Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors

Manish Verma, *Student Member, IEEE*, and Peter Marwedel, *Senior Member, IEEE*

***Abstract*—Energy consumption is one of the important parameters to be optimized during the design of portable embedded systems. Thus, most of the contemporary portable devices feature low-power processors coupled with on-chip memories (e.g., caches, scratchpads). Scratchpads are better than traditional caches in terms of power, performance, area, and predictability. However, unlike caches they depend upon software allocation techniques for their utilization. In this paper, we present scratchpad overlay techniques which analyze the application and insert instructions to dynamically copy both variables and code segments onto the scratchpad at runtime. We demonstrate that the problem of overlaying scratchpad is an extension of the Global Register Allocation problem. We present optimal and near-optimal approaches for solving the scratchpad overlay problem. The near-optimal scratchpad overlay approach achieves close to the optimal results and is significantly faster than the optimal approach. Our approaches improve upon the previously known static allocation technique for assigning both variables and code segments onto the scratchpad. The evaluation of the approaches for ARM7 processor reports, average energy, and execution time reductions of 26% and 14% over the static approach, respectively. Additional experiments comparing the overlayed scratchpads against unified caches of the same size, report average energy, and execution time savings of 20% and 10%, respectively. We also report data memory energy reductions of 45%–57% due to the insertion of a 1024-bytes scratchpad memory in the memory hierarchy of a digital signal processor (DSP).**

***Index Terms*—Code overlay, memory aware code optimization, scratchpad memory (SPM).**

I. Introduction

CONTEMPORARY portable devices demonstrate the integration of a multitude of conventional devices and feature enhancements. The best example, is a mobile phone featuring Bluetooth, GPRS, and a color display. It can perform a host of functions associated with multimedia devices. For example, it can take digital pictures, make videos, play MP3 files, and can also act as a video game console and PDA. It is known, that the next generation of portable devices will feature faster processors and larger memories, both of which require high-operational power. This is expected to impose severe constraints on already limited electrical energy available in the battery of the portable devices. Consequently, reducing power consumption in portable devices has become a dominant design concern.

Manuscript received July 11, 2005; revised January 8, 2006. This work was supported by Deutsch Forschungsgemeinschaft (German Research Foundation) under Grant Ma 943/8-3.

The authors are with the Department of Computer Science 12, University of Dortmund, Dortmund D-44221, Germany (e-mail: Manish.Verma@cs.uni-dortmund.de; Peter.Marwedel@udo.edu).

Digital Object Identifier 10.1109/TVLSI.2006.878469

Memory subsystems have been demonstrated to consume 50%–75% power budget of the entire system [1], [2]. As a consequence, memory hierarchies are being constructed to reduce the energy dissipation of the memory subsystem. Caches and scratchpad memories represent two contrasting memory architectures. A cache, in addition to the data memory, consists of tag memory and address comparison logic. This enables the cache to automatically exploit the spatial and temporal locality present in the application. However, for embedded systems, caches are inappropriate as these additional components consume a significant amount of energy and on-chip area. Additionally, worst case execution time (WCET) bounds are mostly overestimated for a cache-based system [3].

On the other hand, a scratchpad memory (SPM) comprises of a data memory array and logic for address decoding. The absence of the tag memory and the address comparison logic from the SPM makes it both, area and power efficient [4]. However, a careful assignment of instructions and data is required, by the programmer or the compiler, for their efficient utilization. This inturn allows tighter bounds on WCET prediction of the system as the contents of the SPM are known at all times during the execution [3].

In this paper, we present optimal scratchpad overlay (Opt. SO) and near-optimal scratchpad overlay (Near-Opt. SO) approaches which overlay both instruction segments and variables onto the SPM. The approaches generate overlays [5] or memory objects from an application and solve the scratchpad overlay problem in a two-step process. In the first step, the approaches assign memory objects to the SPM such that the SPM size constraint is respected at all execution time instances. They also determine the spill locations for copying the memory objects on and off the SPM at runtime. A 0–1 integer linear programming (ILP)-based optimal approach is used to determine the best set of memory objects assigned to the SPM and the optimal spill locations. In the second step, the approach computes the addresses of the memory objects assigned to the SPM, such that the SPM space is shared by memory objects which are not required at the same time. The second step is solved optimally through an ILP formulation and near-optimally by using a first-fit heuristic-based approach.

The rest of the paper is organized as follows. The following section presents the related work and compares our approach with the related approaches. Section III presents a motivating example for the readers. The scratchpad overlay approaches are presented in Section IV, which is followed by the presentation of the experimental setup. Section VI presents the evaluation of the scratchpad overlay approaches for two contrasting processor architectures. The paper ends with a conclusion and future work.

