基于扩展控制流图的片上存储器分配策略

王学香, 浦汉来, 杨 军
(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: 本文提出一种基于扩展控制流图(ECFG)的片上存储器(Scratch-Pad Memory, SPM)分配策略, 该策略首先把程序划分为全局变量、全局堆栈、指令块等节点, 用包含节点和节点间关系的 ECFG 来描述应用程序, 接着采用考虑了节点间关系的改进的背包算法把选中的节点分配到 SPM 中. 实验表明该策略比采用单纯背包算法的 SPM 分配策略减少应用程序执行时间 11%, 比不使用 SPM 时减少 56%, 大大提高了 SoC 存储子系统的性能.

关键词: 片上存储器; 分配策略; 控制流图
中图分类号: TP343 文献标识码: A 文章编号: 0372-2112 (2007) 08-1558-05

Performance Oriented Allocation Scheme for Scratch-Pad Memory

WANG Xue-xiang, PU Han-lai, YANG Jun
(*National ASIC System Engineering Technology Research Center, Southeast University, Nanjing, Jiangsu 210096, China*)

Abstract: A SPM memory allocation method were proposed based on extend control flow graph. This method transforms the application into a directed graph consisting of nodes and relationships of nodes. In succession, this method applies a refined Knapsack algorithm to solve the problem of SPM memory allocation. In the previous researches, these relationships of nodes are ignored, which result in a considerable expense of memory space during the process of SPM allocation. Our experiments show that our approach conduces to significant performance improvements (11% an average) compared to the previous. And the execution time of the application is reduced to 56% compared to none SPM environment.

Key words: scratch-pad memory; memory allocation scheme; control flow graph

1 引言

随着系统芯片(SoC)设计水平的不断提高, 芯片的高主频速度与片外存储器的低读取速度不相匹配, 很大程度上限制了芯片整体性能的提升. 研究人员提出了许多技术来提高 SoC 存储子系统的性能, 其中之一便是 SPM. 与由硬件管理的 Cache 相比, 采用适当的软件优化方法管理 SPM 时, 针对一些具体应用程序, 可使 SPM 的性能、功耗和面积都优于 Cache[1,2], 适合用于对于实时性要求较高的嵌入式系统. 由于访问 SPM 的时间比访问片外存储器的时间要小得多, 而且不存在命中率的问题, 所以把程序的一部分从片外存储器搬移到 SPM 中可以大大减少应用程序的运行时间.

SPM 分配策略的研究分为两类: 一类是在 Cache 和 SPM 共存的情况下, 研究如何利用 SPM 减少 Cache 冲突次数[3~6], 这类研究仅考虑把数据变量放入 SPM, 并未考虑函数. 另一类是在无 Cache 的情况下, 研究如何把最频繁访问的数据放到 SPM 中来减少片外存储器的访问次数, 这类研究又分为两种: (1) 成组数据访问优化[7~9]; (2) 程序访问优化[10~14]. 前者将成组数据作为唯一的分析对象, 而后者分析的是全部程序, 即使后者有时只分析程序中的数据, 它也包括了全部的数据变量, 而不仅仅是成组数据. 本文研究无 Cache 情况下基于程序访问优化的 SPM 分配策略, 即对于有限的 SPM 空间, 如何确定哪些程序内容存入其中, 从而最大限度的提高系统性能. 在这方面, Sjodin[10] 和 Oren[11] 提出的两种 SPM 分配策略仅考虑了程序中的全局变量和堆栈, 而没有考虑把部分函数放入 SPM 所带来的性能的大幅提升. Steinke[12] 把程序划分为全局变量、函数、函数基本块等节点, 接着采用背包算法把选中的节点分配到 SPM 中, 但是没有考虑把两个连续的函数基本块放入 SPM 对性能的影响. Steinke[13] 基于 Steinke[12], 考虑了把两个连续的函数基本块放入 SPM, 减少跳转指令的条数, 从而可以更充分利用 SPM 空间. Verma[14] 基于 Steinke[13], 对全局变量中的大型数组进一步进行划分, 从而大大降低了功耗, 但是同时会带来性能的降低.

