徐 超[1,2,3] 陈 念[1,3] 廖希密[1,3]

(武汉大学计算机学院 武汉 430072)[1] (徐州工业职业技术学院信息管理技术学院 徐州 221000)[2]

(武汉大学软件工程国家重点实验室 武汉 430072)[3]

摘 要 为了解决嵌入式系统设备总线的功耗问题,从软件方面的功耗优化入手,提出一种面向嵌入式系统总线的低功耗优化方法,即在编译阶段,分别对指令地址总线和数据总线进行优化,以减少总线的翻转次数,降低其功耗。具体方法为:针对指令地址总线,采用改进后的遗传算法进行函数段调用优化,然后结合T0编码,减少总线翻转次数,从而降低其功耗。针对指令数据总线,采用粒子群算法进行指令调度优化,然后结合0-1翻转编码,减少总线翻转次数,从而降低其功耗。为了验证上述方法的正确性和有效性,以HR6P系列微处理器为平台展开实验,实验结果表明,总线功耗的优化效率达到25%左右。该方法明显减少了总线的翻转次数,提高了系统的整体性能。

关键词 低功耗,编译优化,地址总线,数据总线

中图法分类号 TP309.1 **文献标识码** A

Low Power Optimization Method Oriented to Embedded System's Bus

GE Hong-mei[1,2] XU Chao[1,2,3] CHEN Nian[1,3] LIAO Xi-mi[1,3]

(School of Computer, Wuhan University, Wuhan 430072, China)[1]

(Information Management Institute, Xuzhou College of Industrial Technology, Xuzhou 221000, China)[2]

(State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)[3]

Abstract This paper put forward a method which can optimize power consumption of bus in embedded system on the base of software optimization. In the process of compiling, if instruction address bus and data bus can be optimized respectively to reduce the bus-invert frequency, the power consumption will be cut down. The concrete procedure is as follows. For instruction address bus, using the modified genetic algorithm to optimize function call and combinating TO code, bus-invert frequency will be reduced and the power consumption will be reduced as well. For instruction data bus, using particle swarm algorithm to optimize instruction scheduling and combining 0-1 bus-invert code, bus-invert frequency will be reduced and the power consumption will be reduced as well. In order to verify the correctness and effectiveness of the method, HR6P serial microcontroller is used as an experiment platform. The experiment result shows that optimization efficiency of bus power consumption can reach about 25%. Therefore, it means the method obviously reduces bus-invert frequency and improves the whole system's performance.

Keywords Low power, Compiler optimizations, Address bus, Data bus

1 引言

随着嵌入式系统设备快速的发展,芯片集成度越来越高,系统应用越来越复杂,使得功耗问题成为嵌入式系统必须面对的一个关键问题。传统的功耗研究一般都针对硬件,从硬件设计和散热两个方面来降低系统的功耗;然而,随着功耗优化要求的进一步提高,单纯的硬件功耗优化已经不能满足要求;在受到软件性能、空间优化的启发后,人们提出了基于软件的功耗优化,并取得了很好的成效。

在嵌入式系统设备中,总线是信号和数据传输的通道,所有设备之间的通信都需要通过总线,而总线每次传输的数据对应一种传输状态,当传输数据不变的时候,总线电路状态也保持不变,也就不需消耗动态功耗,当传输数据发生变化的时候,则需要引起电路状态的变换,从而产生较大的动态功耗。由此可以看出,总线传输的数据同电路功耗有很强的关联性,其功耗主要是由相邻时钟脉冲传输数据的翻转产生的。如何有效地减少总线的翻转次数,来有效地降低系统功耗。因此,针对总线的功耗优化是低功耗优化的一个非常有意义的研究方向。

以减少总线的翻转次数为目的,可以从两个方面来实现,一是使用低功耗指令及低功耗调度,不同指令对于功耗的要求是不同的,尽可能地使用低功耗指令自然可以减少总线功

到稿日期:2013-04-19 返修日期:2013-07-22 本文受国家自然科学基金重点项目(91118003),国家自然科学基金面上项目(61170022),江苏省高校"青蓝工程"优秀青年骨干教师培养对象资助。

葛红美(1982—),女,讲师,主要研究方向为软件工程、嵌入式系统等;徐 超(1980—),男,博士,副教授,主要研究方向为软件工程、并行分布式处理、可信软件、嵌入式系统等,E-mail:xuch@whu.edu.cn(通信作者)。

耗；另一方面，使用低功耗的指令编码来降低系统功耗，影响总线功耗的因素主要是相邻指令之间的变化，那么使用适当的编码，减少相邻指令的反转次数，自然可以减少功耗。使用低功耗的指令调度就是通过改变指令的存储位置或者执行顺序来减少相邻指令之间的相互作用效应带来的功耗，从而减少功能部件的电路翻转。

