A DYNAMIC CODE MAPPING TECHNIQUE FOR SCRATCHPAD MEMORIES IN EMBEDDED SYSTEMS

by

Amit Arvind Pabalkar

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2008

A DYNAMIC CODE MAPPING TECHNIQUE FOR SCRATCHPAD MEMORIES IN EMBEDDED SYSTEMS

by

Amit Arvind Pabalkar

has been approved

October 2008

Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Karamvir Chatha
Partha Dasgupta

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Design of modern embedded systems has become extremely challenging due to multi-dimensional and stringent design constraints like performance, cost, weight, power, real-time, time-to-market and size. Such systems typically feature low power processors coupled with fast on-chip scratchpad memories (SPMs). Scratchpads are more efficient than caches in terms of energy consumption, performance, area and timing predictability. However, unlike caches which manage the program code and data in hardware, the efficient use of scratchpads requires them to be managed explicitly, usually by the programmer. This involves deciding which code or data objects should be mapped to SPM, when to bring them in and where to bring them within the SPM - termed as the mapping process. The objective is to find a mapping which will minimize the energy consumption and maximize the performance.

In this work, a fully automated, dynamic code mapping technique for SPMs based on compiler static analysis is presented, which alleviates the programmer of this burden. The mapping problem is formulated as a binary integer linear programming problem and a heuristic is proposed to solve the problem in polynomial time. The heuristic simultaneously solves the interdependent sub problems of bin size determination and the function-to-region mapping (SDRM) and prefetches (SDRM-prefetch) code objects to maximize the performance. The technique is evaluated for a subset of MiBench applications on a horizontally split instruction cache and SPM architecture. Compared to a cache-only architecture, SDRM-prefetch on the split architecture gives an average energy reduction of 32.3%, with a performance improvement of 5.78%. Moreover, SDRM-prefetch achieves 25.9% energy reduction compared to the previous known static analysis based mapping heuristic.

To

my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor and committee chair Dr. Aviral Shrivastava, without whom this work would have been impossible. I would like to thank him for those long and sometimes heated discussions we had about almost every detail of this topic, and for his patience for listening to my arguments. His support, invaluable guidance and encouragement helped me throughout the completion of my master's degree program and my research.

I would also like to thank Dr. Karamvir Chatha and Dr. Partha Dasgupta for the invaluable guidance they provided me as committee members. I thank my lab colleagues Jong-eun Lee, Sai Mylavarapu, and Reiley Jeyapaul for the research discussions. Arun, Khushboo, Rooju and Vivek thanks for the all the wonderful and fun moments, and of course the technical discussions. Life in the lab would have been boring without your chatter and jokes.

I would like to acknowledge the Computer Science and Engineering Department for providing me teaching assistantship and the Consortium of Embedded Systems for providing me with the financial support in the form of an internship. I would like to thank Intel Corporations, Chandler and my mentor Dr. Hari Tadepalli for exposure to some cutting edge technologies, which broadened my engineering as well as research perspective.

I am very grateful to my parents. Without their encouragement it would have been impossible for me to finish this work. Last but not the least, I would like to express my heartfelt gratitude for my wife Pashmina, for her unwavering and unflinching support during my research work. She has been a great source of inspiration in my life.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

I. Introduction

The first generation embedded systems were limited to fixed, single functionality devices like digital watches, calculators, washing machines etc. Modern embedded systems have evolved into programmable, highly complex, multi-functionality devices including navigation systems, portable music players, gaming consoles, personal digital assistants and cellular phones. These systems must exhibit high performance while at the same time consume less power, as they operate on battery. Design of such systems thus becomes extremely challenging due to multi-dimensional and stringent design constraints including but not limited to performance, cost, weight, power/energy, real-time, time-to-market, and size.

Modern embedded processors improve performance by employing memory hierarchies consisting of caches or scratchpads or both. As shown in Figure 1 a scratchpad is usually placed at the same level as a L1 cache in the memory hierarchy after the internal CPU registers.

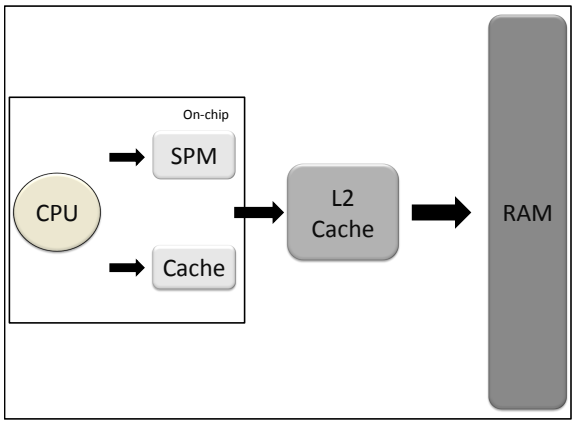

Fig. 1. Memory hierarchy

A. *Cache Memory*

A CPU cache memory is used by the central processing unit of a computer to reduce the average time to access memory. The CPU cache is a smaller, faster memory which stores

copies of the data from the most frequently and recently used main memory locations. As long as most memory accesses are to the cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of the main memory. Figure 2 shows a block diagram of a typical cache which consists of the memory array, address decoder logic, the column circuitry, the tag array, the tag comparators and muxes. While caches improve performance by exploiting the spatial and temporal locality of the application, without any changes to the application itself, these improvements are achieved through use of tag arrays and comparators which in certain processors like StrongARM, can consume more than 40% of the total power budget [1].

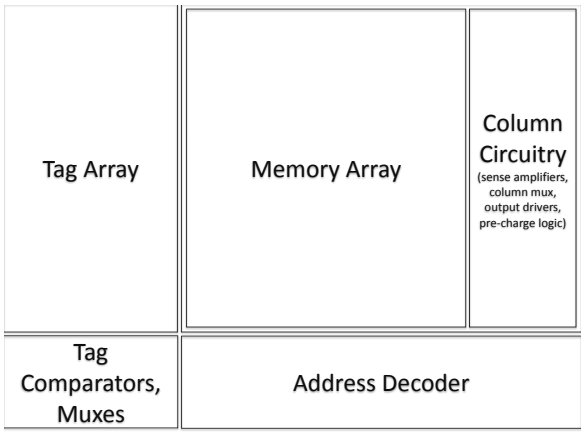

Fig. 2. Cache memory organization

B. *Scratchpad Memory*

Scratchpad Memory (SPM), also known as tightly couple memory (TCM), local store, or simply scratchpad, is typically a high speed, low latency internal memory used for temporary storage of code and/or data, referred to as memory objects. In reference to a CPU, SPM refers to a special high-speed memory circuit used to hold memory objects with explicit instructions to move data to and from the main memory, often using DMA based

data transfer. A system with scratchpads is a system with Non-Uniform Memory Access (NUMA) latencies, because the memory access latencies to the different scratchpads and the main memory vary. Moreover, a system with scratchpad is generally non-coherent; the scratchpad does not commonly contain a updated copy of data that is stored in the main memory.