前人的研究均没有针对 ARM 指令的特点考虑全局变量、堆栈、函数块等节点之间的关系, 虽然 Steinke[13] 考虑了两个连续的函数基本块, 但是过于简单, 未考虑指令节点和数据节点间的关系. 一些 ARM 指令对于地址空间有很严格的限制, 比如跳转指令(B/BL)和数据装载(LDR)指令. 通常情况下由于地址跳转范围过大, 一条跳转指令是不能在片外存储器和 SPM 之间跳转的. 片外存储器中的跳转指令搬到 SPM 中运行后原跳

收稿日期: 2006-11-06; 修回日期: 2007-04-17
基金项目: 国家自然科学基金(No. 60676011)

转指令应该由一个跳转宏来代替,跳转宏包含两条指令,一条跳转指令和一条数据装载指令.同样,SPM 中的一些数据装载指令通常需要额外的空间来保存操作地址.上述情况导致了搬到 SPM 中的节点大小的增加.但是如果选择了两个相互关联的节点,由于长跳转宏和存储数据装载指令的操作地址的数量减少了,SPM 节点大小增加的总数就减少了.因此,忽略节点间的相互关系必然会带来 SPM 空间利用率的减少.针对前人的不足,本文提出了基于 ECFG 的片上存储器分配策略,首先把程序划分为全局变量、全局堆栈、指令块等节点,用包含节点和节点间关系的 ECFG 来描述应用程序,接着采用考虑了节点间关系的改进的背包算法把选中的节点分配到 SPM 中.实验表明该策略比采用单纯背包算法的 SPM 分配策略减少应用程序执行时间 11%,比不使用 SPM 时减少 56%,大大提高了 SoC 存储子系统的性能.

2　基于 ECFG 的片上存储器分配策略

2.1　目标结构模型

为了得到应用程序的实时运行信息,本文用 SystemC 语言建立了一个 cycle 精度级的目标结构模型(图 1),包含四个部分:ARM7TDMI、SPM、SDRAM(片外存储器)、EMI(外部存储器控制接口).

为完成对程序执行过程中访问信息的收集,在 ARM7TDMI 模型、SPM 模型和 SDRAM 模型中添加了跟踪模块(Tracer).从而可以很容易得到应用程序的实时运行信息,包括:SDRAM 中每个地址(以四个字节为单位)的访问延迟、每条指令被执行的次数、每条指令实际产生程序跳转的次数、每个 SDRAM 地址的实际读写访问次数、SPM 中每个地址的访问次数等等,这些实时运行信息用在 ECFG 生成和计算存储子系统性能模型的过程中.

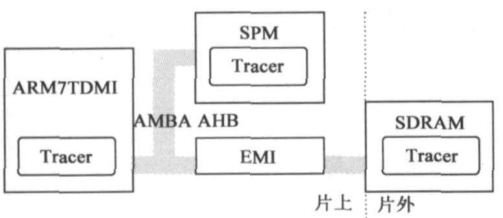

图 1　目标结构模型

2.2　ECFG 生成

为了把部分程序放入 SPM 运行,首先需要把应用程序划分为一系列节点.其中,应用程序中所有符合大小限制的函数按照其内部的跳转指令被切割成多个指令块;程序的全局堆栈被封装成一个全局数据变量.这样,应用程序就被划分为全局数据变量、全局堆栈和函数基本块等节点.这些指令节点和数据节点之间有着千丝万缕的关序,本文用 ECFG 来描述应用程序的节点和节点之间的关系.传统的 CFG 只能反映某个函数的内部结构,而忽略了函数和变量,以及函数之间的关系,ECFG 通过应用程序中所有 CFG 的合理组合,体现了函数和全局变量以及函数之间的关系.