1063-8210/$20.00 © 2006 IEEE

II. Related Work

Global register allocation is one of the most researched and fundamental topics in code optimization and compiler construction [5]. A compiler initially generates code assuming an infinite number of symbolic registers which have to be assigned to the limited number of the processor's real registers. Global register allocation attempts to find an assignment of the symbolic registers to the processor's real registers such that the maximum number of symbolic registers is assigned to the real registers. The allocation problem was proven to be NP-complete [6]. Most of the register allocators [7], [8] are based on the graph coloring heuristic [9]. In the recent past, optimal approaches [10], [11] to solve the register allocation problem, have been proposed. Although global register allocation is NP-complete for arbitrary graphs, coloring of graphs found in real-life programs [12], has been demonstrated to be easier. A study by the authors in [11], empirically demonstrated that it takes $O(n^3)$ to optimally solve the register allocation problem for real-life benchmarks.

Dynamic storage allocation (DSA) has been a fundamental part of operating systems for allocating memory to applications [13]. Applications either make requests for memory or release some of the already allocated memory. The job of the allocator is to satisfy requests for memory from applications, such that the total amount of memory required is minimized. The DSA problem has also been proven to be NP-complete [6]. Several heuristic-based allocation approaches (e.g., first-fit, next-fit, best-fit) have been proposed. The authors of [14] present a good survey and comparison of the various approaches.

The research on scratchpad utilization for single process applications can be classified into two broad categories *viz.* static- and overlay-based allocation techniques. In the former, the scratchpad is loaded once at the start and its contents remain invariant during the entire execution period of the application. In contrast, overlay-based allocation techniques partition the application into overlays [5]. These overlays are copied on and off the scratchpad during the execution to capture the dynamic behavior of the application.

Static allocation techniques [15]–[19], can be classified into techniques which allocate only data elements or only instructions or a combination of both data elements and instructions onto the SPM. The authors in [16] and [17], proposed techniques to assign only data elements onto the SPM, while the authors in [19], utilized the SPM only for instructions. An optimal approach for assigning both variables and instruction segments is presented in [18]. In a different approach, authors [15] generated partitioned SPMs customized according to the application.

Overlay-based allocation techniques [1], [20]–[25] can also be similarly classified. References [1], [21], and [22] proposed techniques to dynamically copy overlays of data elements onto the SPM, whereas, the authors in [20], [23], and [24] proposed approaches to assign only instructions onto the SPM. A recent approach [25] assigns both instructions and data onto the SPM.

There are only two approaches [18], [25] known for assigning both instructions and variables onto the SPM. The approach [18] formulates ILP problem and determines an optimal static assignment. However, the approach [18], is constrained for the large applications, as they consist of several hotspots and not all of them can be statically assigned to the SPM. The other approach [25], generates overlays from the application, which are then dynamically copied on and off the SPM at runtime. The scratchpad overlay approaches (*viz.* Opt. SO and Near-Opt. SO) presented in this paper are an extension of the approach [25] by the same authors. The previous approach [25] analyzes local control flow graphs, while the presented approaches analyze the inter-procedural control flow graph of the application. Consequently, the presented approaches are able to determine spill locations across the function boundaries and result in lower energy consumption values.

Fig. 1. Workflow of edge detection application.

The second step of the Opt. SO approach, solves an ILP formulation to determine addresses to the memory objects assigned to the SPM. For large benchmarks and large SPM sizes, it requires long computation time. Consequently, we present the Near-Opt. SO approach which retains the first step of the Opt. SO approach but replaces the ILP formulation of the second step by a first-fit heuristic-based approach. The Near-Opt. SO approach is motivated by the following two empirical observations:

1) optimal solution to the memory assignment problem can be determined in $O(n^{1.3})$ time [26] for most cases;
2) first-fit-based heuristic achieves close to optimal allocation for real-life benchmarks [14].

The Near-Opt. SO approach for our set of benchmarks, achieved close to optimal results and required negligible computation time. In the following section, the benefit of overlaying the SPM is presented with help of an example.

III. Motivating Example

We start with presenting a motivating example to demonstrate that real-life applications consist of multiple hotspots. These hotspots have nonconflicting live ranges and can be overlayed on the SPM to reduce the energy consumption of the application. Fig. 1 presents the workflow of edge detection application, which determines edges in a tomographic image. The application consists of three sequential steps called `GaussBlur`, `ComputeEdges`, and `DetectRoots`. Each of these steps processes a given input image and writes the resulting image as output which is then passed to the next stage in the workflow.