B.1. *SPM Examples*

- NVIDIA's 8800 GPU running under CUDA provides 16KiB of Scratchpad per thread-bundle when being used for gpgpu tasks [2].
- Each SPE's in CELL BE architecture have its own private local store and relies on DMA for transfer to/from main memory and the local store. There is no coherence between the local stores as each processor's workspace is separate and private.
- SH2, SH4 used is Sega's consoles lock cachelines to an address outside of the main memory, for use as a SPM.
- Intel's IXP1200 and the later processors have scratchpad accessible from both the microengines and the control, enabling more memory operations in parallel.

As shown in Figure 3 scratchpad memory is a memory array with a decoding and column circuitry logic and a conspicuous absence of tag array, tag comparators and the muxes. This model is designed keeping in view that the memory objects are mapped to the scratchpad in the last stage of the compiler or by the programmer himself. The basic assumption here is that the scratchpad occupies one distinct part of the memory address space with the rest of the space occupied by main memory. Thus, we need not check for the

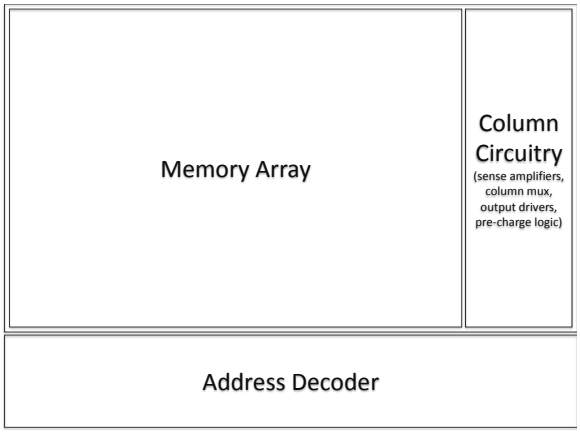

Fig. 3. Scratchpad memory organization

availability of the memory objects in the scratchpad. It reduces the tag comparator and the signal miss/hit acknowledgement circuitry, which is needed in the cache. This leads to energy as well as area reduction of a scratchpad compared to a cache.

While previous works have demonstrated that a scratchpad 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 contents of the scratchpad. 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 the scratchpad thus becomes very critical for efficiently utilizing it with minimal runtime transfer overhead.

C. *Code vs Data*

During a program execution, the instruction cache is accessed for each instruction fetch while the data cache is accessed for load and stores instructions only. Additionally, while data may show more varied accessed pattern, instructions exhibit more predictability and locality because of their higher execution count and smaller size [4]. Thus data can be mapped to the main memory (cached) and its performance can improved by techniques like

dynamic prefetching. Instructions on the other hand are likely to show more improvements in performance and energy consumption if mapped to the scratchpad. Therefore, we focus on developing techniques for mapping code objects to SPM.

D. *Profiling vs Static Analysis*

Most code mapping techniques for SPM 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.

E. *Scratchpad Code Mapping Problem*

Traditional approaches for SPM utilization breaks down the SPM mapping problem into two smaller problems.

- *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.

- *Address assignment* or *'where to map'*: involves determining the addresses on the SPM where the code will be mapped. The focus of this dissertation is on this second problem

F. *Static vs Dynamic Mapping*

The SPM code mapping techniques 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

- Partition of the SPM into optimal number of regions

- Overlaying the code objects onto the regions

Although previous dynamic approaches viz. first-fit [5] and best-fit [6] 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 sub-problems 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 solution compared to the ILP solution and it is 25.9% better than the solution obtained by first-fit heuristic.

G. *Summary of Contributions*

Here we summarize our contributions in this thesis:

- We propose a novel, fine-grained dynamic code mapping technique for scratchpad that simultaneously solves the independent problems of region size determination and function to region mapping.
- Our approach is based on static analysis and does not require expensive and prohibitive task of profiling.
- Our approach is purely compiler based and does not require and changes to the underlying hardware.
- Experiments show that, with our approach, we can achieve an energy reduction of 25.9% compared to previous known approach based on static analysis.
- The prefetching optimization results in a performance improvement of 5.78% in terms of execution cycles.

H. *Organization of the Thesis*

In Chapter II, we present a detailed discussion of existing SPM mapping techniques, particulary for mapping code. We present both dynamic techniques and static techniques based on profiling and static code analysis for mapping code. In Chapter III, we give a formal problem definition with input, output and the objective function for the code mapping problem. In Chapter IV we discuss our approach to solve the code mapping problem. In Chapter V we formulate a binary ILP and present a heuristic as a solution to the mapping problem. The implementation of the heuristic and the scratchpad overlay manager is discussed in brief in Chapter VI while Chapter VII details out the performance overhead and the prefetching technique to minimize the overhead. In Chapter IX, we discuss our experimental setup, energy and performance models used and a detailed analysis of the results obtained. Future work and conclusions are described in Chapter X.

II. Related Works

A. *Architecture*

Horizontally Partitioned or Split Memory Architectures is a popular architecture in embedded systems. For example, in the Intel XScale, the main cache is 32 KB, and is augmented by a 2KB mini-cache, which is at the same level of memory hierarchy(L1). In this work, we also use a horizontally split memory architecture where the larger instruction cache is divided equally into a smaller instruction cache and a scratchpad.

The authors in [7] propose a horizontal partitioning of memory architectures for energy reduction. They show that by cleverly allocating the data objects to the smaller cache, a substantial amount of energy can be saved due to less energy per access of smaller caches. Our technique builds on the same concept where energy reduction is achieved by allocating the most used code objects to the scratchpad memory.

The authors in [8] also propose a horizontally partitioned memory subsystem for energy reduction by introducing a scratchpad and a mini-cache. However their technique uses an architectural modification where they use an MMU and a microTLB. Our work is a pure software method and does not require any hardware enhancement and optimization. However, a disadvantage of our method compared to [8] is that we require source code access for code insertions and transformations while their technique can work on any binary by using a post-pass compiler to do the same.

B. *Mapping Techniques*

As discussed in the previous section, SPM mapping techniques can be classified into static and dynamic techniques for both code as well as data. In static SPM mapping, the SPM is initialized with the contents of the program code and data at load time and the contents do not change during runtime. On the other hand, dynamic SPM mapping is

characterized by the fact that the contents of the SPM change during program execution. Program points where code and/or data are moved between the main memory and SPM are identified at compile time and additional instructions are inserted at these points to carry out the movement. Both static and dynamic techniques can be further classified into techniques that consider only data, only instructions (code) or both. Figure 4 shows the taxonomy of SPM mapping techniques.