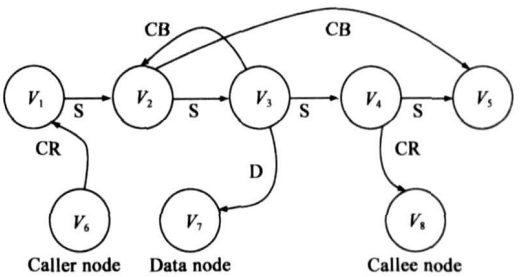

图 2　ECFG

ECFG 是一种有向图,由节点和边组成,图 2 示出了应用程序 ECFG 的一个局部视图.其中,节点用来表示各种程序内容,包括指令节点和数据节点.V_i 代表第 i 个结点;边表示节点之间的连线,即节点之间的关系,箭头发起端便是该关系的发起者,箭头指向端是该关系的接收者.在图 2 中,节点 V_1 被另一个函数的调用节点所调用,而节点 V_4 调用了一个被调用节点.节点 V_3 在执行的过程中访问了一个数据节点.用 $R_{type}(i,j)$ 表示 V_i 和 V_j 之间的关系,可知,边分为五种,见表 1.除了 $R_1(i,j)$ 以外,所有的边在图 2 中都有对应的图示.$R_3(i,j)$ 表示两个相连的指令节点之间的关系,即无跳转的顺序执行.

表 1　边的类型

边的类型	符号	说明
$R_1(i,j)$	UB	节点之间是无条件转移
$R_2(i,j)$	CB	节点之间是条件转移
$R_3(i,j)$	S	节点之间是顺序执行
$R_4(i,j)$	CR	节点之间是调用关系
$R_5(i,j)$	D	节点之间是数据访问

2.3　存储子系统性能模型

存储子系统性能模型用于计算把部分节点放入 SPM 后所带来的性能提升,同时为了使得节点大小不超过 SPM 总大小还需要计算出放入 SPM 的节点的总大小.

当数据节点或指令节点 V_i 在 SDRAM 中运行时,假设 V_i 对应的地址空间大小为 S_i,其中地址 n 的访问延迟记为 $T_{access\_delay}(n)$,那么 V_i 在 SDRAM 中的访问时间等于其地址空间内所有地址访问延迟的累计:

$$T_{sd}(V_i)=\sum_{n=1}^{S_i}T_{access\_delay}(n) \qquad (1)$$

当节点 V_i 搬入 SPM 后,假设其对应的地址空间内,每个地址的实际读写访问次数分别为 N_{core_rd} 和 N_{core_wr},而且 SPM 的读写操作访问时间分别表示为

T_{sp_rd}和T_{sp_wr}(两者都等于一个时钟周期),那么 V_i 被搬入 SPM 后,它的访问时间将变为:

$$T_{sp}(V_i) = N_{core\_rd} \times T_{sp\_rd} + N_{core\_wr} \times T_{sp\_wr} \quad (2)$$

这两个公式给出了节点 V_i 分别在 SDRAM 和 SPM 中的访问时间,$T_{sd}(V_i)$和 $T_{sp}(V_i)$之间的差值越大,表示该节点被搬入 SPM 后,程序性能提高的程度也越大. 但实际上,单纯依靠这两个数值来判断节点搬入 SPM 后对程序性能的改善,显得过于简单. 这种计算方法的前提是节点在搬移过程中大小和内容不会发生变化,而实际上在节点搬移过程中,由于 ARM 对于跳转指令和数据装载指令的地址空间的限制,会引入额外指令和 DCD 数据,或者修改了原指令,从而导致节点访问时间和节点本身的大小发生了变化. 表 2 列出了五种关系在搬移过程中节点的大小变化和访问时间的变化. 其中 ΔT_{start}表示关系发起者节点被搬入 SPM 后程序执行时间的变化;ΔT_{end}表示关系接收者节点被搬入 SPM 后程序运行时间的变化;ΔS 表示关系发起者节点搬入 SPM 后节点大小的变化.