本文对于总线功耗的优化就是从上述的两个方向出发的，在编译阶段，一方面分别对数据总线和地址总线进行低功耗的指令调度，同时辅以适当的总线编码，最终实现总线翻转次数的减少，从而达到降低总线功耗的目的。本文第 2 节介绍了本文的目标实验平台 HR6P 系列处理器的体系结构；第 3 节详细介绍 0-1 翻转编码和粒子群算法的数据总线优化方法以及 T0 编码和改进的遗传算法的地址指令调度方法的实现过程；第 4 节是实验结果与分析；最后是相关工作和总结。

2 实验平台体系结构概述

本文以 HR6P 系列微处理器为平台展开研究。海尔 HR6P 系列芯片为精简指令集(RISC)MCU。该系列芯片是在自有微处理器架构的基础上，经过创新设计而成。同时它又能够方便客户使用，客户可以进行全新的设计，或是对原有系统进行改进，也可以非常轻松地替代。该系列芯片核心是一个采用两级流水线的哈佛型结构 CPU，程序和数据总线相互独立。指令存储体数据字宽为 16 位，几乎所有指令都是单字指令，能在一个机器周期内从总线取出。指令集为 48 条，编码效率高，容易进行指令扩展，如图 1 所示。

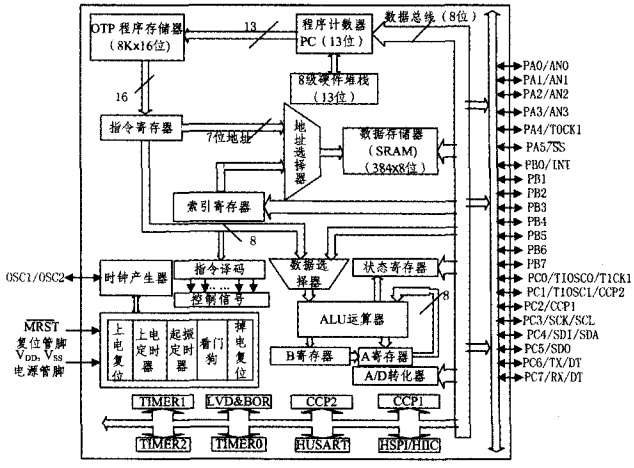

图 1 HR6P 系列某芯片结构框图

3 面向嵌入式系统总线的低功耗优化方法

3.1 面向指令数据总线的优化方法

数据总线主要用于数据信息的传送，是双向三态形式的总线，具有数据随机性大的特点。本文提出一种面向指令数据总线的优化方法。首先针对每一个基本块进行数据依赖分析，得到分析结果，然后采用粒子群算法和 0-1 翻转编码进行优化，从而得到优化后的目标代码。具体描述如图 2 所示。

(1)数据依赖关系分析

数据总线的低功耗优化的主要思想是减少总线的翻转次数，这是与相邻指令的机器码密切相关的，所以要通过研究指令间的相互关系，来找到最有利于优化的指令执行次序。因此必须首先构建算法 1 所示的数据依赖关系图(DAG)。

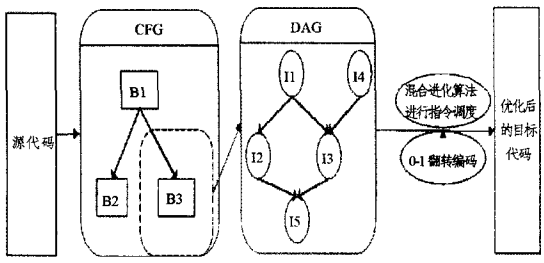

图 2 数据总线优化模型

算法 1

Input：加权基本块控制流图 WCFG

Output：基本块内的 DAG 图

```
1. foreach i in B;//基本快内的指令 i
2.      tmp=m. insMap[i. opName];//包含该指令的定义使用信息
3. foreach s in tmp. uses;//分析使用情况
4.      if(s. Equals("op");
5.   if 该数据的定义在定义映射表中存在;
6.      将该语句的定义作为该语句的父语句;
7.   else 该数据的定义在定义映射表中不存在;
8.      不做处理;
9.      Endif
10.      endif
11. endfor
12. foreach s in tmp. defs;//分析定义情况
13.      if(s. Equals("op");
14.        if 该数据的定义在定义映射表中存在;
15.          替换原有定义语句;
16.        else 该数据的定义在定义映射表中不存在;
17.          增加新的定义语句;
18.      endif
19.   endfor
20. endfor
```