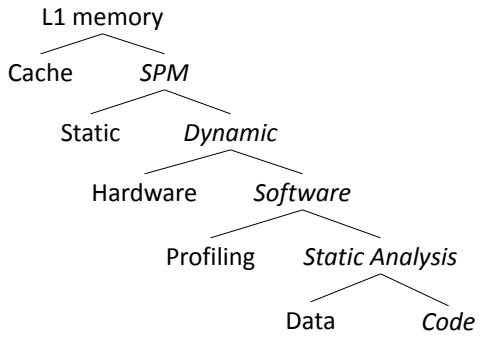

Fig. 4. Taxonomy of SPM mapping techniques

B.1. *Static Techniques*

Papers [9–16] present static techniques for SPM allocation. Authors in [13] use a knapsack algorithm for static assignment of code and data objects. Authors in [9] and [10] propose a dynamic programming approach to select and statically assign code objects to maximize energy saving. While [9] splits the available SPM into several partitions using a special hardware, [10] makes use of a post-pass optimizer to modify the code, so that the program runs on a unified SPM. The static approach in [16] concentrates only on data objects. In a slightly different approach, authors in [14] present a cache aware scratchpad allocation algorithm, where the objects with conflicting cache addresses are determined and statically allocated to SPM. They formulate an ILP to select the optimal set of memory

objects. In [11], the authors have proposed a SPM allocation technique independent of the SPM size. The technique uses profiling information embedded into the application binary to delay the address assignment of blocks till the application is loaded onto the SPM, making it independent of the SPM size. The works in [12] and [15] aim at multi-tasking systems. While authors in [12] propose an API to help the programmer in moving blocks between main memory and SPM, the authors in [15] presents three sharing strategies: non-saving, saving and hybrid.

B.2. *Dynamic Techniques*

While static approaches are easy to formulate, they significantly limit the scope of energy reduction. Therefore a majority of research [5, 6, 8, 17–22] have focussed on solving both code and data mapping problem using dynamic techniques. The works in [19] and [20] focus on data arrays accessed from well-structured loop kernels. The arrays are tiled to allow only certain parts of an array to be copied to the SPM, thus allowing arrays larger than SPM size to be mapped to the SPM. Authors in [21] also focus on mapping data arrays to the SPM. Their technique assign registers to register classes on the basis of their size, where each register class gets a fixed portion of the SPM. Then using a conflict graph of live ranges, they propose a graph coloring algorithm which determines the array to be SPM mapped at a program point.

In this research work, we also propose a dynamic technique, but overlay only code objects due to greater energy reduction potential. The approach in [22] 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. Authors in [17] formulate a mixed ILP to solve the 'what to map' as well as 'where to

map' problem. However the method is sub-optimal since the selected code blocks are copied even if they are already present in SPM. The research in [6] 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 will be prohibitive in embedded systems. Authors in [18] propose dynamic profile technique based on concomitance metric, which measures how temporally related two code blocks are. However their technique does not produce a running version of SPM-enabled binary.

Except [5] 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 [5]. 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. Since they do not give any details, we assume that these regions are of variable size found by exploration. 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.

III. Generic Problem Definition

INPUT:

- Global Call Control Flow Graph (GCCFG). GCCFG is an ordered directed graph D=(V_f, V_l, V_i, V_c, 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, $v_c \in V_c$ represents a computation or C-node and edge $e_{i,j} \in E \ni v_i, v_j \in V_f \bigcup V_l \bigcup V_i \bigcup V_c$ is a directed edge between F-nodes, L-nodes, I-nodes and C-nodes. If v_i and v_j are both F-nodes, the edge represents a function call. If either one is a L-node or C-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 $R = \{R_1, R_2 .... R_r\}$, such that $\sum S_r \leq SPMSize$.

- Function-to-Region mapping, $X[f, r] = 1$, if f is mapped to 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

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

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

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

IV. 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.

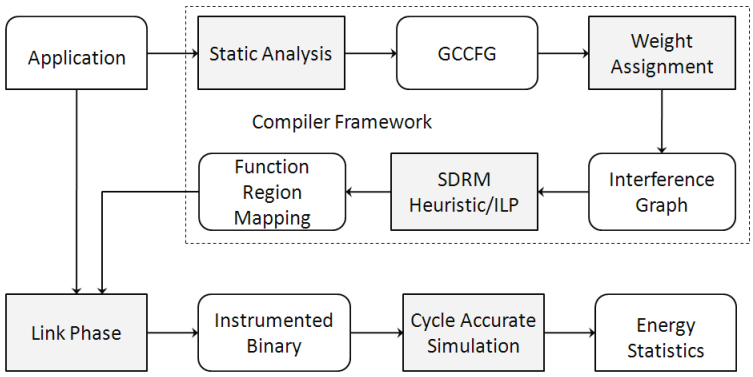

Fig. 5. Scratchpad overlay workflow

A. *Overview*

Figure 5 depicts the workflow of our scratchpad overlay approach. Static analysis is applied to the application code to create a Global Call Control Flow Graph (GCCFG). Weights are assigned to nodes of the GCCFG , which is subsequently 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, GCCFG weight assignment and I-Graph are explained in the following subsections with the help of an example code shown in Figure 6

B. *Granularity of Code Objects*

Due to small size of typical scratchpads, smaller sized code objects would lead to better mapping and hence greater benefits of scratchpad mapping. One way of achieving this is

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

Fig. 6. Example code

to consider code at the granularity of basic blocks. However, since we instrument the code by inserting function calls to the scratchpad manager, such a small granularity is likely to impact performance due to overhead of additional function calls. The other drawback is that the ILP formulation proposed becomes intractable as the number of code objects increases. The other way of obtaining small code objects is to outline function calls [23]. However, due to high implementation cost of both the above methods, we in this work operate at the granularity of function calls.

C. *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. The GCCFG also contains computation nodes or C-nodes, which will be introduced and discussed in detail in Section B.

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 [24]. Therefore, although the GCCFG contain the I-nodes, the interference graph construction algorithm in Section E 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.