表 2 各种节点间关系搬移到 SPM 后节点大小和节点访问时间的变化

关系	大小的变化	访问时间的变化
UB	$\Delta S = +4$	ΔT_{start}; ΔT_{end}
CB	$\Delta S = +8$	ΔT_{start}; ΔT_{end}
S	$\Delta S = +8$	ΔT_{start}; ΔT_{end}
CR	$\Delta S = +8$	ΔT_{start}; ΔT_{end}
D	$\Delta S = +4$	ΔT_{start}; 0

以数据访问关系(关系 D)为例,发起者节点是一个指令节点,接收者节点是一个数据节点. 一旦指令节点被放入SPM,必须要复制一份保存数据节点地址信息的 DCD 数据放入 SPM. 因此,指令节点搬入 SPM 后大小增加了 4 个字节(一个 DCD 数据),访问时间的减少 ΔT_{start} 为:一次读 SDRAM 和 SPM 操作之间的时间差,乘上指令节点的执行次数. 数据节点搬入 SPM 后大小不变,对程序的执行时间没有任何额外影响,故 $\Delta T_{end} = 0$.

本文用关系矩阵表示节点之间关系对程序性能的影响,包括性能矩阵 $\boldsymbol{TRM}_{ij}$和大小矩阵 $\boldsymbol{SRM}_{ij}$,分别表示当且仅有节点 V_i 被搬入 SPM 时,节点 V_j 与 V_i 之间所有关系对程序执行时间的影响和对节点大小的影响,用公式(3)表示:

$$TRM_{ij} = \sum_{type=1}^{5}(\Delta T_{start}(R_{type}(i,j)) + \Delta T_{end}(R_{type}(j,i)))$$
$$SRM_{ij} = \sum_{type=1}^{5}\Delta S(R_{type}(i,j)) \quad (3)$$

那么,当且仅有节点 V_i 被搬入 SPM 时,程序性能的提高值 $T(V_i,0)$和节点 V_i 搬入 SPM 后所占用的空间大小 $S(V_i,0)$可以用公式(4)计算,其中 $S(V_i)$指节点 V_i 原始大小:

$$T(V_i,0) = T_{sd}(V_i) - T_{sp}(V_i) - \sum_{j=1}^{n} TRM_{ij}$$
$$S(V_i,0) = S(V_i) + \sum_{j=1}^{n} SRM_{ij} \quad (4)$$

如果此前已经选中若干节点 I 搬入 SPM,则程序性能的提高值 $T(V_i, I)$和节点 V_i 搬入 SPM 后所占用的空间大小 $S(V_i, I)$可以用公式(5)计算:

$$T(V_i,I) = T(V_i,0) + \sum_{j=1}^{n} I(V_j) \times TRM_{ij}$$
$$S(V_i,I) = S(V_i,0) - \sum_{j=1}^{n} I(V_j) \times TRM_{ij}$$
$$I(V_j) = \begin{cases} 1, & V_j \text{ 被选中搬入 SPM} \\ 0, & V_j \text{ 仍处于 SDRAM} \end{cases} \quad (5)$$

2.4 SPM 分配算法

本文采用改进的背包算法把部分节点放入 SPM,如图 3 所示,步骤如下,与单纯的背包算法相比改进点主要在第(3)步:

(1)将所有节点按照各自的优先级排列. 节点 V_i 的优先级定义如下,

$$priority(V_i,0) = T(V_i,0)/S(V_i,0) \quad (6)$$

(2)把优先级最高的节点 V_1(前提条件是节点 V_1 的大小 $S(V_1) <$ SPM 的大小 S_{SPM})放入 SPM,$I(V_1)$置为 1,更新选中节点大小 TTS 和选中节点放入 SPM 后的性能提高值 TTT.

(3)重新计算在节点 V_1 放入 SPM 后各节点的优先级:

$$priority(V_i,I) = T(V_i,I)/S(V_i,I) \quad (7)$$

(4)根据此时各个节点的优先级,选择放入 SPM 的节点 V_i(前提条件是 $TTS + S(V_i,I) < S_{spm}$),$I(V_i)$置为 1,更新 TTS 和 TTT.