(2)基于 0-1 翻转编码和粒子群算法的指令调度优化算法

问题描述：基本块内一共有 N 条指令，指令间的数据依赖关系用图 DAG 表示，要求在符合该 DAG 图的数据关系的情况下，找出能得到最小总线翻转次数的指令调度和编码方法。下面是模型映射及主要数据说明

①顺次编号每条指令为 $1-N$；

②$1-N$ 的一个排列表示一个粒子的位置，即一个粒子对应一种指令调度序列，如$(1,2,3,\cdots,N)$；

③粒子的速度表示为一个链表 $V=\{\langle m_1,n_1\rangle,\langle m_2,n_2\rangle,\cdots,\langle m_k,n_k\rangle\}$，链表中每个元素$\langle m_2,n_2\rangle$表示交换序号 m_k 和序号 n_k 的指令的位置；

④速度元素系数 $k_{v(m,n)}$ 表示元素$\langle m,n\rangle$ 在速度 v 中出现的次数；

⑤ 速度平均系数如式(1)所示：

$$avg(v)=\frac{\sum_i k_{v(m_i,n_i)}}{\|v\|} \tag{1}$$

式中，$\|v\|$ 表示 v 中不同元素的个数；

⑥速度的加法使用如式(2)所示：

$$V_1 \oplus V_2=\{\langle M,N\rangle \mid K_{(M,N)} \geqslant avg(v_1+v_2)\} \tag{2}$$

式中，v_1 表示 v_1 链表和 v_2 链表的简单链接，以保留主速度为宗旨，选择速度原始系数大于等于速度平均系数的速度元素；

⑦系数与速度的乘法如式(3)所示：

$$cv=\begin{cases}0, & c=0\\ (m_k,n_k),k=1,2,\cdots,\lfloor \|cv\| \rfloor, & c\in(0,1]\\ v+c'v, & c>1,c=x+c',x\in N,c'\in(0,1)\end{cases} \quad (3)$$

⑧位置减法 x_1-x_2 表示从 x_2 变化到 x_1 对应的速度 v；

⑨位置与速度的加法 $x_{k+1}=x_k+v_{k+1}$，表示从 x_k 通过 v_{k+1} 中所列出的变换方式变换到 x_{k+1}；

⑩速度位置变换如式(4)所示：

$$v_{k+1}=c_0v_k\oplus rand()(p_{i,k}-x_k)\oplus rand()(p_{g,k}-x_k) \quad (4)$$

$$x_{k+1}=x_k+v_{k+1} \quad (5)$$

⑪目标函数如式(6)所示：

$$\min f(x)=\sum_{i\in x}\tilde{l}_{i,i+1}+\alpha * l_{\max} \quad (6)$$

且 $\alpha=0$，其中 $\tilde{l}_{i,i+1}$ 表示从第 i 条指令到第 j 条指令编码后的跳变数，其值与 x 的序列相关，不同 x 序列其值不同，α 表示 x 中违背约束(即数据依赖关系 DAG 图)的位置数，$l_{\max}$ 为一个较大值，以使得一旦该 x 序列违背 DAG 图中的约束，程序就能自动淘汰该种调度方法。

⑫0-1 翻转编码，设总线宽度为 n，当 $\tilde{l}_{i,i+1}\leqslant\frac{n}{2}$ 时，$\tilde{l}_{i,i+1}$ 不变，否则 $\tilde{l}_{i,i+1}=n-\tilde{l}_{i,i+1}$。

(3)具体求解如算法 2 所示。

算法 2

```
1. for 每个粒子 i
2. 随机生成 xi, vi
3. 设 pi :=+∞
4. endfor
5. g :=+∞
6. iterNumber :=0
7. do
8. pi :=+∞
9. for 每个粒子 i
10.   p :=根据式(6)计算适应值
11.   If p<pi then
12.     pi :=p
13.     xbest,i :=xi
14.     if p<g && α=0
15.         g :=p
16.       xgBest :=xi
17.     endif
18.   endif
19. endfor
20. for 每个粒子 i
21.   根据式(4)计算每个粒子的速度
22.   根据式(5)计算每个粒子的位置
23. endfor
24. iterNumber++;
25. while iterNumber<MAXNUM
26. return xgBest
```

3.2 面向指令地址总线的优化方法

地址总线主要用于查找程序指令在内存中的位置，它大部分程序采用顺序访问，除在跳转和函数调用处，地址总线上的数据基本是以固定步长增加的。本文提出一种面向指令地址总线的优化方法。首先对指令进行控制流图分析，然后根据控制流图，在函数内部对于顺序执行的，采用 T0 编码，尽可能减少函数内部的翻转次数，在函数调用处，利用改进后的遗传算法，通过调整函数在内存中的存放位置来进行函数段调用优化。其操作流程如图 3 所示。