The execution profile of the edge detection application is presented in Fig. 2. We have scaled down the input image to speedup the profiling of the application. A point (x, y) in the figure represents that the xth executed instruction in the instruction trace of the application was fetched from the address y in the memory. The dark regions in Fig. 2 correspond to the execution of the stages of the edge detection application. For example, the largest region in the center of the figure, correspond to the execution of the `ComputeEdges` stage of the application. From Fig. 2, we observe that each stage of the application is a hotspot and that the stages do not interfere with each other. Hence, the contents of each stage can be overlayed onto the SPM. However, the contents of each stage needs to be copied on and off the SPM before entering and after leaving the stage, respectively.

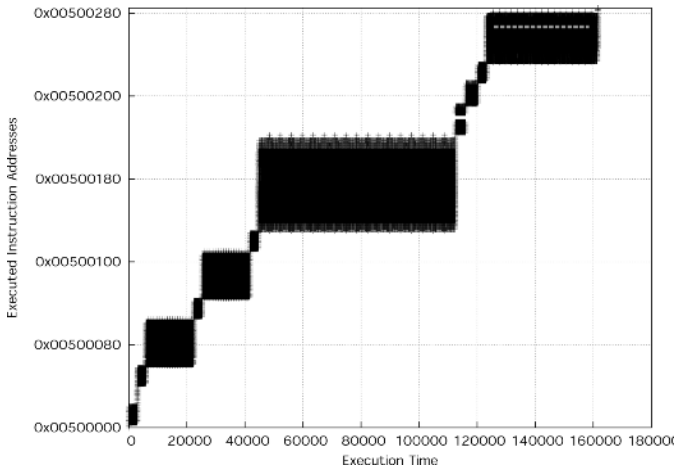

Fig. 2. Execution profile of edge detection application.

TABLE I
EXECUTION AND ACCESS COUNTS FOR FUNCTIONS AND ARRAYS OF EDGE DETECTION APPLICATION

Instruction Memory			Data Memory		
Function	Size (bytes)	Exec. Count	Array	Size (bytes)	Access Count
ReadImage	32	38092	in_image	5120	9400
GaussBlur	324	785313	gb_image	5120	41320
ComputeEdges	144	1137824	ce_image	5120	50720
DetectRoots	224	964147	out_image	5120	5648
WriteImage	20	14411	tmp_image	5120	9960
			Gauss	16	13686
			x_offset	32	36480
			y_offset	32	36480
Total Inst.	744	2939787	Total Data	30800	203694

Now, we would like to determine what fragments of the application should be considered as memory objects or as candidates for overlay on the scratchpad memory. Should the set of memory objects consist of only data elements or only instructions, or a combination of both? In order to obtain the answer to the above question, we compiled the edge detection application for ARM7 processor using an energy-optimizing research compiler (ENCC) [27]. The profile information, gathered by profiling the generated application binary, is presented in Table I. The left-hand side and right-hand side of the table present the profile information for the functions and for the arrays of the application, respectively. The execution count for a function is the sum of the execution counts of every instruction in the function, whereas the access count for an array is the sum of the access counts of each array element.

We make the following observations upon studying Table I. First, the total instruction size is much smaller than the total data size of the application, while the total execution count for instructions is an order of magnitude larger than the total access count for data arrays. This implies that instructions should belong to the set of memory objects as they have high execution counts and consume much less space. Second, the access count per unit size for array `Gauss` is larger than the execution count per unit size for function `WriteImage`. Similarly, the access counts per unit size for arrays `x_offset` and `y_offset` are comparable to that of `ReadImage` and are larger than that of `WriteImage`. This implies that arrays should also be included in the set of memory objects. Moreover, all the access functions of the image arrays in the application are affine functions. Hence, data transfer and storage exploration (DTSE) techniques [28] can be utilized to generate small slices for the image arrays which can be assigned to the small SPMs. We did not consider generating array slices as DTSE techniques are orthogonal to our approach. However, they can be implemented as prepass optimizations in our setup. Finally, we conclude that the best set of memory objects should comprise of data elements, as well as code segments.

