SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories*

Amit Pabalkar, Aviral Shrivastava, Arun Kannan and Jongeun Lee

Department of Computer Science and Engineering,
Arizona State University, Tempe, AZ 85281
{amit.pabalkar,aviral.shrivastava,
arun.kannan,jongeun.lee}@asu.edu

Abstract. Many programmable embedded systems feature low power processors coupled with fast compiler controlled on-chip scratchpad memories (SPMs) to reduce their energy consumption. SPMs are more efficient than caches in terms of energy consumption, performance, area and timing predictability. However, unlike caches SPMs need explicit management by software, the quality of which can impact the performance of SPM based systems. In this paper, we present a fully-automated, dynamic code overlaying technique for SPMs based on pure static analysis. Static analysis is less restrictive than profiling and can be easily extended to general compiler framework where the time consuming and expensive task of profiling may not be feasible. The SPM code mapping problem is harder than bin packing problem, which is NP-complete. Therefore we formulate the SPM code mapping as a binary integer linear programming problem and also propose a heuristic, determining simultaneously the region (bin) sizes as well as the function-to-region mapping. To the best of our knowledge, this is the first heuristic which simultaneously solves the interdependent problems of region size determination and the function-to-region mapping. We evaluate our approach for a set of MiBench applications on a horizontally split I-cache and SPM architecture (HSA). Compared to a cache-only architecture (COA), the HSA gives an average energy reduction of 35%, with minimal performance degradation. For the HSA, we also compare the energy results from our proposed SDRM heuristic against a previous static analysis based mapping heuristic and observe an average 27% energy reduction.

Key words: Compilers, Code overlay, Static code analysis, Scratchpad memory.

1 Introduction

The first generation embedded systems were limited to fixed, single functionality devices like digital watches, calculators, coffee makers etc. Modern embedded systems have evolved into programmable, highly complex, multi-functionality devices including portable music players, gaming consoles, PDAs, GPSs and cellular phones. These systems must exhibit high performance while at the same time consume less power,

* This work was partially funded by grants from Microsoft, Raytheon and Startdust Foundation.

as they operate on battery. Design of such systems thus becomes extremely challenging due to multi-dimensional and stringent design constraints.

Modern embedded processors increase performance by employing memory hierarchies consisting of caches or scratchpads or both. Caches improve performance by exploiting the spatial and temporal locality in the application, without any changes to the application itself. However, these improvements are achieved through use of tag arrays, comparators and management logic which in certain processors like StrongARM, can consume more than 40% of the total power budget [5].

Scratchpad Memories (SPM) on the other hand are devoid of power hungry tag arrays and comparators. Compared to caches they consume less energy per access and occupy smaller on-chip area. While previous works have demonstrated that a SPM may require on an average 40% less energy and 34% less die area compared to a cache of same size [3], the compiler is now responsible for managing the SPM contents. This involves inserting explicit instructions in the program to move code or data between SPM and the main memory. A good technique for mapping the program contents onto SPM thus becomes very critical for efficiently utilizing the SPM with minimal runtime transfer overhead. Since code exhibits more locality than data, mapping code should provide us with more power reduction. Therefore, in this work we focus on mapping application code onto the SPM.

Most code mapping techniques for SPMs require profiling to find the optimal mapping of applications. Profiling however, limits their applicability, not only because of the difficulty in obtaining reasonable profiles, but also due to high space and time requirements to generate a profile. Instead, in this work, we use compile time static analysis to eliminate profiling and the overhead associated with it. Our static analysis is based on a new data structure, Global Call Control Flow Graph (GCCFG), which captures the function call *sequence* as well as the control flow information like loops and conditionals. Our GCCFG can give not only the execution counts (estimated from the control flow) but also the execution sequences of functions (from control flow, call graph, and call sequence). This makes GCCFG more precise than just a call or a control flow graph in modeling the runtime behavior of an application.

Traditional approaches for SPM utilization breaks down the SPM mapping problem into two smaller problems. The first problem, termed as *memory assignment* or *'what to map'* involves partitioning the application code into SPM mapped and main memory spilled. This division eliminates code segments whose cost of transfer from memory to SPM is greater than the profit of execution from SPM. However since our architecture has a direct memory access controller, the transfer cost is negligible and it is always profitable to execute the entire code from the SPM. We therefore do not consider the 'what to map' problem in this work. The focus of this work is the second problem, termed as address assignment or 'where to map' which involves determining the addresses on the SPM where the code will be mapped.