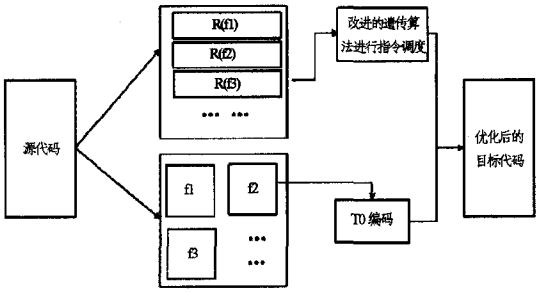

图 3 指令地址总线优化模型

在图 3 中，$f1$，$f2$，$f3$ 分别代表不同的函数，$R(f1)$，$R(f2)$，$R(f3)$，$R(f4)$是为了方便函数调用自己定义的描述函数间调用关系的表示方法。

(1)函数调用依赖关系分析

指令地址总线的翻转次数与指令的存放地址密切相关，需要弄清楚函数之间的调用关系，然后根据函数大小、函数调用关系等信息分配函数的存储位置，从而找到最优的分配方式。因此，构建函数调用依赖关系如算法 3 所示。

算法 3

```
1. foreach (s in CA. functions. Values)
2. //构建函数直接调用次数
3.   Func tmp=CA. functions[s];
4.     if s 不在调用当前函数的函数表中
5.       将 s 插入到调用当前函数的函数列表中
6.       将当前函数放在被 s 调用的函数列表中
7.         else
8.             当前函数被 s 调用的次数加 1
9.             s 调用当前函数的次数加 1
10. endif
11. endfor
12. 设置初始结点的权值 1
13. bool flag=true;//标记程序终止条件
14. while (flag)
15. flag=false;
16.     foreach 列表中的函数 fc
17.     foreach 被 fc 调用的函数
18.         tmp=fc 所有父节点调用次数之和
19.         if fc. Wr!=tmp
20.           flag=true;
21.       endif
22.           fc. Wr=tmp
23.       endfor
24.       endfor
25. endwhile
```

(2)基于 T0 编码和改进的遗传算法的地址指令调度方法

①编码：在函数的存储顺序问题中，对所有的函数依次编号，从 0 开始记录，则 n 个函数分别表示为 0 到 $n-1$，则这 n 个数的任何一个排列即为该问题的一个可行解，而该可行解可以用一位数组表示。

②种群初始化：根据函数个数 n 和种群规模 N，随机产生 N 个 0—($n-1$)的全排列(为了程序书写方便，数组下标从 0 开始，函数下标也从 0 开始，0 表示标号为 0 的函数)。随机

产生 0—(n−1)的全排列，具体如算法 4 所示(其中 array 为一个可行解，即函数排列的数组)。

算法 4

```
1. foreach 0≤i<N
2. foreach 0≤j<n
3.     随机产生一个小于 j 的非负整数 irand
4.     array[I,j]和 array[I,irand]互换
5.     Endfor
6. endfor
```

③评估：是根据适应值函数计算每个个体的适应值。当前适应值即为当前分配方式下总线翻转的次数，假设函数个数为 n，初始化的时候共产生 N 个个体，第 k 个个体的排列表示为$[t_0,t_1,\cdots,t_{n-1}]$，则根据这个排列以及每个函数的大小，我们就可以得到每个函数的实际地址，然后根据调用关系，则可得到适应值函数，如式(7)所示：

$$f(k)=\sum_{i\in F}\sum_{j\in C_i}\sum_{k=p_{i,j}+Hamdis(begin_j+size(j),begin_i+offset_{i,k}+sizeIns(Call))}^{N_{i,j}\times(Hamdis(begin_i+offset_{i,k},begin_j))} \tag{7}$$

式中，F 表示程序中所有可供调用的函数的集合；C_i 表示所有被函数 i 调用的函数的集合；$P_{i,j}$ 表示函数 i 中所有调用函数 j 指令的程序点位置的集合；$begin_i$ 表示函数 i 在程序中起始位置；$offset_{i,k}$ 表示程序点 k 相对函数 i 起始位置的偏移地址；$size(j)$ 表示函数 f 占用空间的大小；$sizeIns(Call)$ 表示函数调用语句占用空间的大小；$Hamdis(m,n)$ 表示整数 m 和 n 之间的哈明距离；N_{ij} 表示函数 i 在程序点 k 处的执行次数。