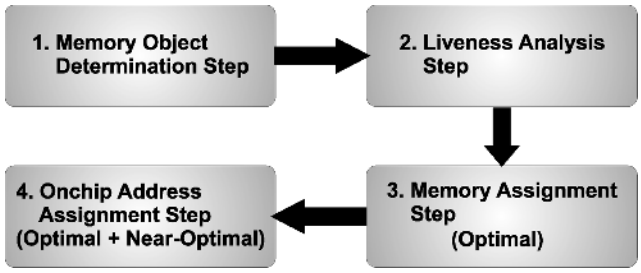

Fig. 3. Workflow of the scratchpad overlay approaches.

IV. SCRATCHPAD OVERLAY APPROACHES

The scratchpad overlay approach copies memory objects onto the SPM at runtime when they are required and copies them off when not required. The scratchpad overlay problem is a weighted version of the global register allocation problem for complex instruction set computer (CISC) architectures. The global register allocation problem for CISCs is also NP-complete [26] and, as a result, the scratchpad overlay problem is also NP-complete. In order to efficiently solve the scratchpad overlay problem, we break it into two smaller problems. The first problem assigns memory objects to the SPM or to the main memory and also determines the optimal locations for the insertion of the spill code. The second problem computes the addresses of the memory objects assigned to the SPM. Unfortunately, both the problems are known to be NP-complete. The Opt. SO approach computes optimal solutions for both the first and the second problems. In contrast, the Near-Opt. SO approach computes optimal and near-optimal solutions for the first and the second problems, respectively.

The scratchpad overlay problem is solved using the workflow shown in Fig. 3. In the first step, variables and code segments from the application code are identified as overlays or memory objects. Liveness analysis is performed in the second step to determine the live ranges of these memory objects. In the third step, assignment of memory objects to the SPM and optimal spill locations are determined. In the final step, an optimal approach or a first-fit heuristic-based near-optimal approach is used to compute the addresses of the memory objects assigned to the SPM.

A. Memory Objects

We consider the following set of variables and code segments as candidates for scratchpad overlay:

- global variables including *scalar* and *non-scalar* variables;
- code segments including *functions* and *traces*.

We do not consider scalar local variables as candidates for scratchpad overlay, as we assume that frequently accessed scalar variables will be assigned to the registers by a good register allocator. The entire stack [18] or a per-function stack frame [16], can be used as memory objects for overlay. Stack is not considered as a memory object in this approach, though, it is a part of the future work. Global variables are considered as they occupy space in the data memory. A trace is a frequently executed straight-line path consisting of basic blocks which are laid out contiguously in memory [29]. Traces improve the processor performance by enhancing the spatial locality present in the program code. We do not generate traces for small functions as it is not beneficial to break them into smaller traces. The above set of candidates is termed overlays or memory objects (MO). In the following section, we describe the computation of live range of each memory object.

B. Liveness Analysis

Liveness analysis is performed on the inter-procedural control flow graph (IPCFG) $G(N, E)$ of the application. The node set N, is the set of basic blocks present in all the functions of the application, and the edge set E represents the flow edges connecting the basic blocks. The edge set E includes the additional flow edges between basic blocks of the caller and the callee functions which represent the call to and the return from the callee function. The concept of DEF–USE chains [30] is extended to compute the liveness of memory objects. A reference to a memory object can be classified as a DEF, a MOD, or a USE. If a reference assigns a value to all the elements of a memory object, then it is classified as a DEF. If only some elements but not all are being assigned, then the reference is assumed to be a MOD. Any reference reading a value of the element(s) of a memory object is assumed to be a USE.

The nodes of the IPCFG are attributed with DEF–MOD–USE information for all memory objects. Both static and profiling-based methods are utilized to classify references to a memory object. Profiling information is used to differentiate between DEF and MOD references while static analysis is used to determine basic blocks containing USE references. If insufficient profile information is available, then a reference is conservatively classified as a MOD reference. Moreover, if a reference to a memory object lies in an unexecuted region of the application, then it is assumed to be a USE reference. This guarantees the correctness of the overlay approaches for all possible input sets. Since code segments are always Read only, traces and functions can only have USE references. Consequently, the live range of each trace starts from the root node of the IPCFG and ends at its last USE reference. A fixed-point iterative algorithm is then used to compute the live range of each memory object. In the following subsection, we describe the ILP formulation of the memory assignment problem.