Code mapping techniques for SPM can be classified into static and dynamic techniques. In static techniques, SPM is loaded once during program initialization occupying the entire SPM and the contents do not change during the execution of the program. This implies that the static techniques need not address the 'where to map' issue; they only solve the 'what to map' issue. The reduced utilization of SPM

at runtime means less scope for energy reduction. Dynamic techniques on the other hand, replenishes the contents of the SPM with different code segments during program execution by overlaying multiple code segments. For most efficient management, the SPM can be partitioned into bins or regions and multiple code segments with non-overlapping live ranges should be mapped to different regions. Thus a dynamic technique for code mapping can be broken down into

1. Partition of the SPM into optimal number of regions
2. Overlaying the code objects onto the regions

Although previous dynamic approaches viz. first-fit [11] and best-fit [10] have proposed solutions for the second subproblem, none of the above approaches determine the optimal size and number of regions. These heuristics assume a pre-determined number of regions and may cause spilling of critical functions to the main memory. In fact, the above two sub-problems have a cyclic dependency and if solved independently one after another, the combined solution is sub-optimal. In this paper we propose a Simultaneous Determination of Region and Function-to-Region Mapping (SDRM) technique which solves the two subproblems at the same time. Regions are created as each function gets mapped to the SPM and are resized if the mapped function is greater than the existing region size, without violating the total size constraints. To compare the optimality of our technique, we also formulate a binary ILP to solve the code mapping problem. Our experiments using MiBench benchmark suite indicate that our technique can find near-optimal solutions compared to the ILP solutions and they are 27% better than the solution obtained by first-fit heuristic.

2 Related Work

As discussed in the previous section, SPM mapping techniques can be classified into static and dynamic techniques for both code and data. Papers [1, 2, 8] present static techniques for SPM allocation. While authors in [8] use a knapsack algorithm for static assignment of code and data objects, authors in [1] propose a dynamic programming approach to select and statically assign code objects to maximize energy saving. The static approach in [2] concentrates only on data objects.

While static approaches are easy to formulate, they significantly limit the scope of energy reduction. Therefore a majority of research [4, 9–11] have focussed on solving both code and data mapping problem using dynamic techniques. In this research work, we also propose a dynamic technique, but overlay only code objects due to greater energy reduction potential.

The approach in [9] formulates a binary ILP to select an optimal set of code blocks and corresponding copy points which minimize energy consumption. However their approach does not solve the 'where to map' problem. The authors in [4] propose another dynamic technique for systems with virtual memory, where the page fault exception mechanism of MMU is used to copy code blocks to SPM on demand. However this technique dictates some hardware enhancement. On the contrary our technique is a pure software method and does not impose any architectural changes. The research in [10] proposes yet another dynamic profile SPM allocation technique where the authors give a heuristic for classification of code, stack and global data into SPM and cache, and

a best-fit heuristic to solve the 'where to map' problem. However their technique use compaction to minimize fragmentation which can incur a significant overhead and can be prohibitive in embedded systems.

Except [11], which use static analysis for code objects, all the above techniques use profiling to find the execution count of objects. A relative advantage of static analysis over profiling has already been discussed in the previous section. The technique that we propose is closest to the approach presented by authors in [11]. They formulate an Integer Linear Programming (ILP) problem to partition the memory objects into SPM and main memory and then use another ILP to determine the address assignment. Since an ILP is intractable for large size programs they propose a first-fit heuristic to solve the 'where to map' problem. However, the heuristic in their work use a predetermined number and size of regions. In contrast, the technique in our work computes the number and size of regions while solving the mapping problem itself. We also formulate a binary ILP and show that our heuristic is near-optimal to the ILP solution. In the next section we formulate a generic problem definition for the mapping of code to SPM.