④父本选择：本文采用轮盘赌方式进行父体选择。一个轮盘被划分为 N 个扇形，每个扇形表示种群中的一个染色体，而每个扇形的面积与它所表示的染色体的适应值成正比。为了选择种群中的个体，设想有一个指针指向轮盘，转动轮盘，当轮盘停止后，指针所指向的染色体被选择。因此一个染色体的适应值越大，表示该染色体的扇形面积就越大，因此它被选择的可能性也就越大。其具体过程如下：

1. 计算种群中所有个体适应值之和：

$$Total=\sum_{i=0}^{n-1}f(i) \tag{8}$$

2. 计算每个个体的选择概率：

$$nSelPro(k)=\frac{f(k)}{Total}(k=0,1,\cdots,n-1) \tag{9}$$

3. 计算每个个体的累计概率：

$$nToPro(k)=\sum_{i-0}^{k}nSelPro(i)(k=0,1,\cdots,n-1) \tag{10}$$

4. 转动轮盘 N 次。用[0,1]中的一个随机数 r 来模拟转动一次轮盘、轮盘停止转动后指针所指向的位置。若 $r\leqslant nToPro(0)$，说明指针指向第一个扇形，这时选择第一个个体，一般若 $nToPro(k-1)\leqslant r\leqslant nToPro(k)$，说明指针指向第 k 个扇形，这时选择第 k 个染色体。具体如算法 5 所示。

算法 5

```
1. parantselect
2. foreach i
3.    使用式(9)计算个体 i 的选择概率
4. Endfor
5. foreach i
6.    使用式(10)计算个体 i 的累计概率
7. Endfor
8. Forench i
9.    随机生成浮点数 randdouble
10.      j=0
11.   While j<N 且 randdouble 大于 j 的累计概率
12.      j++
13.   endwhile
14.      选择第 j 个个体
15. Endfor
```

⑤杂交：选择部分映射杂交。即通过从一个父体中选择一个子序列，并尽可能多地保持另一个父体中函数的次序和位置的方式产生后代。例如有两个个体：$P1$=[2,6,4,7,3,5,0,1]，$P2$=[4,5,2,1,0,7,6,3]，随机产生的两个杂交点 $pos1$=3 和 $pos2$=6，则杂交后可得到 $O1$=[2,3,4,1,0,7,6,5]，$O2$=[4,1,2,7,3,5,0,6]。其具体流程如图 4 所示。

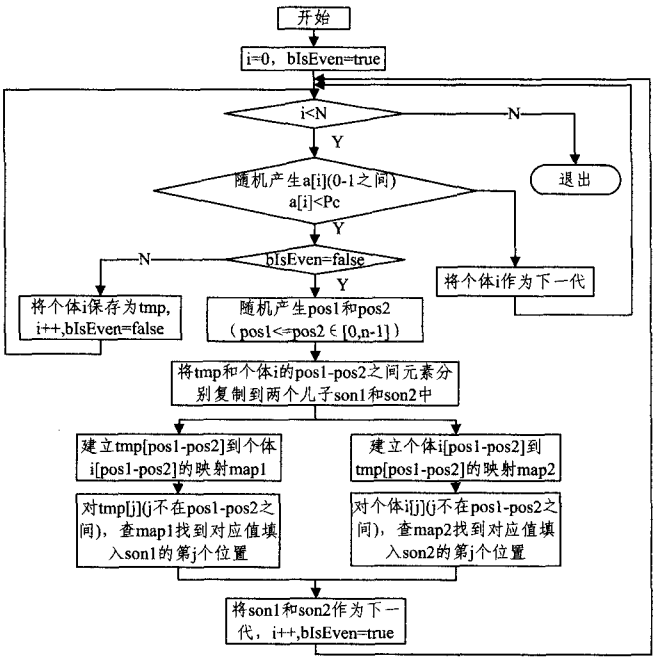

图 4　杂交流程图

其中，bIsEven 用来标识是否已选择出两个用来杂交的父本，Pc 是决定当前个体是否杂交的一个临界值，在实际操作时，我们会针对每一个个体随机生成一个 0-1 之间的数，当这个数值大于 Pc 时，该个体直接作为下一代，否则参与杂交。在杂交过程中，先随机生成两个在 0 到 N−1 之间的整数 $pos1$ 和 $pos2$，假设参与杂交的两个个体分别叫做个体 i 和个体 j，而这两个个体产生的后代分别叫 $son1$ 和 $son2$，则先分别将 i 个体的 $pos1-pos2$ 之间的值赋给 $son2$，将 j 个体的 $pos1-pos2$ 之间的值赋 $son1$。然后分别建立一个 $i[pos1-pos2]$到 $j[Pos1-pos2]$的映射 $map1$，一个 $j[pos1-pos2]$到 $i[pos1-pos2]$的映射，然后根据查询 $map1$ 补齐 $son1$ 中的元素，查询 $map2$ 补全 $son2$ 中的元素，$son1$ 和 $son2$ 就是杂交后得到的下一代。