D. *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

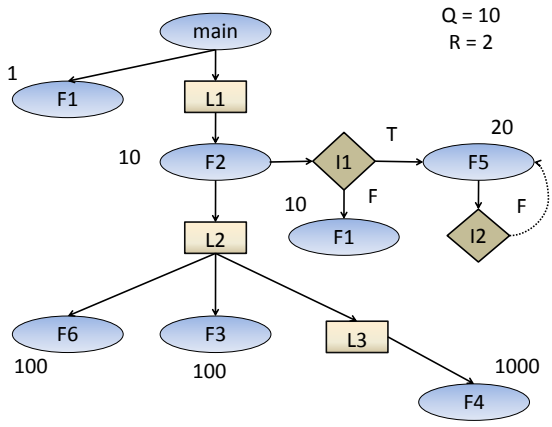

Fig. 7. Global call control flow graph

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 GCCFG shown in Figure 7, we choose Q = 10 and R = 2.

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 ; $cost_e$ [v_i,v_j] = 0;
    else if (node == L-Node) **then**
      relation(v_i,v_j) = callee-callee-in-loop ; $cost_e$ [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 ; $cost_e$ [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 ; $cost_e$ [v_i,v_j] = (s_i + s_j) × w_j
      else
        relation(v_i, v_j) = caller-callee-no-loop ; $cost_e$ [v_i,v_j] = (s_i + s_j) × w_j
      end if
    end if
  end for
end for

E. *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.

TABLE I
Interference relationships for the example GCCFG

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

As outlined in Algorithm 1, we transform the GCCFG into an I-Graph I = (V_f, E'), where each node $v_i \in V_f$ is an F-node from the GCCFG and each edge $e_{ij} \in E'$ connects a pair of interfering F-nodes or L-nodes. For all pair of nodes (v_i, v_j), we find the least-common-ancestor (LCA) using Tarjan's least common ancestors algorithm [25]. The ancestor may or may not be either of v_i and v_j. For example, the LCA of F6 and F2 is F2 itself, while the LCA of F6 and F3 is L2. If the ancestor is the *main* function, it means that both $v_i and v_j$ are called directly by *main*, not within any loop and we ignore such function pairs. If the LCA is neither v_i nor v_j, then the interference relationship is of type callee-callee-x, where x is either "in-loop" (6–7) if the LCA is a L-node or its is "no-loop" if

the LCA is another F-node (8–9). In both cases, the cost function given by the summation of their size multiplied by the minimum of their weights from the GCCFG. This is because, for callee-callee relationships, the number of times such functions will always swap each other out is determined by the lower of their execution frequencies.

Table I shows the interference relationships and Figure 8 depicts the corresponding I-Graph between different nodes for the example GCCFG in Figure 7. 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.

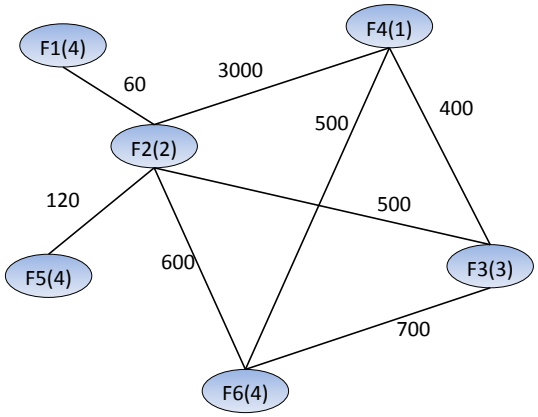

Fig. 8. Interference graph derived from the GCCFG

V. Address Assignment

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.

A. *Optimal Solution - Binary ILP*

The input to the ILP is the I-Graph I = (V_f, E') constructed in Section E 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 $X[v_i, r]$ such that

$$X[v_i, r] = \begin{cases} 1, & \text{if } v_i \text{ is mapped to region } r \text{ in SPM} \\ 0, & \text{otherwise} \end{cases}$$

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 (V.1) and subject to the constraints (V.2) and (V.3)

$$\text{Minimize} \sum_{(v_i,v_j)\in E'} X[v_i, r] \times X[v_j, r] \times cost[v_i, v_j], \quad \forall r \in R \tag{V.1}$$

$$\sum_{r\in R} \max_{v_i \in V_f} (X[v_i, r] \times s_i) \leq SPMSize \tag{V.2}$$

$$\sum_{r\in R} X[v_i, r] = 1, \quad \forall v_i \in V_f \tag{V.3}$$

The first constraint (V.2) 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 (V.3) 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 (V.1), the objective function is non-linear and cannot be modeled using integer linear programming. To make the above function linear, a new binary variable *U[v_i,v_j,r]* is introduced where

$$U[v_i, v_j, r] = \begin{cases} 1, & \text{if both } v_i \text{ and } v_j \text{ are mapped to same region } r \\ 0, & \text{otherwise} \end{cases}$$

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

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

Thus, the linear form of objective function in (V.1) becomes

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

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

B. *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* in in Algorithm 2 maps nodes (functions) of the Interference Graph for the given size of the scratchpad. The output is the array R representing

Algorithm 2 SDRM: Overlay-I-Graph (I-Graph,SPM-Size)

```
 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
```

region sizes and array *node-address* representing the start address of each function. The start address of a function is the address of the region to which it is mapped and thus represents the function-to-region mapping. Line (3) sorts the edges of I-Graph in decreasing order of their weights and node pairs connected by these edges are considered for mapping in this order. This ensures that the most interfering nodes are placed in separate regions of scratchpad if not constrained by the SPM size. It then calls the routine *Determine-Region* in Algorithm 3 to find the region mapping for all unmapped nodes (4–7) and updates the corresponding node address and 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 it cannot be mapped to an existing region, it checks if the node can be allocated to the remaining space, thereby creating a new region (7–10). If the remaining space is not enough to allocate the new node, the heuristic finds and existing region so that the cost of the region after overlaying the node is minimum (12). The cost of a region is defined as the summation of interference cost of

Algorithm 3 SDRM: Determine-Region (Function v_k)

```
1: global int num_regions = 0
2: global int size_remaining = SPM-Size
3: array address[]
4: for all  r in R, starting with least cost do
5:    find r, s.t. e = (v_k,v_j) ∉ E', v_j = MAP(r)
6:    if ( found r) then
7:       return r
8:    end if
9: end for
10: if (size(v_k) ≤ size_remaining) then
11:    r = ++num_regions
12:    address[r] = SPM-Size - size_remaining
13:    size_remaining - = size(v_k)
14: else
15:    find r, s.t. cost of placing v_k to r is min
16: end if
17: return r
```

all nodes mapped to that region. The total cost is the summation of cost of all regions created by the heuristic.

a. *Worst-Case Complexity Analysis*

In the worst case, all nodes or functions of the application will interfere with one another, thus having a complexity $O(E')$. Moreover the computation of the cost function will involve checking every node, complexity $\mathrm{O}(V_f)$. Hence the overall worst-case runtime complexity of the heuristic is $\mathrm{O}(V_f \times E')$.