3 Problem Definition

INPUT:

- Global Call Control Flow Graph (GCCFG). GCCFG is an ordered directed graph D=(V_f, V_l, V_i, E), where each node $v_f \in V_f$ represents function or F-node, $v_l \in V_l$ represents a loop or L-node, $v_i \in V_i$ represents a conditional or I-node and edge $e_{i,j} \in E \ni v_i, v_j \in V_f \bigcup V_l \bigcup V_i$ is a directed edge between F-nodes, L-nodes and I-nodes. If v_i and v_j are both F-nodes, the edge represents a function call. If either one is a L-node, the edge represents a control flow. If either one is a I-node, the edge represents a conditional flow. If both are L-nodes the edge represents nested control flow. Recursive functions are represented by edges whose source and destination are the same. The edges of a node are ordered, i.e. if a node has two children, the left node is called before the right node in the control flow path of the program. Each F-node is assigned a statically determined weight w_i representing its execution count.
- Set S = $\{s_1, s_2 ... s_f\}$, representing the functions sizes (F-nodes V_f in the GCCFG).
- $E_{spm/access}$ and $E_{i-cache/access}$, representing the energy per access for SPM and Instruction Cache, respectively.
- E_{mbst}, energy per burst for the main memory.
- E_{ovm}, energy consumed by instructions in overlay manager code.

OUTPUT:

- Set $\{S_1, S_2 ... S_r\}$, representing sizes of regions $R = \{R_1, R_2 .... R_r\}$, s.t. $\sum S_r \leq SPMSize$.
- Function-to-Region mapping, $X[f, r] = 1$, if f is mapped to region r, s.t. $\sum s_f \times X[f, r] \leq S_r$.

OBJECTIVE:

Minimize Energy Consumption for the given application. Given the GCCFG of an application, the objective is to create regions and function-to-region mapping such that when the application instrumented with this binary is executed on the given SPM, the total energy consumed is minimized. The total energy consumption is a summation of $E_{hit}^{v_i}$, (energy on SPM hit) and $E_{miss}^{v_i}$ (energy on SPM miss) where $v_i \in V_f$. While

$E_{hit}^{v_i}$ consists of energy consumed by the overlay manager to check if the function v_i is present in SPM and energy consumed by the execution of the function from SPM, $E_{miss}^{v_i}$ has an additional energy component for moving the called function v_i from main memory to SPM and then moving the caller function back v_j on return. Code is transferred in burstsize of N_{mbst}. $nhit_{v_i}$ and $nmiss_{v_i}$ represents the number of hits and misses for the function v_i. The following equations characterizes the objective function

$$E_{hit}^{v_i} = nhit_{v_i} \times (E_{ovm} + E_{spm/access} \times s_i) \tag{1}$$

$$E_{miss}^{v_i} = nmiss_{v_i} \times (E_{ovm} + E_{spm/access} \times s_i + \frac{E_{mbst} \times (s_i + s_j)}{N_{mbst}}) \tag{2}$$

$$E_{total} = \sum_{v_i \in V_f} (E_{hit}^{v_i} + E_{miss}^{v_i}) \tag{3}$$

4 Our Approach

The goal of our approach is to use static analysis to dynamically map application code to regions on the SPM. Since the two sub-problems viz. region size determination and function-to-region mapping have a cyclic dependency, solving them independently will lead to sub-optimal results. Therefore, we require a technique to simultaneously solve the two sub-problems.

4.1 Overview

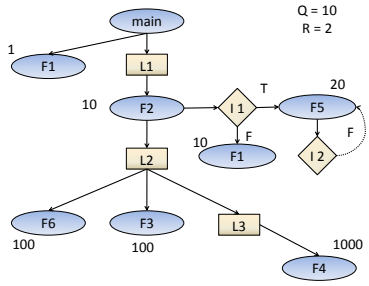
We first apply static code analysis to create a Global Call Control Flow Graph (GCCFG). Weights are assigned to nodes of the GCCFG , which is then transformed into an Interference Graph (I-Graph). The I-Graph and SPM size are then used as input to an ILP or SDRM heuristic to determine the number of regions and function-to-region mapping. The construction of GCCFG, weight assignment and I-Graph are explained in the following subsections with the help of an example shown in Figure 1(a).

```
MAIN ( )
    F1( )
    FOR
        F2 ( )
    END FOR
END MAIN

F5 (condition)
    IF ( condition )
        ……
    ELSE
        F5( condition)
    END IF
END F5
```

```
F2 ( )
    FOR
        F6 ( )
        F3 ( )
        WHILE
            F4 ( )
        END WHILE
    END FOR
    IF ( )
        F5 ( )
    ELSE
        F1( )
    END IF
END F2
```

(a) Example Code



(b) Global Call Control Flow Graph

Fig. 1. Construction of GCCFG

4.2 Construction of GCCFG

The GCCFG is an extension of the traditional Control Flow Graph (CFG) which is a representation of all paths that might be traversed through a function during its execution. A CFG is constructed for each function in the program and then all the CFGs are combined into a GCCFG in two passes. In the first pass the basic blocks are scanned for presence of loops (back edges in a dominator tree), conditional statements (fork and join points) and function calls (branch and link instructions). The basic blocks containing a loop header are labeled as L-node, those containing a fork point are labeled as I-node and the ones containing a function call are labeled as F-node.