C. Optimal Memory Assignment

The memory assignment problem is formulated such that the memory objects are assigned to the SPM on the edges rather than at the nodes of the IPCFG. The edge-based formulation enables the determination of the optimal points for the spill code insertion. We define the following static attributes ($\mathrm{Attrib_{STATIC}}$) for every memory object on each edge of the IPCFG:

$$\mathrm{Attrib_{STATIC}} = \{\mathrm{DEF}, \mathrm{MOD}, \mathrm{USE}, \mathrm{CONT}\}$$

where a DEF attribute is defined on every edge originating from a node with a DEF attribute. In contrast, MOD or USE attributes are defined on all edges entering a node with MOD or USE attribute, respectively. If a memory object is live on an edge, then the CONT attribute for the memory object is defined at that edge. In a scenario, where an edge can be assigned more than one static attribute for a memory object, the following priority order is used to determine the appropriate attribute:

$$\mathrm{DEF} > \mathrm{MOD} > \mathrm{USE} > \mathrm{CONT}.$$

In addition to the static attributes, spill attributes ($\mathrm{Attrib_{SPILL}}$) are defined on edges to model appropriate spilling of memory objects.

$$\mathrm{Attrib_{SPILL}} = \{\mathrm{LOAD}, \mathrm{STORE}\}.$$

The LOAD attribute is defined on edges which have MOD, USE, or CONT attribute defined, or which originate from a diverge node. A diverge node is a node whose out-degree is greater than 1. Similarly, the STORE attribute is defined on edges which have a DEF attribute defined or which enter a merge node. A merge node is a node whose in-degree is greater than 1. We would like to state that a spill attribute can be defined only on those edges where a static attribute is already defined.

Next, we define a binary variable $x^i_{j\,k}$ representing the assignment of memory object mo_k to the SPM on edge e_i

$$x^i_{j\,k} = \begin{cases} 1, & \text{if } mo_k \text{ is present on SPM at edge } e_i \text{ and the operation corresponding to attribute } at_j \text{ is performed on edge } e_i \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

where $e_i \in E$, $at_j \in \mathrm{Attrib_{STATIC}} \cup \mathrm{Attrib_{SPILL}}$ and $mo_k \in MO$. For example, if the value of binary variable $x^i_{\mathrm{USE}\,k}$ is equal to 1, then the memory object mo_k is present on the SPM at the edge e_i, and is also used (USE) on the same edge. Similarly, if the value of $x^i_{\mathrm{LOAD}\,k}$ is equal to 1, then the memory object mo_k is spill-loaded onto the SPM at edge e_i. We first describe the objective function and then, the constraints present in the proposed ILP formulation. The objective function represents the energy savings that can be achieved by overlaying the SPM. The energy savings (objective function) need to be maximized in order to minimize the energy consumption of the system

$$E = \sum_i \sum_k \Big\{ E_{\mathrm{profit}}(i, j, mo_k) * x^i_{j\,k} - E_{\mathrm{load\_cost}}(i, mo_k) * x^i_{\mathrm{LOAD}\,k} - E_{\mathrm{store\_cost}}(i, mo_k) * x^i_{\mathrm{STORE}\,k} \Big\} \tag{2}$$

where $E_{\mathrm{profit}}(i, j, mo_k)$ is the energy savings achieved by assuming that memory object mo_k is present on the SPM at edge e_i. $E_{\mathrm{load\_cost}}(i, mo_k)$ and $E_{\mathrm{store\_cost}}(i, mo_k)$ are the energy overheads of spilling memory object mo_k to and from the SPM at edge e_i, respectively. For the sake of brevity, we

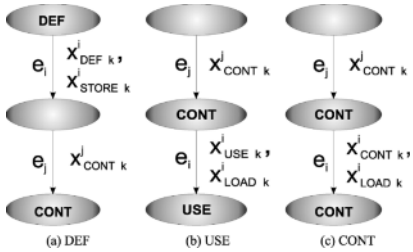

Fig. 4. Flow constraints

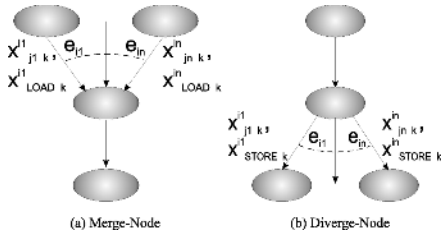

Fig. 5. Flow constraints

refrain from explaining the computation of E_{profit}, $E_{\text{load\_cost}}$, and $E_{\text{store\_cost}}$, computed using an accurate energy model [31].