VI. Scratchpad Overlay Manager

The last step in the mapping process involves instrumenting the code with the mapping information obtained from ILP or SDRM and linking this information with the code of the SPM overlay manager (SOVM). The mapping information for each function consists of

- the region number to which the function is mapped
- start address of the region which becomes the address of the function in SPM and also called virtual memory address of VMA
- address of the function in main memory also called logical memory address or LMA
- size of the function

TABLE II
Overlay table

Function ID	Region	VMA	LMA	Size
1	0	0x30000	0xA00000	0x100
2	0	0x30000	0xA00100	0x200
3	1	0x30200	0xA00300	0x1000
4	1	0x30200	0xA01300	0x300
5	2	0x31200	0xA01600	0x500

The overlay manager is responsible for keeping a track of every function call and its return. The manager code maintains two data structures, the region table and the overlay table. The overlay table as shown in Table II is filled with the mapping information during the linking phase. The region table as shown in Table III keeps a track of all functions which currently reside in various regions of SPM. Every function is assigned a unique identification number during the linking process. Each function call and return statement in the application code is replaced by a stub function call to the overlay manager, with the function id and its arguments passed as arguments to the overlay manager. The manager

looks up the region table to determine if the callee function (during function call) or the caller function (during function return) is currently mapped to an SPM region. If yes, then the manager simply jumps to the first instruction in the target function with the passed arguments for function calls or jumps to the caller function with the return value as argument for function returns. If no, then manager uses the function id to look up the overlay table to find the VMA, LMA and size of the function. It issues DMA instructions to transfer the function code from main memory to SPM. Note that since we are dealing with code, the contents of the SPM region can simply be overwritten with the new function code and need not be copied back to the main memory.

TABLE III
Region table

Region	Function ID
0	1
1	4
2	5

VII. Runtime Performance

A. *Performance Overhead*

The overall performance penalty is a summation of various factors as described in the following subsections.

A.1. *Scratchpad Manager Overhead*

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

A.2. *Branch Prediction Table Overhead*

The architecture used for the purpose of our experiments employs a branch target buffer (BTB) table for branch prediction. The target of a branch instruction is stored in the BTB table and this target value used for subsequent invocations of the same branch instruction, thereby saving precious cycles. However, this prediction scheme requires that each time an overlayed function is transferred from main memory to SPM, the table has to be flushed. This is essential, otherwise branch instructions will jump to invalid addresses from the previous overlayed functions, thereby crashing the application. The flushing of the table contributes to the performance penalty, albeit by a small amount.

A.3. *CPU Cycle Stalls*

A major contributor to the overall performance penalty is due to processor stalls during transfer of code blocks from main memory to SPM. The command to transfer code blocks is on demand; i.e. the code blocks are not transferred to SPM until they are required for execution. Consequently, the SOVM cannot transfer program control to the first instruction

in the overlayed function until the entire function is transferred. The processor must therefore stall till the transfer command completes and this contributes to the major portion of the penalty. The processor stalls can be avoided if the overlayed functions can be prefetched into the SPM, rather than fetching them when they are absolutely required. The next section discusses a pre-fetch based algorithm which reduces the performance penalty.

B. *Prefetch Aware Mapping*

In the previous section Section A, we discussed that processor stalls during code transfers contributes significantly to the performance overhead. As we shall discuss in Section C, the average performance degradation is around 2.08% for the SDRM technique on the split architecture compared to the performance on the instruction cache only architecture. One way to reduce the number of stalls is to prefetch the overlayed functions into the SPM regions, instead of fetching on demand. Since our architecture uses a direct memory access engine, the processor can continue executing the instructions of the current function while the next function to be called is prefetched, thereby effectively overlapping the processing and communication time.

To identify such prefetch opportunities in a program, we introduce additional nodes in the GCCFG representing computations, the C-nodes. Such nodes are characterized by the computation time in cycles which is determined statically. This is done by summing the cycles required by each instruction in the C-node. If the C-node contains a loop, then we multiply the aggregate cycle time of such instructions with a fixed quantum C equal to 10. The computation time is thus a conservative approximation and does not account for processor stall cycles due to hazards, cache misses or other factors. Figure 9 shows a sample

```
MAIN ( )                    F2 ( )
    F1( )                       FOR
     FOR                            computations..
          F2 ( )                    F6 ( )
    END FOR                         computations..
END MAIN                            F3 ( )
F3 (  )                             WHILE
    …                                    F4 ( )
END F3                              END WHILE
F4()                            END FOR
    …                           computations..
END F4                          F5 ( )
F5()                        END F2
END F5
```

Fig. 9. Example code for prefetching

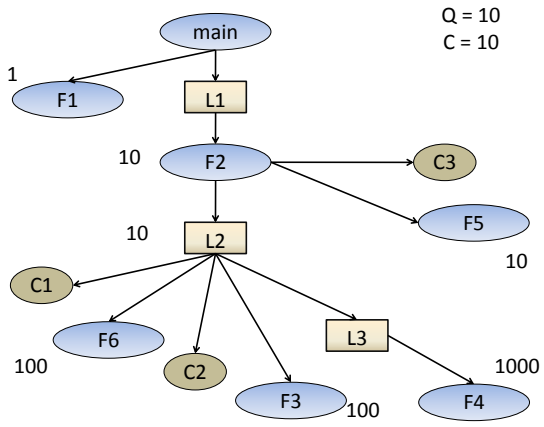

Fig. 10. Global call control flow graph with c-nodes

code with computation instructions and Figure 10 shows the corresponding GCCFG with the C-nodes C1, C2 and C3.

Prefetching is beneficial only if there is more than one node mapped to a region, the nodes have callee-callee-in-loop relationship and the nodes are the same nested level. For callee-callee-no-loop relationship, prefetching is a one time act and there is no benefit in analysis of the corresponding region containing such nodes. Moreover, prefetching is not possible for regions containing caller-callee-in-loop or caller-callee-no-loop relationships, as the prefetched code will overwrite the currently executing code and crash the application.

For the GCCFG in Figure 10, F6 and F3 has callee-callee-in-loop relationship. Thus we can insert prefetching code at the start of C1 and at the start of C2. F3 and F4 also has a callee-callee-in-loop relation but note that F4 is called inside the loop L3 and is not at the same nested level as F3. Hence once we fetch F4, there is no point prefetching it every time as it will just overwrite the code and consume more energy due to extra writes to SPM.