⑥变异：在解决该功耗优化问题时选取的变异算子为倒位变异。倒位变异是指首先在父体上随机地选择两个点，然后将这两个点之间的子序列反转。假设现在有一个体 P=[0,1,2,3,4,5,6,7]，随机产生两个变异点 $pos1$=3 和 $pos2$=6，则经过变异后产生的个体 O=[0,1,2,6,5,4,3,7]。当然，在变异之前，我们同样会给定一个变异临界值并随机生成一个 0-1 之间的随机数，如果随机数大于该临界值，则不进行变异，否则进行变异，其实现的具体流程如图 5 所示。

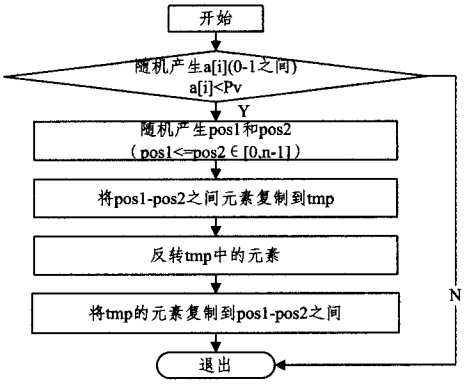

图 5　变异算法

⑦停止条件：由用户输入迭代次数 Num，每次产生下一代种群之前判断计算次数是否达到 Num，若没有达到，则继续进行遗传迭代，每次迭代完之后将每次迭代后的个体的最优解与之前最优解进行判断，取二者的更优者作为全局最优解，直到达到迭代次数 Num，退出返回全局最优解。

4　实验结果分析

为了测试该基于 0-1 翻转编码的粒子群指令调度算法对总线翻转次数的优化效果，我们以 MiBench 作为基准测试用例集，以 arm 作为目标体系结构，与 arm-linux-gcc 3.3.2 原始指令调度算法进行了对比实验。本实验的测试平台具体配置如表 1 所列。

表 1　测试平台

CPU	Intel Pentium 4 3.06G
操作系统	Windows XP sp3
内存	DDR3 内存，容量 2G
开发环境	Visual Studio 2010
目标芯片	合作公司的 RISC 芯片
测试用例	MiBench

从图 6 可以看出，该优化算法具有较快的收敛速度，能够在大概 1500 次迭代时获得较好的优化效果。

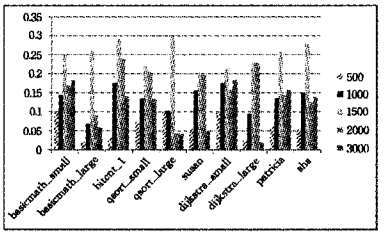

图 6　基于 0-1 翻转编码的粒子群指令调度算法对总线翻转次数的优化百分比

为了测试初始粒子数对优化效果的影响，我们将迭代次数为 1500 时不同数量的初始粒子数（粒子数分别为 5，10，15，20，25）对总线翻转次数的优化结果进行了测试，其结果如图 7 所示。

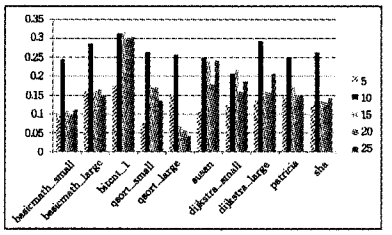

图 7　初始粒子数对优化效果的影响百分比图

由此可见，初始粒子数的大小对优化效果也有一定的影响，粒子数较少时优化效果并不十分明显，但过大的粒子数只会增加运行的开销，对优化效果的提升没有太大贡献。

通过以上实验可以看出，基于 0-1 翻转编码的粒子群指令调度算法对总线翻转次数具有较好的优化效果，其平均优化可以达到 25% 左右。而且其收敛速度较快，迭代次数在 1500 次左右即可获得较好的优化效果，粒子数开销也较低，使用的初始粒子数大概为 10 即可，因此可以以较小的时间和空间开销获得较好的优化效果，是降低总线功耗的一种可行而高效的优化方法。

同时，为了测试该基于 T0 编码和改进遗传算法的指令调度算法对指令地址总线翻转次数的优化效果，同样，以 arm 作为目标体系结构，与 arm-linux-gcc 3.3.2 原始指令调度算法进行了对比实验。