Constraints have to be added to prevent the binary variable $x^i_{j\,k}$ from assuming arbitrary values and in order to obtain a legitimate solution to the memory assignment problem. We first explain the flow constraints that are added to maintain a legal flow of liveness of memory objects. The flow constraints are repeated for all memory objects. The following is a DEF-constraint which is added for all edges with DEF attribute:

$$x^i_{\text{DEF}\,k} - x^j_{\text{CONT}\,k} - x^i_{\text{STORE}\,k} = 0 \qquad \forall mo_k \in MO. \quad (3)$$

In the above constraint, edge e_i [refer to Fig. 4(a)] contains a DEF attribute, while edge e_j is chosen such that the source node of edge e_j is same as the target node of edge e_i. Informally, the DEF-constraint states that if a memory object mo_k is defined (DEF) on the SPM on an edge e_i, then it can continue (CONT) to remain on the SPM on the following edge e_j or it can be spill-stored (STORE) to the main memory on the edge e_i. Similarly, MOD-constraints or USE-constraints are added for edges e_i with MOD or USE attribute defined.

$$x^i_{\text{USE}\,k} - x^j_{\text{CONT}\,k} - x^i_{\text{LOAD}\,k} = 0 \qquad \forall mo_k \in MO \quad (4)$$

$$x^i_{\text{MOD}\,k} - x^j_{\text{CONT}\,k} - x^i_{\text{LOAD}\,k} = 0 \qquad \forall mo_k \in MO. \quad (5)$$

In each of the above constraints, edge e_j [refer Fig. 4(b)] is chosen such that the target node of edge e_j is the same as the source node of edge e_i. Informally, the USE-constraint states that if a memory object mo_k is being used (USE) on the SPM on an edge e_i, then it was already continuing (CONT) on the SPM on a previous edge e_j or it was spill loaded (LOAD) on the edge e_i. A similar explanation exists for the MOD-constraint. The following flow constraint is added for edges with CONT attribute:

$$x^i_{\text{CONT}\,k} - x^j_{\text{CONT}\,k} - x^i_{\text{LOAD}\,k} = 0 \qquad \forall mo_k \in MO. \quad (6)$$

As shown in Fig. 4(c), edge e_i contains the CONT attribute, while edge e_j is an edge whose target node is the source node of edge e_i. The informal explanation states that if a memory object mo_k is continuing (CONT) on the SPM on an edge e_i, then it was already continuing (CONT) on a previous edge e_j or it was spill loaded (LOAD) onto the SPM on the edge e_i. The following flow constraints (7)–(10) are added to ensure a legal flow of liveness on merge and diverge nodes, respectively. More importantly, the constraints ensure an optimal spill code placement [11]. The following merge-node constraints are added for all merge nodes:

$$x^i_{\text{LOAD}\,k} - x^i_{j\,k} \le 0 \qquad \forall e_i \in \{e_{i1}, \ldots, e_{in}\}$$
$$at_j \in \{at_{j1}, \ldots, at_{jn}\} \quad (7)$$

$$x^{i1}_{j1\,k} = \cdots = x^{in}_{jn\,k} \qquad \forall mo_k \in MO \text{ s.t.}$$
$$at_{j1}, \ldots, at_{jn} \in \text{Attrib}_{\text{STATIC}}. \quad (8)$$

In the above constraints, edges $e_{i1}, \ldots, e_{in}$ [cf. Fig. 5(a)] constitute all the edges entering a merge node. The first constraint (7) ensures that if a memory object mo_k is spill loaded (LOAD) on an edge e_i, then it must be assigned to the SPM on the same edge. The second constraint (8) ensures that if a memory object mo_k is assigned to the SPM on one of the edges entering the merge node, then it must be assigned on each of the remaining edges. For all the diverge nodes, the following constraints (diverge-node constraints) are added:

$$x^i_{\text{STORE}\,k} - x^i_{j\,k} \le 0 \qquad \forall e_i \in \{e_{i1}, \ldots, e_{in}\}$$
$$at_j \in \{at_{j1}, \ldots, at_{jn}\} \quad (9)$$

$$x^{i1}_{j1\,k} = \cdots = x^{in}_{jn\,k} \qquad \forall mo_k \in MO \text{ s.t.}$$
$$at_{j1}, \ldots, at_{jn} \in \text{Attrib}_{\text{STATIC}}. \quad (10)$$

As shown in Fig. 5(b), edges $e_{i1}, \ldots, e_{in}$ denote all the edges emerging from a diverge node. In order to maintain the legality of flow, if a memory object mo_k is assigned to the SPM on one of the edges exiting a diverge node, then it must be assigned to the SPM or spill-stored (STORE) to main memory on each of the remaining edges. Finally, we append the scratchpad size constraint which ensures that the aggregate size of all memory objects assigned to the SPM on an edge should be less than the size of the SPM. The following constraint is added for all edges where a memory object mo_k is being defined (DEF) or spill-loaded (LOAD):

$$\sum_k x^i_{j\,k} * \text{Size}(mo_k) \le \text{ScratchpadSize} \qquad \forall e_i \in E. \quad (11)$$