Having identified such nodes with prefetching opportunities, we derive the following cost function for performance penalty for such nodes. The section Section C gives the formula for deriving $latency_{cycles/byte}$.

$$cost_p[v_i, v_j] = (s_i + s_j) \times min(w_i, w_j) \times latency_{cycles/byte} - (C_i + Cj) \quad \text{(VII.1)}$$

For nodes which are not prefetched, the C_i or C_j is nil and the performance penalty which is a function of the size of the functions and their weights will be maximum. The energy cost $cost_e$ is determined by the Algorithm 1. The total interference cost is defined as the product of performance cost $cost\_p$ and the energy cost $cost\_e$ and is given as input to the ILP and the SDRM heuristic.

$$cost[v_i, v_j] = cost_e[v_i, v_j] \times cost_p[v_i, v_j] \quad \text{(VII.2)}$$

The SDRM heuristic using the new cost function is termed as SDRM-prefetch. However, for the same size of scratchpad, the SDRM-prefetch may result in a lesser energy reduction compared to SDRM heuristic. Consider the GCCFG in Figure 10 and suppose a possible mapping given by SDRM in Table V(a). Now because of the C-nodes C1 and C2, and the callee-callee-in-loop relation between F3 and F6, the SDRM-prefetch mapping will generate the mapping shown in Table V(b). While the performance will improve because

TABLE IV
SDRM vs SDRM-prefetch

Region	Function ID
0	F2,F1
1	F4,F5
2	F3
3	F6

(a) SDRM mapping

Region	Function ID
0	F2,F1
1	F4
2	F3,F6
3	F5

(b) SDRM-prefetch mapping

of prefetching of F3 and F6, every fetch of F3 and F6 will now cause additional SPM writes and this will lead to more energy consumption and hence lower reduction. If the SDRM-prefetch does not change the mapping, then the energy reduction will be same as that of SDRM heuristic with at least some performance improvement. Thus there is always a tradeoff between energy reduction and the performance improvement. As we shall see in Section IX, while the tradeoff is not very large for our benchmark applications, they might be substantial for other applications and hence prefetching becomes a major design decision.

VIII. Setup and Models

A. *Experimental Setup*

The instrumented binary is executed on the cycle-accurate *simple-scalar* simulator [26] modeling ARMv5TE instruction set. The details of the instruction set can be obtained from [27]. The simulator has been modified to model an on-chip SPM at same level as the level 1 instruction cache. The system modeled has a 16KB L1 direct mapped instruction cache, 16KB 4-way set associative data cache and a SPM, the size of which can be selected by the system designer. The size is typically changed by writing to a 32 bit register. The SPM is designed to be used as part of the physical map of the system, and is not backed by a level of external memory with the same physical addresses. The L1 instruction cache and the SPM are incoherent i.e. the memory locations are contained either in the SPM or the cache, not in both. The simulator models a low power 32MB SDRAM from Micron, MT48V8M32LF [7], as the main memory.

B. *Energy Model*

TABLE V
Energy per access (.13 μm)

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

We compare only the memory subsystem energy since the processor energy overhead of our technique is very negligible (The overhead in terms of performance is reported in Section C). The energy figures for SPM and I-Cache are given in Table V. We assume $E_{IC-READ/ACCESS}$ and $E_{IC-WRITE/ACCESS}$ to be equal. The energy per memory burst

E_{MBST} is 32.5 nJ [7]. The total energy E_{TOTAL} consumed by the memory system is given by the following equations.

$$E_{TOTAL} = E_{SPM} + E_{I-CACHE} + E_{TOTAL\_MEM}$$

$$E_{SPM} = N_{SPM} \times E_{SPM/ACCESS}$$

$$E_{I-CACHE} = E_{IC-READ/ACCESS} \times \{N_{IC-HITS} + N_{IC-MISSES}\} + E_{IC-WRITE/ACCESS} \times 8 \times N_{IC-MISSES}$$

$$E_{TOTAL-MEM} = E_{CACHE-MEM} + E_{DMA}$$

$$E_{CACHE-MEM} = E_{MBST} \times N_{IC-MISSES}$$

$$E_{DMA} = (N_{DMA-BLOCK} \times E_{MBST} \times 4)$$

C. *Performance Model*

The performance of the application is given by the total simulation time in cycles. This simulation time also includes the processor stalls due to non-prefetched DMA instructions. The L1 cache and SPM access latency is 1 cycle each. We assume a non-pipelined main memory model and transfer data in chunks determined by the memory access bus width equal to 64 bits. The transfer takes place in chunks, where the first chunk takes 18 cycles and the rest of the chunks take 2 cycles each. The total latency is the summation of all the chunks. The effective latency per byte is the ratio of total memory latency and blk_size, where blk_size is the number of bytes to be transferred by the overlay manager from main memory to SPM. The main memory access latency is given by the following set of equations.

$$mem\_lat[0] = 18 \quad [first \quad chunk]$$

$$mem\_lat[1] = 2 \quad [inter \quad chunk]$$

$$chunks = \frac{blk\_size + (bus\_width - 1)}{bus\_width}$$

$$total\_lat = mem\_lat[0] + mem\_lat[1] \times (chunks - 1)$$

$$latency\_cycles/byte = \frac{total\_lat}{blk\_size}$$

D. *Benchmarks Used*

TABLE VI
Details of the benchmark programs

Benchmark	Size(Bytes)	Description
Dijkstra	1588	Shortest Path(network)
Patricia	2904	Routing(network)
Rijndael	21050	Encryption(security)
SHA	2376	Message Digest(security)
Susan	46808	Edge Detection(automotive)
FFT	4688	Signal Processing(telecom)
ADPCM	1436	Audio Compression(telecom)
Blowfish	9308	Cipher(security)

The ARM architecture is popular in handheld media devices and therefore we perform our experiments on a set applications from the MiBench suite [28]. Table VI presents the applications used, along with their respective code sizes. The reported size does not include the scratchpad overlay manager (SOVM) size and shared libraries as they are not subject to our code overlaying.

IX. Experimental Results

A. *Cache-only vs Horizontally Split Architecture*

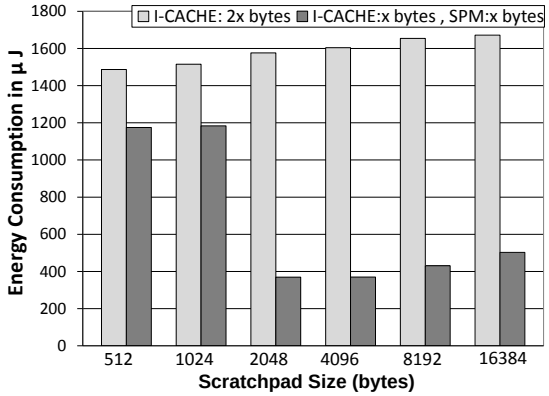

Fig. 11. SHA: Energy comparisons between cache only and horizontally split architecture with SDRM

In this section, we compare our mapping technique for the horizontally split architecture against the cache-only architecture. The cache only architecture consists of 2x bytes of instruction cache while split architecture consists of x bytes of scratchpad and x bytes of instruction cache, where the size of x is varied from 512 bytes to 16384 bytes. Figure 11 shows how the cache only architecture performs in comparison with the split architecture with SDRM technique for *sha* benchmark. Note that although we present the results for only *sha*, similar trends are observed for other benchmarks as well, the only difference being that the trough of split-SDRM curve depends on the optimal size of scratchpad for that particular benchmark.