(5)重复第四步,直到 SPM 空间已经无法再放入任何节点,此时 TS 和 TE 分别为总的节点大小和程序性能提高值. 并且得到了选中放入 SPM 的节点列表 I.

```
TTS = S(V1, 0);
TTT = T(V1, 0);
for(i = 2; i < n, TTS < Sspm; i++){
  if(TTS + S(Vi, I) < Sspm){
    I(Vi) = 1;
    TTS = TTS + S(Vi, I);
    TTT = TTT + T(Vi, I);
  }
  else
    I(Vi) = 0;
}
TS = TTS;
TT = TTT;
```

图 3 改进的背包算法

2.5 重新链接

得到在当前 SPM 容量下被选中搬入 SPM 的节点列表 I 之后, 通过一个用 C 语言编写的链接器重新链接程序中所有节点, 并自动生成新的二进制文件. 新的二进制文件中包括三个部分:初始化部分, SDRAM 部分和 SPM 部分. SDRAM 部分和 SPM 部分都根据节点搬移情况修改了跳转指令. 新的程序将从初始化部分开始执行, 将 SPM 部分复制到真实的 SPM 空间内, 然后跳转到原程序的 *main* 函数首地址开始执行. 如果 *main* 函数首节点被选中搬入 SPM, 则初始化部分将直接跳到 SPM 对应位置开始执行.

3 试验环境建立

图 4 描述了本文的 SPM 分配策略的工作流程. 步骤如下:

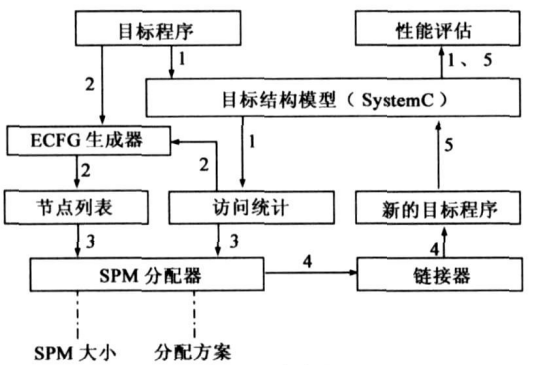

图 4 工作流程

(1) 将编译器生成的二进制目标程序送入目标结构模型中的片外存储器执行, 得出优化前程序的运行时间, 以及运行过程中处理器对片外存储器的实时访问统计信息.

(2) 将原程序的访问统计信息和二进制目标程序输入 ECFG 生成器, 把原程序划分成一系列节点和节点间的关系. 把节点列表传递给 SPM 分配器.

(3) SPM 分配器根据节点列表, 在给定的 SPM 大小情况下, 分别采用单纯的背包算法和改进的背包算法选择部分节点放入 SPM, 得到两种算法情况下放入 SPM 的节点列表.

(4) 根据选中放入 SPM 的节点列表, 链接器自动修改原目标程序, 生成新的二进制目标程序, 使选中节点被搬入 SPM 运行.

(5) 再次使用目标结构模型, 得到优化后的二进制目标程序的运行时间, 从而评估 SPM 分配策略对程序性能的优化能力.

4 实验结果

本文采用基准测试程序 Dijkstra(使用 Dijkstra 算法寻找最短路径)、Jpeg2bmp(将 160×120 大小的 JPEG 图片转换成 BMP 格式文件)、MP3player(解码 20 帧 MP3 数据)、Gunzip(解压缩 11K 字节大小的文件)、Bubblesort(冒泡排序算法) 和 CRC32(32 位 ANSI X3.66 CRC 文件校验和求解), 通过图 4 的工作流程得到它们在 SPM 大小为 2Kbytes 时全部在 SDRAM 中运行的运行时间 T (10^6 时钟周期)、采用 Steinke[12] 单纯背包算法的 SPM 分配策略的运行时间 T_k (10^6 时钟周期)、采用我们的策略的运行时间 T_o (10^6 时钟周期), 如表 3 所示, 其中:

$$R_{ok}=\frac{T_o-T_k}{T_k}\times 100\%,\ R_o=\frac{T_o}{T}\times 100\%$$

表 3 程序运行时间比较

测试程序	T	T_K	T_O	R_{OK}	R_O
Dijkstra	44.7	28	15.9	-43%	36%
Jpeg2bmp	191	141	121	-14%	63%
Gunzip	15.7	12.1	11.4	-6%	73%
MP3player	379	291	282	-3%	74%
Bubblesort	140	87.5	87.3	-0.2%	62%
CRC32	9.21	2.71	2.71	0	29%
平均				-11%	56%

从表 3 中可以看出, 我们的策略比不考虑节点间关系的单纯背包算法减少应用程序的执行时间 0~43%, 平均为 11%. 只有应用程序的大小和 SPM 的大小接近的情况, 如 CRC32 程序和 Bubblesort 程序, 两种策略对于应用程序执行时间的减少方面没有什么差别. 从表 3 中还可以看出, 采用我们的策略, 应用程序的执行时间比在 SDRAM 中运行平均减少了 56%.

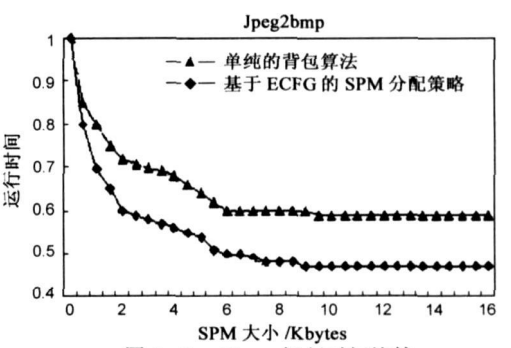

图 5 Jpeg2bmp:运行时间比较

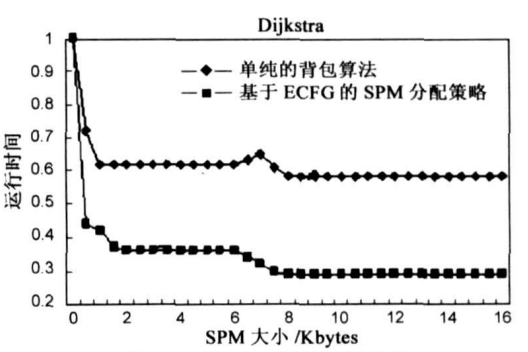

图 6 Dijkstra:运行时间比较

图 5 和图 6 分别示出了对于 Jpeg2bmp 程序和 Dijkstra 程序, 利用单纯的背包算法和我们的策略分别进行性能优化的比较结果. 图中横坐标表示 SPM 大小(Kbytes), 纵坐标表示采用单纯的背包算法和我们的策略把部分程序放在 SPM 中时程序运行时间与全部在 SDRAM 中运行时程序运行时间的比值. 从图 5 中可以看出, 我们的策略比单纯的背包算法平均提高程序性能 15% 左右, 与程序全部在 SDRAM 中运行相比, 平均提高程序性能 50% 左右. 从图 6 中可以看出, 使用单纯的背包算法应用程序的运行时间并不总是随着 SPM 大小的增加而减少, 原因是单纯的背包算法在 SPM 分配过程中把节点搬到 SPM 运行的那些指令所带来的开销大于节点本身在 SPM 中运行所带来的性能提升. 我们的分配策略通过在 SPM 分配过程中避免这种情况而带来了应用程序运行时间的大幅度减少. 另外, 还可以根据图 5 和图 6 中运行时间下降的曲线, 得到对于特定应用程序的最佳 SPM 容量, 以获得性能的最大提升.