If a function is called inside a loop, the corresponding F-node is joined to the loop header L-node with an edge. L-nodes representing nested loops, if any, are also joined. F-nodes not inside any loop are joined to the first node of the CFG. The first node, F-nodes , L-nodes and corresponding edges are retained, while all other nodes and edges are removed. Essentially this step trims the CFG, while retaining the control flow and call flow information. In this paper we assume that both paths, i.e. T and F edges, of a I-node will be executed, which is very similar to branch predication [7]. Therefore, although the GCCFG contain the I-nodes, the interference graph construction algorithm in Section 4.4 does not consider the presence of I-Nodes to determine the interference relationships between the F-nodes.

In the second pass, all CFGs are merged by combining each F-node with the first node of the corresponding CFG. Recursive functions are joined by a dashed edge. The merge ensures that strict ordering is maintained between the CFGs, i.e. if two functions are called one after another, the first function is a left child and the other function is a right child of the caller function. Thus the GCCFG is an approximate representation of the runtime execution flow of the program.

4.3 GCCFG Weight Assignment

For all F-nodes $v_f \in V_f$ of GCCFG, weights w_f, defaulting to unity, are assigned. The GCCFG in traversed in a top-down fashion. When an L-node is encountered, the weights of all descendent F-nodes are multiplied by a fixed quantum, Loop Factor Q. This ensures that a function which is called inside a deeply nested loop will receive a greater weight than other functions. For an F-node representing recursive function, the weight of the node is multiplied by a different fixed quantum, Recursive Factor R. This ensures that a recursive function will receive a greater weight than non-recursive ones. For the example shown in Figure 1(b), we choose Q = 10 and R = 2.

4.4 Interference Graph Construction

The weighted GCCFG has to be augmented considering the fact that if one function calls another function mapped to same region, then they will swap each other out during the function call and return back. Also if two functions mapped to same region are called one after another in the same nested level, then they will thrash excessively. Such functions are said to be interfering with one another and the GCCFG is not adequate to capture these interfering relationships. We transform the GCCFG into an

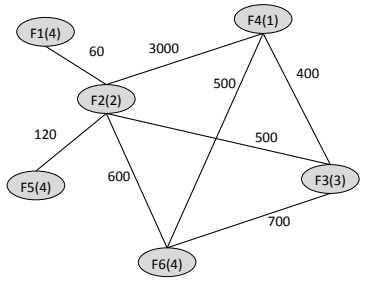
Algorithm 1 CONSTRUCT-IGRAPH (GCCFG = (V_f,V_l,E))

for v_i = v_1 to ($v_f \bigcup v_l$) **do**
  for v_j = v_i to ($v_f \bigcup v_l$) **do**
    node = *least-common-ancestor*(v_i,v_j)
    if (node == main) **then**
      relation(v_i, v_j) = NULL ; cost [v_i,v_j] = 0;
    else if (node == L-Node) **then**
      relation(v_i,v_j) = callee-callee-in-loop ; cost [v_i,v_j] = (s_i + s_j) × MIN (w_i, w_j)
    else if (node == ($v_k \neq \{v_i, v_j\}$)) **then**
      relation(v_i,v_j) = callee-callee-no-loop ; cost [v_i,v_j] = (s_i + s_j) × MIN (w_i, w_j)
    else if (node == v_i || node == v_j) **then**
      if (L-node in path from v_i to v_j) **then**
        relation(v_i, v_j) = caller-callee-in-loop ; cost [v_i,v_j] = (s_i + s_j) × w_j
      else
        relation(v_i, v_j) = caller-callee-no-loop ; cost [v_i,v_j] = (s_i + s_j) × w_j
      end if
    end if
  end for
end for

I-Graph as outlined in Algorithm 1. Figure 2(a) shows the interference relationships and Figure 2(b) depicts the corresponding I-Graph between different nodes for the example GCCFG in Figure 1(b). In the next section we discuss an ILP and a heuristic which takes the *nodes* and the *cost* from the I-Graph as input and determines the region as well as the node (function)-to-region mapping.