在图 8 中同样可以看出，该优化算法具有较快的收敛速度，能够在大概 2000 次迭代时获得较好的优化效果。

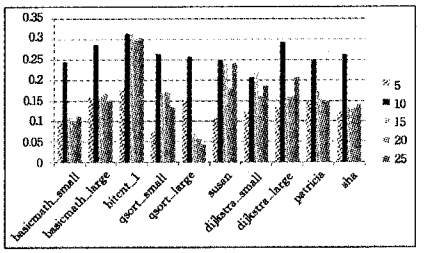

图 8　基于 T0 编码和改进遗传算法的指令调度算法对总线翻转次数的优化百分比

此外，为了测试历史队列长度和杂交系数对优化效果的影响，将迭代次数设置为 2000 次，变异因子设置为 0.05，对不同历史队列长度（队列长度分别为 1，5，10，15，20）和不同杂交系数值（杂交系数分别为 0.6，0.65，0.7，0.75，0.8）分别进行了测试，其结果如图 9 和图 10 所示。

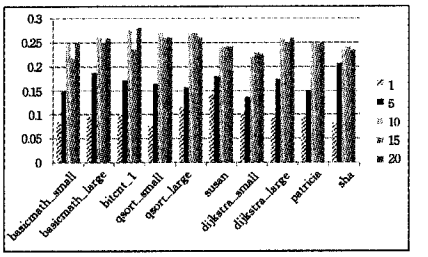

图 9　历史队列长度对优化效果的影响百分比图

从图 9 可以看出，增加的历史队列对遗传算法的优化效果有着重要影响，当历史队列长度增加到一定程度（如为 10）时，优化效果有显著的增加，而且此时并不会随着队列长度的进一步增加而提高优化率，因此该算法具有较好的收敛性。

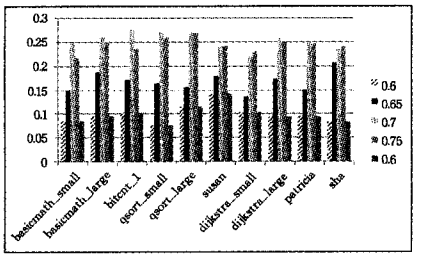

图 10　实验结果 5

对于杂交系数，从图 10 可以看出，不同的杂交系数对优化效果有较为明显的影响，过低的杂交系数由于交换区间过小，优化率也较低，因此，选择合适的杂交系数对于优化的改进十分重要，在该测试用例中，我们发现杂交系数为 0.7 时，平均优化率最高，可以达到 25%左右。

通过以上 3 个对比实验可以看出，不同的历史队列长度、不同的迭代次数以及杂交系数对该算法的优化效果均有不同的影响。当迭代次数为 2000、历史队列长度为 10、杂交系数为 0.7 时，能够获得较好的优化效果，平均优化率可以达到 25%左右。因此，为减少指令地址总线的功耗，基于 T0 编码和改进遗传算法的指令调度算法同样是降低总线功耗的一种可行而高效的优化方法。

结束语 在嵌入式系统功耗问题中，总线翻转次数是评估系统功耗的一个重要指标，很多学者对如何减少总线翻转次数进行了深入的研究，并针对不同的体系结构的特点，设计了各种方法[1-3]，以减少系统功耗。总的来说，这类方法主要分为两类：指令调度和总线编码。

在指令调度方面，Parikh A 等人总结了各种低功耗指令调度算法，并同性能优先指令调度算法进行了对比[4,5]，指出性能最佳的调度序列并不一定是功耗最低的调度序列[6]。LEE C 等人针对 VLIW 体系结构，提出了水平调度和垂直调度两种调度方法来减少总线翻转次数，以达到降低系统功耗的目的[7]。Shao Z 等人证明低功耗指令调度问题是 NP 完全问题，并在 VLIW 体系结构中针对总线翻转次数和调度长度[8]，提出了 3 种启发式调度算法，以尽可能获得较[9,10]低的功耗。

在总线编码方面，前人已经提出了很多有效的方法，如 Stan 和 Burleson 的总线翻转编码[11]、Lv T 等人的数据总线编码[12]、Benini L 等人的 T0 编码[13]、Mehta H 等人的格雷编码[14]等。最近，Hui Guo 等人针对地址总线，除去了格雷编码复杂的解码电路，而使用格雷码直接对地址总线进行编码，从而进一步减少系统功耗[15]。Suresh D C 等人为减少数据总线的功耗，提出了 VALVE 和 TUBE 两种新的编码方法[16-18]。这两种方法通过一条额外的控制信号线，对片外数据总线上可变长度连续和非连续重复位进行编码，以减少片外数据总线的翻转次数，达到降低功耗的目的。