A commercial ILP solver [32] is used to obtain an optimal assignment of memory objects to the SPM which maximizes the energy savings while satisfying the above constraints. For every edge $e_i \in E$ and for every memory object $mo_k \in MO$, we need a constant number of binary variables. Consequently, the total number of binary variables in the 0–1 ILP formulation is

$O(|MO| * |E|)$. The maximum and the average runtimes of the ILP solver for all our experiments were found to be 1333.0 and 9.9 CPU s on a Sun Sparc 1300-MHz compute machine, respectively. We have computed the assignment of the memory objects onto the SPM. However, the scratchpad overlay problem is solved only when we have computed the addresses of the memory objects assigned to the SPM. The ILP formulation to compute the addresses of the memory objects, is presented in the following subsection.

D. Optimal Address Assignment

In the previous step, an implicit assumption was made while formulating the memory assignment problem. The assumption was that if the aggregate size of the memory objects assigned to the SPM on each edge was less than the scratchpad size, then the on-chip addresses can be computed for those memory objects. This assumption can fail due to a bad address assignment strategy, which causes fragmentation of the SPM address space. As a result, memory objects cannot be assigned on-chip addresses, despite the scratchpad size constraint being satisfied. The problem of on-chip address assignment is trivial if all the memory objects are of the same size. However, the problem becomes NP-complete when the memory objects are of different sizes [6]. In this step, we formulate the address assignment problem as an ILP problem to compute a valid solution.

In order to compute the address of a memory object, we compute the offset of its start address from the base address of the SPM. The integer variable O^i_j represents the offset of the memory object mo_j at the edge e_i and it satisfies the following constraint:

$$0 \leq O^i_j \leq \text{ScratchpadSize} - \text{Size}(mo_j). \tag{12}$$

We start with the description of the constraints present in the ILP formulation. Satisfying one of the following two constraints ensures that the offsets of no two memory objects defined at the same edge overlap with each other:

$$O^i_j - O^i_k \geq \text{Size}(mo_k) \qquad \textbf{XOR} \tag{13}$$
$$O^i_k - O^i_j \geq \text{Size}(mo_j). \tag{14}$$

The first constraint (13) of the above set of constraints, implies that on edge e_i the start address (O^i_j) of the memory object mo_j is greater than the end address ($O^i_k + \text{Size}(mo_k)$) of memory object mo_k. The second constraint (14) implies the reversed placement of the memory objects. The XOR operator in the above set of constraints cannot be modeled using linear programming. Hence, we add a binary variable $u^i_{j\,k}$ to linearize the set of constraints

$$u^i_{j\,k} = \begin{cases} 0, & \text{constraint (13) is to be satisfied} \\ 1, & \text{constraint (14) is to be satisfied.} \end{cases} \tag{15}$$

The following is the linearized form of the above set of constraints with $\mathbf{L}$ being a sufficiently large constant:

$$O^i_j - O^i_k + \mathbf{L} * u^i_{j\,k} \geq \text{Size}(mo_k) \qquad \forall e_i \in E \tag{16}$$
$$O^i_k - O^i_j - \mathbf{L} * u^i_{j\,k} \geq \text{Size}(mo_j) - \mathbf{L} \qquad \forall e_i \in E. \tag{17}$$

The above set of constraints is repeated for all pairs of memory objects which are assigned to the SPM on edge e_i. Subsequently, they are also repeated for all edges $e_i \in E$ with more than one memory object assigned to the SPM. Next, a constraint is added to restrict the offset of a memory object mo_k to the same value for all the edges on which it is assigned to the SPM

$$O^i_k - O^j_k = 0 \qquad \forall e_i, e_j. \tag{18}$$

In the above constraint, edges e_i and e_j are chosen such that source node of edge e_j is the target node of edge e_i. Any change in the offsets of the memory object mo_k on edges e_i and e_j is captured using the following binary variable:

$$v^{ij}_k = \begin{cases} 1, & \text{if } O^i_k \neq O^j_k \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

The unit value of the variable v^{ij}_k would imply an invalid solution to the address assignment problem. Equation (18) is transformed to the following form after the insertion of the binary variable v^{ij}_k:

$$O^i_k - O^j_k - \mathbf{L} * v^{ij}_k = 0 \qquad \forall e_i, e_j \in E. \tag{20}$$