NODE	NODE	INTERFERENCE RELATION
F2	F3	caller-callee-in-loop
F2	F4	caller-callee-in-loop
F2	F5	caller-callee-no-loop
F2	F6	caller-callee-in-loop
F3	F4	callee-callee-in-loop
F3	F6	callee-callee-in-loop
F4	F6	callee-callee-in-loop
F1	F2	caller-callee-in-loop

(a) Interference Relationships for the example GCCFG



(b) Interference Graph derived from the GCCFG

Fig. 2. Construction of I-Graph

5 Address Assignment: Where To Map

The problem of mapping functions-to-regions is a harder problem than the bin packing problem as the size of regions or bins is not fixed and each function (item to be placed in a bin) has an associated cost. Therefore, we propose a binary ILP and a heuristic to solve the 'where to map' problem.

5.1 Optimal Solution: Binary ILP

The input to the ILP is the I-Graph I = (V_f, E') constructed in previous section with s_i representing the size of node $v_i \in V_f$ and a $cost[v_i, v_j]$ associated with each edge (v_i, v_j). The output of the ILP is the function-to-region mapping $MAP : V_f \rightarrow R$, where R is the set of regions created. We define a binary integer variable $X[v_i, r]$ which is set to 1 if v_i is mapped to region r in SPM and set to 0 otherwise.

The cost of a region is the cost of placing two or more interfering nodes in the same region. The total cost is the summation of the cost of each region. The objective function to be minimized is the total cost of the interference graph which is given by (4) and subject to the constraints (5) and (6).

$$\text{Minimize} \sum_{(v_i,v_j)\in E'} X[v_i, r] \times X[v_j, r] \times cost[v_i, v_j], \quad \forall r \in R \tag{4}$$

$$\sum_{r\in R} \max_{v_i \in V_f} (X[v_i, r] \times s_i) \leq SPMSize \tag{5}$$

$$\sum_{r\in R} X[v_i, r] = 1, \quad \forall v_i \in V_f \tag{6}$$

The first constraint (5) ensures that the sum of the sizes of all regions doesn't exceed the SPM size. The size of a region is the size of the largest function mapped to the region. Although the *max* function used above makes the constraint non-linear, it is linearized during implementation by making sure that all possible combinations of regions and functions mapped to the SPM does not exceed its size. The second constraint (6) ensures that a function is not mapped to more than one region. Because of the presence of two variables $X[v_i, r]$ and $X[v_j, r]$ in (4), the objective function is non-linear and cannot be modeled using LP. To make the above function linear, we introduce a new binary variable *U[*v_i*,*v_j*,r]* which is set to 1 if both v_i and v_j are mapped to same region r and set to 0 otherwise. The linearized objective function is given by equation (7).

$$U[v_i, v_j, r] \geq X[v_i, r] + X[v_j, r] - 1$$

$$U[v_i, v_j, r] \leq \frac{X[v_i, r] + X[v_j, r]}{2}$$

$$\text{Minimize} \sum_{(v_i,v_j)\in E'} U[v_i, v_j, r] \times cost[v_i, v_j], \quad \forall r \in R \tag{7}$$

Since solving ILP may require prohibitively large computation resources, in the next section we propose a heuristic to solve the 'where to map' problem.

5.2 SDRM Heuristic

Our heuristic is based on the following observation. If two functions are joined by an edge in the I-Graph, then mapping them to the same region will incur a cost equal to the edge weight. The total cost of a region is the summation of edge weights of all such interfering functions. Algorithm 2 outlines the mapping procedure. The routine *Overlay-I-Graph* maps nodes of the I-Graph for the given size of the SPM. The output is the array *R* representing region sizes and array *node-address* representing the function-to-region mapping.. Line (3) sorts the edges of I-Graph in decreasing order