For small sizes of scratchpad, the critical functions do not fit into the SPM at all and are spilled to cache. Hence there is no significant difference between the cache only and the split architecture. As we increase the size to 2048 bytes, all functions can fit into the scratchpad, 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 scratchpad, we see a significant reduction in energy as all the program code is fetched and executed from the scratchpad instead of the instruction cache. At a larger size of 4096 bytes, all the functions can be mapped onto the SPM at distinct addresses without any overlay. Thus there are no calls to the SPM overlay manager and no runtime performance degradation due to memory transfers. We should therefore have observed a further decrease in energy consumption. However, since we assume a energy model where the energy per access for SPM increases with size, table V, we observe an increase in the total energy consumption with increasing size of the scratchpad memory. As shown in Figure 11 for *sha* benchmark, the split architecture shows a reduction of 77% compared to the cache only architecture and the reduction is maximum at 2048 bytes. Across all the benchmarks, SDRM exhibits an average energy reduction of 35% while SDRM-prefetch exhibits an average energy reduction of 32.3%, at their respective optimal SPM sizes. The performance impact is discussed in Section C.

This experiment demonstrates the effectiveness of a split memory subsystem architecture when supported by an intelligent mapping technique like SDRM. It shows that, 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 scratchpad memory and a smaller instruction cache. The small SPM and the instruction cache will also have a lesser area overhead compared to the original cache. Given such a split architecture, we can then use a pure compiler technique like SDRM requiring just a simple recompilation of the application, with no profiling overhead.

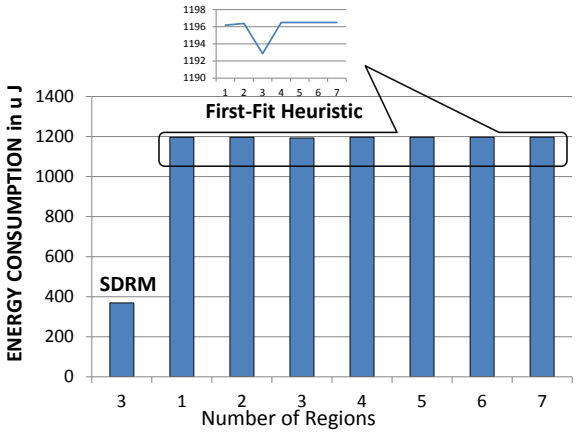

Fig. 12. SHA Benchmark: first-fit heuristic with varying number of variable sized regions

B. *First-Fit vs SDRM for Horizontally Split Architecture*

In this section we compare the total energy consumption of the split architecture between the ILP, SDRM and the first-fit heuristic for various benchmarks. For the first-fit heuristic we assume that the scratchpad is divided into variable sized regions (We found, experimentally that regions of variable size gives better results than equal sized regions). The previous approach does not precisely state a way of finding these region sizes. To be unbiased towards first-fit heuristic, we performed an exhaustive 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. Allocation to the regions was performed in decreasing order of the region size. Figure 12 demonstrates the first-fit energy consumption trend for *sha* as we explore the number of regions for a 2KB scratchpad.

As shown in the graph, there is an optimal number of regions ($r = 3$) for 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 due to higher energy per access of the instruction cache. As we increase the number of regions, more functions will be overlayed and the energy consumption decreases, reaching a local minimum at ($r = 3$). However, if we compare this value with the first bar which indicates the energy consumption for 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 in this paper does not incur this exploration overhead since it simultaneously finds the optimal number of regions and their sizes while solving the mapping problem itself. The first bar in the graph shows the energy results obtained by SDRM for a 2KB scratchpad. The SDRM technique divides the scratchpad 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.

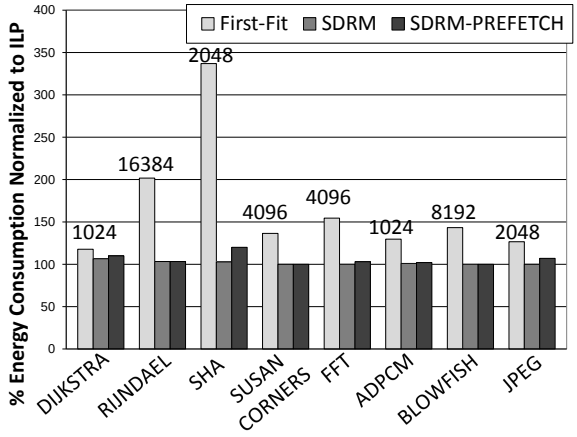

Fig. 13. Energy comparisons between ILP, SDRM, SDRM-prefetch and first-fit for various benchmarks

Figure 13 shows the comparison of energy consumption between SDRM, SDRM-prefetch and first-fit heuristic for various benchmarks from Table VI. The energy results are normalized to the ILP energy results. The optimal number and size of the regions for first-fit are found by exploration as discussed previously. From the graphs, 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 as long as the size of the SPM is greater than the size of the largest function in the benchmark. On an average we observe a 25.9% energy reduction for SDRM compared to the first-fit technique. The SDRM-prefetch technique results in a lesser energy reduction compared to SDRM, but as discussed in Section C, it achieves much better performance improvement. On an average, SDRM-prefetch results in 22% energy reduction compared to the first-fit technique.