本文主要研究了基于编译的总线低功耗优化方法。在分析了系统功耗模型以后，发现动态功耗是系统功耗的一个重要组成部分，而与动态功耗息息相关的总线翻转次数，正好是软件可控的。所以本文以此为突破点，分别用基于 0-1 翻转编码和粒子群算法的指令调度算法以及基于 T0 编码和改进遗传算法的函数段分配算法对数据总线和地址总线进行优化，通过减少总线的调转次数，达到降低系统功耗的目的。本文设计和实现了基于 0-1 翻转编码和粒子群算法的指令调度算法以及基于 T0 编码和改进遗传算法的指令调度算法，以 MiBench 作为基准测试用例集，以 arm 作为目标体系结构，分别与 arm-linux-gcc 3.3.2 原始指令调度算法进行了对比实验。结果表明，本文所采用的优化算法对于低功耗的优化是非常有效的。

参 考 文 献

[1] Jain V, Rele S, Pande S, et al. Code restructuring for improving execution efficiency, code size and power consumption for embedded DSPs[C]// 12th International Workshop on Languages and Compilers for Parallel Computing. 1999

[2] Stubbs, Clyde (HI-TECH Software LLC). Compilation strategies for low-power designs [J]. ECN Electronic Component News, 2008, 52(9): 29-30

[3] Stubbs, Clyde (HI-TECH Software). Compilation strategies: Alternate approaches to achieve low power consumption [J]. ECN Electronic Component News, n SUPPL., 2008(4): 11-13

[4] Jiang Xiang-tao, Hu Zhi-gang, He Jian-biao. Call chain analysis for low power compile optimization[J]. Journal of Jilin University (Engineering and Technology Edition), 2009, 39(1): 143-147

[5] Lorenz M, Leupers R, Marwedel P, et al. Low-energy DSP code generation using a genetic algorithm[C]// Proceedings-IEEE International Conference on Computer Design: VLSI in Computers and Processors. 2001: 431-437

[6] Parikh A, Kim S, Kandemir M, et al. Instruction Scheduling for Low Power[J]. The Journal of VLSI Signal Processing, 2004, 37(1): 129-149

[7] Lee C, Lee J K, Hwang T T. Compiler optimization on Instruction Scheduling for Low Power[C]// ISSS2000. 2000: 20-22

[8] Shao Z, Zhuge Q, Zhang Y, et al. Efficient Scheduling for Low-Power High-Performance DSP Applications [J]. International Journal of High Performance Computing and Networking IJHCN, 2004, 1: 3-16

[9] Lakshminarayana G, Raghunathan A, Jha N K. Incorporating Speculative Execution into Scheduling of Control-Flow-Intensive Designs[J]. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 2000, 19(3): 308-324

[10] Chung E-Y, Benini L, De Micheli G. Energy Efficient Source Code Transformation based on Value Profiling[C]// Iit wnrhhnp fw Cnmpilen and Opemtmng Syrtem fw Lnrn-Pmncr. 2000: 1-7

[11] Stan M R, Burleson W P. Bus-invert coding for low power I/O [J]. Microelectronics Reliability, 1996, 36(4)

[12] Lv T, Henkel J, Lekatsas H, et al. An adaptive dictionary encoding scheme for SOC data buses[M]. Design, Automation and Test in Europe Conference and Exhibition, 2002: 1059-1064

[13] Benini L, De Micheli G. System-Level Power Optimization Techniques and Tools[J]. ACM TODAES, 2000, 5(2): 115-182

[14] Mehta H, Owens R M, Irwin M J. Some Issues in Gray Code Addressing[C]// Proceedings of the 6th Great Lakes Symposium on VLSI. 1996: 178

[15] Gu Ji, Guo Hui. An 3fficient segmental bus-invert coding method for instruction memory data bus switching reduction[C]// EURASIP Journal on embedded Systems. 2009: 20-29

[16] Suresh D C, Agrawal B, Yang J, et al. Power Efficient Encoding Techniques for Off-Chip Data Buses[C]// Proc. of Compilers and Architecture and Synthesis for Embedded Systems (CASES). San Jose, CA, Oct. 2003

[17] Julien N, et al. Power consumption modeling and characterization of the TI C6201[C]// IEEE Micro. 2003: 40-49

[18] Lorenz M, et al. Compiler based Exploration of DSP Energy Savings by SIMD Operations[C]// Asia South Pacific Design Automation Conference C. Yokohama, Japan, 2004: 838-841