Algorithm 2 SDRM Heuristic

Overlay-I-Graph(I-Graph,SPM-Size)
global int num_regions = 0
global array address[]

```
1:  R[]: array of integer (size)
2:  node-address[]: array of integers
3:  sort-decreasing(E')
4:  for all e=(v_i,v_j) in E' do
5:     for v_k = v_i, v_j; v_k ≤ SPM-Size do
6:        if (node-address[v_k]==NULL) then
7:           r = Determine-Region(v_k)
8:           node-address[v_k] = address[r]
9:           R[r] = max(R[r], size(v_k))
10:       end if
11:    end for
12: end for
13: return R and node-address
```

Determine-Region(Function v_k)
global int size_remaining = SPM-Size

```
1:  for all  r in R, starting with least cost do
2:     find r, s.t. e = (v_k,v_j) ∉ E', v_j = MAP(r)
3:     if ( found r) then
4:        return r
5:     end if
6:  end for
7:  if (size(v_k) ≤ size_remaining) then
8:     r = ++num_regions
9:     address[r] = SPM-Size - size_remaining
10:    size_remaining - = size(v_k)
11: else
12:    find r, s.t. cost of placing v_k to r is min
13: end if
14: return r
```

of their weights. This ensures that the most interfering nodes are placed in separate non-overlapping regions of SPM if not constrained by the SPM size. It then calls the routine *Determine-Region* to find the region mapping for all unmapped nodes (4–7) and updates the corresponding region size after the node is mapped (8–9).

The routine *Determine-Region* determines the region for each unmapped node. It first checks if the node can be mapped to an existing region such that there is no interference with already mapped nodes in that region (1–6). If not, it checks if the node can be assigned to the remaining space, thereby creating a new region (7–10). Otherwise it finds an existing region such that the cost of the region after overlaying the node is minimum (12). In the worst case, all nodes will interfere with one another, complexity $O(E')$. Moreover the computation of the cost function will involve checking every node, complexity O(V_f). Hence the runtime complexity of the algorithm is O($V_f \times E'$).

6 Scratchpad Overlay Manager

The final step in the mapping process involves instrumenting the code with the mapping information obtained from SDRM or ILP and linking it with the SPM overlay manager (SOVM). The SOVM is responsible for keeping a track of function call and return during program execution. It has two data structures, the overlay table and region table. The overlay is filled with the mapping information during linking phase. The region table is used to keep a track of all functions currently residing in each region of SPM. Each function call and return statement in the application code is replaced by a stub function call to the SOVM. If the called function is not currently residing in SPM, the SOVM issues a direct memory access (DMA) command to transfer the function from main memory. The SOVM manager code then transfers the program control to the first

Table 1. Energy Model

Size(KB)	SPM(nJ)	4-way Cache(nJ)	Size(KB)	SPM(nJ)	4-way Cache(nJ)
0.5	0.107	0.534	4	0.145	0.551
1	0.128	0.538	8	0.173	0.564
2	0.134	0.542	16	0.206	0.587

instruction in the overlayed function. The SOVM and its data are mapped to the main memory to reduce the mapping pressure on the heuristic. Since the SOVM instructions and its associated data structures are fetched from the cache, we might see some runtime performance degradation. Our experiments show that the degradation is minimal.

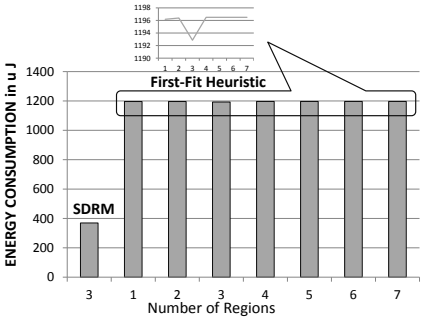
7 Experimental Setup

The instrumented binary is executed on a *cycle accurate simulator* that models an Intel XScale processor. The simulator has been augmented to model an on-chip SPM at the same level as instruction cache. The system modeled has a 32 KB instruction cache, 32 KB data cache, and a SPM, the size of which can be selected by the system designer. The simulator models a low power 32MB SDRAM from Micron as the main memory. The SPM is physically addressed and incoherent with the main memory subsystem. We perform our experiments on a set of embedded applications from [6]. The applications used and their respective code sizes are dijkstra(1588), patricia(2904), rijndael(21050), sha(2376), susan(46808), fft(4688), adpcm(1436), blowfish(9308). The per access energy numbers for SPM and I-Cache are given in table 1.