5 结论

本文研究面向性能优化的 SPM 分配策略, 用 ECFG 把应用程序描述为全局变量、全局堆栈、函数基本块等节点和节点间的关系, 在 SPM 分配过程中采用考虑了节点间关系的改进的背包算法. 实验结果表明, 采用基于 ECFG 的 SPM 分配策略把应用程序的一部分从 SDRAM 中移到 SPM 中运行可以平均减少应用程序的运行时间 56% 左右; 与单纯的背包算法相比, 平均减少应用程序运行时间 11% 左右. 下一步的研究将会进一步分割数据变量和全局堆栈节点, 并且用更加复杂的测试程序来评估我们的 SPM 分配策略.

参考文献:

[1] Banakar R, Steinke S, Lee B, et al. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems [A]. Proceedings of the Tenth International Symposium on Hardware/Software CoDesign[C]. New York, USA: 2002. 73-78.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee. Comparison of Cache and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption [R]. Dortmund, Germany: 2001.

[3] P R Panda, N D Dutt, A Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems[J]. ACM Trans Design Automation of Electronic Systems, 2000, 5(3): 682-704.

[4] Panda P R, Dutt N D, Nicolau A. Local memory exploration and optimization in embedded systems[J]. Computer-Aided Design of Integrated Circuits and Systems, 1999, 18(1): 3-13.

[5] Ranjan Panda P, Dutt N D, Nicolau A, et al. Data memory organization and optimizations in application-specific systems[J]. IEEE Design & Test of Computers, 2001, 18(3): 56-68.

[6] Absar M J, Catthoor F. Compiler-based approach for exploiting scratch-pad in presence of irregular array access[A]. Proceedings of the conference on Design Automation and Test in Europe[C]. ICM, MESSE Munich, Germany, 2005. 1162-1167.

[7] Marteil F, Julien N, Senn E, et al. A complete methodology for memory optimization in DSP applications[A]. Euromicro Symposium on Digital System Design[C]. Rennes, France, 2004. 98-103.

[8] Issenin I, Dutt N. FORAY-GEN. Automatic generation of affine functions for memory optimizations[A]. Proceedings of the conference on Design, Automation and Test in Europe[C]. ICM, MESSE Munich, Germany, 2005. 808-813.

[9] Kandemir M, Kadayif I, Sezer U. Exploiting scratch-pad memory using presburger formulas[A]. Proceedings. The 14th International Symposium on System Synthesis[C]. Montreal, Quebec, Canada, 2001. 7-12.

[10] J Sjodin, B Froderberg, T Lindgren. Allocation of global data objects in on-chip ram[A]. In the Proceeding of Workshop on Compiler and Architectual Support for Embedded Computer Systems[C]. Washington DC, USA: 1998.

[11] O Avissar, R Barua, D Stewart. An optimal memory allocation scheme for scratch-pad based embedded systems[J]. ACM Transactions on Embedded Computing Systems, 2002, 1(1): 6-26.

[12] S Steinke, C Zobiegala, L Wehmeyer, P Marwedel. Moving program objects to scratch-pad memory for energy reduction [R]. Dortmund, Germany: 2001.

[13] Steinke S, Grunwald N, Wehmeyer L, et al. Reducing energy consumption by dynamic copying of instructions onto on-chip memory[A]. 15th International Symposium on System Synthesis[C]. Kyoto, Japan, 2002. 213-218.

[14] Verma M, Steinke S, Marwedel P. Data partitioning for maximal scratchpad usage[A]. Proceedings of the Asia and South Pacific Design Automation Conference [C]. Kitakyushu, Japan, 2003. 77-83.

作者简介:

王学香 女, 1972 年 9 月出生, 东南大学电子工程系教师, 在职博士研究生. 主要专业及研究方向: SoC 体系架构, SoC 存储子系统设计等. E-mail: wxx@seu.edu.cn

浦汉来 男, 1980 年生, 东南大学电子工程系博士研究生. 主要专业及研究方向: SoC 体系架构, SoC 存储子系统设计等.