C. *Performance Overhead*

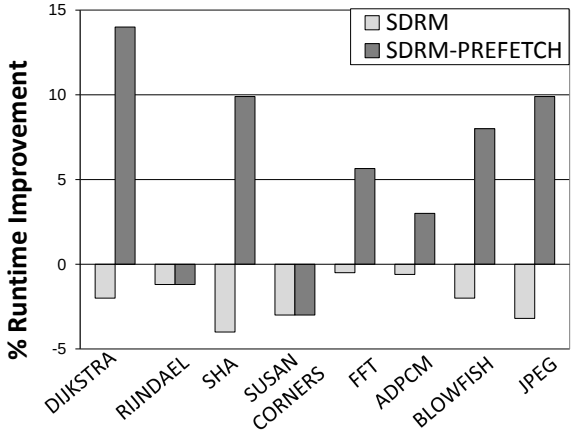

Fig. 14. Performance improvement: SDRM vs SDRM-prefetch

As discussed in Section A, a major contributor to the performance degradation of SDRM overlayed code is the processor stalls during code transfer from main memory to the SPM. To minimize the performance degradation and potentially turn it into a performance enhancement, we modified the cost function to be aware of prefetching opportunities in the program and fed the new cost to SDRM to obtain a new mapping denoted by SDRM-prefetch. Figure 14 shows the comparison between plain SDRM and SDRM-prefetch mappings on a split architecture, normalized to the performance of the instruction cache only architecture. While the average performance degradation is 2.08% for SDRM, SDRM-prefetch exhibits an average performance improvement of 5.78%. Note that we do not see any improvements in *rijndael* and *susan corners* as there are virtually no prefetching opportunities in these benchmarks.

X. Conclusion and Future Work

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 split architecture succeeds in achieving a greater energy reduction against a previous approach and a unified instruction cache only architecture, respectively. Compared to the best performing previously known heuristic our approach achieves an average energy reduction of 25.9%.

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. Our experiments show this reduction to be 35% with a performance degradation of just 2.08%. We also introduced a prefetch aware technique SDRM-prefetch, which turns the performance degradation into a performance improvement by trading off a small amount of energy reduction. By using the SDRM-prefetch on the split architectures, we achieve an average energy reduction if 32.3% and an average performance improvement of 5.78%.

We have many directions for future work. In this work, our static analysis assumes predication when dealing with conditional statements, i.e. both the branches of a *if* statement and all cases of a *switch* statement will be executed with equal probabilities. This limits the energy reduction and performance enhancement and better results can be achieved if a more accurate analysis can be done. Our work also assumes that the scratchpad size should be atleast as large as the size of the largest function in the program. This limitation

can be removed if functions can be outlined and some analysis can be done to find out the degree of outlining. Another focus area to reduce energy would be to bank the available SPM and use compile time inserted instructions to put unwanted banks to sleep. We also plan to enhance our compiler technique for using scratchpads in a multi-tasking environment where sharing the SPM among different processes to reduce context switch overhead would be a challenge.

REFERENCES

[1] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, Gullu, and Stephen C. Thierauf, "A 160-mhz, 32-b, 0.5-w cmos risc microprocessor," *Digital Tech. J.*, vol. 9, no. 1, pp. 49–62, 1997.

[2] "Scratchpad RAM," http://en.wikipedia.org/wiki/Scratchpad_RAM.

[3] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," *CODES '02: Proceedings of the 10th International Symposium on Hardware/software Codesign*, pp. 73–78, 2002.

[4] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid, "Tiny instruction caches for low power embedded systems," *Trans. on Embedded Computing Systems*, vol. 2, no. 4, pp. 449–481, 2003.

[5] Manish Verma. and Peter Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 802–815, Aug. 2006.

[6] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *Trans. on Embedded Computing Systems*, vol. 5, no. 2, pp. 472–511, 2006.

[7] Aviral Shrivastava, Ilya Issenin, and Nikil Dutt, "Compilation techniques for energy reduction in horizontally partitioned cache architectures," *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 90–96, 2005.

[8] Bernhard Egger, Jaejin Lee, and Heonshik Shin, "Scratchpad memory management for portable systems with a memory management unit," *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pp. 321–330, 2006.

[9] Federico Angiolini, Luca Benini, and Alberto Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 318–326, 2003.

[10] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri, "A post-compiler approach to scratchpad mapping of code," *CASES '04:*

Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 259–267, 2004.

[11] Nghi Nguyen, Angel Dominguez, and Rajeev Barua, "Scratch-pad memory allocation without compiler support for java applications," *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 85–94, 2007.

[12] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias, "An integrated hardware/software approach for run-time scratchpad management," *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pp. 238–243, 2004.

[13] Stephan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel, "Assigning program and data objects to scratchpad for energy reduction," *DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe*, p. 409, 2002.

[14] Manish Verma, Lars Wehmeyer, and Peter Marwedel, "Cache-aware scratchpad allocation algorithm," *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, p. 21264, 2004.

[15] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: a first approach," *3rd Workshop on Embedded Systems for Real-Time Multimedia*, pp. 115–120, Sept. 2005.

[16] Oren Avissar, Rajeev Barua, and Dave Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *Trans. on Embedded Computing Systems*, vol. 1, no. 1, pp. 6–26, 2002.

[17] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min, "A dynamic code placement technique for scratchpad memory using postpass optimization," *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 223–233, 2006.

[18] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran, "A novel instruction scratchpad memory optimization method based on concomitance metric," *ASP-DAC '06: Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pp. 612–617, 2006.

[19] Mahmut Kandemir and et al., "Dynamic management of scratch-pad memory space," *DAC '01: Proceedings of the 38th Conference on Design Automation*, pp. 690–695, 2001.

[20] Mahmut Kandemir and Alok Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," *DAC '02: Proceedings of the 39th Conference on Design Automation*, pp. 628–633, 2002.

[21] Lian Li, Lin Gao, and Jingling Xue, "Memory coloring: A compiler approach for scratchpad memory management," *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pp. 329–338, 2005.

[22] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel, "Reducing energy consumption by dynamic copying of instructions onto onchip memory," *ISSS '02: Proceedings of the 15th International Symposium on System Synthesis*, pp. 213–218, 2002.

[23] Peng Zhao and J.N. Amaral, "Function outlining and partial inlining," *17th International Symposium on Computer Architecture and High Performance Computing*, pp. 101–108, Oct. 2005.

[24] James Smith, "A study of branch prediction strategies," *ISCA '81: Proceedings of the 8th Annual Symposium on Computer Architecture*, pp. 135–148, 1981.

[25] Harold N. Gabow and Robert Endre Tarjan, "A linear-time algorithm for a special case of disjoint set union," *STOC '83: Proceedings of the Fifteenth Annual ACM Symposium on Theory of computing*, pp. 246–251, 1983.

[26] "Simplescalar Simulator," http://www.simplescalar.com.

[27] "ARMv5 ARM Architecture version 5 (ARMv5TE)," http://www.arm.com.

[28] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," *IEEE International Workshop on Workload Characterization*, pp. 3–14, 2 Dec. 2001.

[29] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[30] Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[31] Alexandre Eichenberger, John O'Brien, and et al., "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine Architecture," 2006.

[32] Lars Wehmeyer and Peter Marwedel, "Influence of onchip scratchpad memories on wcet prediction," 2004.

[33] Manish Verma, Lars Wehmeyer, and Peter Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 104–109, 2004.