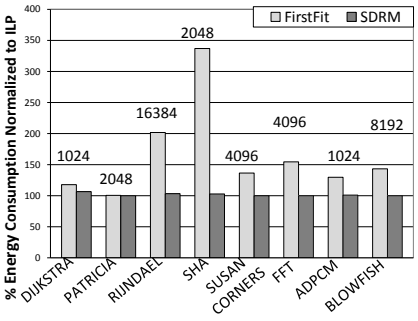
8 Results

8.1 First-Fit vs SDRM

In this section we present a comparison of total energy consumption between the ILP, SDRM and the first-fit heuristic for various benchmarks. For first-fit we assume that the SPM is divided into variable sized regions (Experimentally we found that variable-sized region gives better results than equal-sized region). The previous approach does not precisely state a way of finding these region sizes. To be unbiased, we performed an exploration for various sizes and number of regions. For example, for x bytes of SPM, we divided it into x/2, x/4, x/8,... x/r, where the value of r was found by exploration. The regions were considered in the same order for allocation. Figure 3(a) show the first-fit energy consumption trend for *sha* as we explore the number of regions for a 2KB SPM. As shown in the graph, there is an optimal number of regions ($r = 3$) in first-fit, at which the energy consumption is minimum. For smaller number of regions ($r < 3$), not all interfering functions can be mapped, since the number of such functions is higher than the number of regions. Some functions are spilled to main memory, resulting in a higher energy consumption. As we increase the number of regions, more functions will be overlayed and the energy consumption decreases, reaching a local minimum



(a) SHA Benchmark: first-fit heuristic with varying number of **variable** sized regions



(b) Energy Comparisons between ILP, SDRM and first-fit for various benchmarks

Fig. 3. First-Fit vs SDRM

at $(r = 3)$. However, if we compare this value with the first bar which indicates the energy consumption from SDRM mapping, it is significantly higher. The reason is that the critical function for sha does not fit into any region of the SPM, corroborating our argument that pre-determining the number of regions does not lead to optimal solution. Further increase in number of regions $(r > 3)$ fragments the SPM into smaller sized regions. As large sized functions cannot fit, this again results in spilling of such functions to the main memory which causes a rise in energy consumption. On further increase $(r > 4)$, the SPM gets more fragmented, but the mapping does not change and there is no change in energy consumption.

To the best of our knowledge, none of the previous approaches have demonstrated any technique for finding the optimal number of regions at which the energy consumption would be minimal. The only way to find this number is by exploration of the entire solution space by varying the number and size of regions. The search space can be reduced by smart exploration techniques, but only up to a limited extent as the exploration process is a time consuming task involving recompilation and execution of program every time. The SDRM technique proposed does not incur this expense as it simultaneously finds the optimal number of regions and their sizes while solving the mapping problem. The first bar in the graph shows the energy results obtained by SDRM for a 2KB SPM. SDRM divides the SPM into three variable sized regions and exhibit a 69% energy reduction compared to first-fit which divides the SPM into three variable sized but pre-determined regions.

Figure 3(b) shows the comparison of energy consumption between SDRM and first-fit heuristic for various benchmarks, normalized to the ILP energy values. The optimal number and size of the regions for first-fit are found by exploration as discussed previously. From the figure, we observe that the energy for SDRM is always close to 100%, indicating that the solution obtained from the SDRM heuristic is close to the optimal ILP solution. Moreover, the maximum energy reduction is observed for *sha*, where the first-fit performs poorly, as the most critical region does not even fit into any region. On the other hand, since the SDRM does not predetermine the region sizes, the critical functions are always mapped to some region of the SPM. On an average we observe a 27% energy reduction for SDRM compared to the first-fit technique.

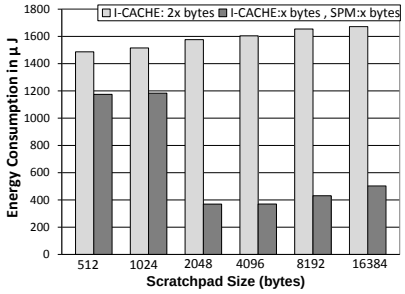


Fig. 4. SHA: Energy comparisons between COA and HSA SDRM

8.2 Cache-only vs Horizontally Split Architecture

In this experiment, we compare our mapping technique for a HSA against a COA. The COA architecture consists of 2x bytes of I-cache while the HSA consists of x bytes of SPM and x bytes of I-cache. Figure 4 shows how the HSA architecture with SDRM technique performs in comparison with a COA for *sha* benchmark.

For small sizes of SPM, the critical functions do not fit into the SPM at all and are spilled to cache. Hence there is no significant difference between a COA and a HSA. As we increase the size to 2048 bytes, all functions can fit into the SPM, and the functions would need to be overlayed as the aggregate size of 2376 bytes for *sha* is greater than 2048 bytes. At this size of SPM, we see a significant reduction in energy as all the code is fetched and executed from the SPM instead of the I-cache. At a larger size of 4096 bytes, all the functions can be mapped onto the SPM without any overlay, which means no calls to the SPM overlay manager and no runtime overhead due to DMA transfer instructions. We should therefore have observed a further decrease in energy consumption. However, since we assume a model in table 1 where the energy per access for SPM increases with size, we observe an increase in energy consumption with increasing size of SPM. For *sha* benchmark the HSA shows a reduction of 77% compared to the COA. The average reduction is 35% across all benchmarks.

This experiment demonstrates the effectiveness of a split memory subsystem architecture when supported by an intelligent mapping technique like SDRM. In other words, given an architecture with only an instruction cache, we can always reduce the energy consumption by splitting the power hungry instruction cache equally into a SPM and a smaller instruction cache. A pure compiler technique like SDRM can then be used, requiring just a simple recompilation of the application, with no profiling overhead.

8.3 Performance Overhead

Since the SOVM code is fetched and executed from the I-cache, there is a performance penalty in terms of runtime cycles due to the extra instructions. One way to reduce this overhead would be to map the SOVM code to the SPM instead of cache. However, this would mean less space available to map the functions themselves resulting in a

potential spill of some critical functions to the cache, which means a greater energy consumption. An additional penalty is incurred due to clearing of the branch target buffer table, each time an overlayed function is transferred from main memory to SPM. This is essential, otherwise branch instructions would jump to invalid addresses from the previous overlayed function, thereby crashing the application. There is also an additional penalty due to stalls during code transfer from main memory to SPM. We observe an average performance degradation of 1.9% across all benchmarks.

9 Conclusion

In this paper, we presented a fully-automated, dynamic, code overlaying technique based on pure compiler analysis for energy reduction for on-chip scratchpad memories in embedded processors. We formulated an ILP which gives an optimal solution and a heuristic which gives a near-optimal solution and simultaneously addresses both the important issues of region size determination and function-to-region mapping. The proposed technique and HSA architecture succeeds in achieving a greater energy reduction against a previous approach and a unified instruction COA architecture, respectively. Compared to the best performing previously known heuristic our approach achieves an average energy reduction of 27% with an average performance degradation of just 1.9%. We also demonstrated that by splitting the I-cache into equal sized smaller I-cache and SPM and using a pure compiler technique like SDRM, we can always reduce the total energy consumption. This paves the path of reducing the memory subsystem energy even in general purpose processors employing the split architecture.

References

1. F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In Proc. CASES, pages 259-267, 2004.
2. O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. Trans. on Embedded Computing Sys., 1(1):6-26, 2002.
3. R. Banakar, S. Steinke, et al. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In Proc. CODES, pages 73-78, 2002.
4. B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In Proc. EMSOFT, pages 321-330, 2006.
5. J. Montanaro, R. T. Witek, et al. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. Digital Tech. J., 9(1):49-62, 1997.
6. M. Guthaus, J. Ringenberg, et al. Mibench: A free, commercially representative embedded benchmark suite. Workshop on Workload Characterization, pages 3-14, 2 Dec. 2001.
7. J. E. Smith. A study of branch prediction strategies. In Proc. ISCA, pages 135-148, 1981.
8. S. Steinke, L.Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In Proc. DATE, page 409, 2002.
9. S. Steinke, N. Grunwald, et al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In Proc. ISSS, pages 213-218, 2002.
10. S. Udayakumaran, A. Dominguez, et al. Dynamic allocation for scratch-pad memory using compile-time decisions. Trans. on Embedded Computing Sys., 5(2):472-511, 2006.
11. M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. IEEE Trans. on VLSI, 14(8):802-815, Aug. 2006.