The above constraint is repeated for all memory objects assigned to SPM on both the edges $e_i, e_j \in E$ and also for all such valid pair of edges. A valid solution is characterized by the fact that the offsets of memory objects on all pair of edges remain invariant. The summation of the binary variable v^{ij}_k for all valid pairs of edges and for all memory objects is denoted as the objective function of the ILP formulation

$$\sum_i \sum_j \sum_k v^{ij}_k. \tag{21}$$

For a valid solution, the value of the objective function should be zero which is achieved by minimizing the objective function. The ILP formulation consists of both binary and integer variables. The number of integer variables in the above formulation is $O(|MO| * |E|)$, while the number of binary variables is $O(|MO| * |E|^2)$. The problem is solved using the branch and bound technique of the ILP solver [32], which can take substantial time for certain problem instances having a large number of variables.

E. Near-Optimal Address Assignment

We propose an algorithm which uses the first-fit heuristic [6] for assigning addresses to the memory objects. The algorithm assigns addresses to the memory objects on every edge which are allocated to the SPM and are not already assigned an address. The first-fit heuristic is used to determine an available address region for the memory object. A pseudo-code for the address assignment algorithm is presented in Fig. 6.

We implemented the variant of the first-fit heuristic which divides the SPM address space into $|MO|$ variable sized regions. In order to reduce unused address space, the start boundary of an empty region is adjusted to the end address of its previous region

```
     void AddressAssignment() {
1    for (k = 1; k ≤ |MO|; k++) {
2     for (i = 1; i ≤ |E|; i++) {
3      /* Is MO mo_k mapped to the SPM on edge e_i? */
4      if (x^i_DEF k || x^i_LOAD k) {
5       /* Does MO mo_k already have a valid address? */
6       if (AddressVector[k] == INVALID) {
7        /* No, compute a valid address using first-fit
            heuristic. */
8        address = FirstFitAddress(i, k);
9        if (address == INVALID) {
10        /* First-Fit couldn't find a valid address then
             make mo_k offchip. */
11        SetMemoryObjectOffchip(k);
12       } else {
13        /* Assign the valid address to mo_k. */
14        AddressVector[k] = address;
15       }
16      }
17     } else if (x^i_MOD k || x^i_USE k || x^i_CONT k) {
18      /* Does MO mo_k already have a valid address? */
19      if (AddressVector[k] == INVALID) {
20       /* Get Address from one of the previous edges. */
21       AddressVector[k] = AddressPreviousEdge(i, k);
22      }
23     } else {
24      AddressVector[k] = INVALID;
25     }
26    }}}}
```

Fig. 6. First-fit heuristic-based address assignment algorithm.

which is assigned to a memory object. The first-fit heuristic assigns a memory object the first empty region which can accommodate the memory object. This might lead to the problem of fragmentation as some SPM space may remain unused. The address assignment algorithm calls the first-fit heuristic on an edge only when a memory object is either defined or loaded onto the SPM and the memory object does not already have a valid address on the same edge. If the first-fit heuristic fails to assign an address to the memory object, then the memory object is assigned to the main memory for its entire lifetime. If a memory object is used, modified, or continued on an edge and does not have a valid address, then it is assigned an address from one of the immediately previous edges. The AddressPreviousEdge routine also checks if the addresses on all the immediately previous edges to the current are equal, otherwise the error flag is set. Our implementation of the first-fit heuristic has the runtime complexity of $O(|E| * |MO|)$. Consequently, the runtime complexity of the address assignment algorithm is $O(|E|^2 * |MO|^2)$. The algorithm required negligible (less than 1 CPU seconds) computation time for all our experiments. Next, we describe the experimental setup used to conduct experiments.

V. Experimental Setup

The evaluation of the proposed approaches is performed for two contrasting processors. The first one is a RISC processor (ARM7TDMI), while the second one is a low-power digital signal processor (M5 DSP). The processor architectures and their energy models are briefly presented in the following subsections.

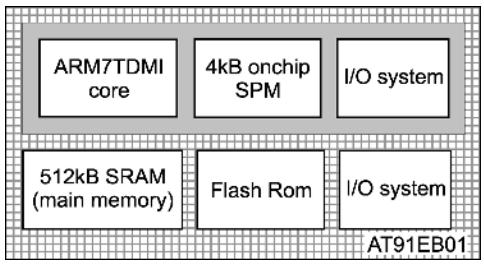

Fig. 7. ATMEL evaluation board.

TABLE II
Energy/Access and Access Times for Memories

Memory	Access width	Energy per access (uJ)	Access Time (CPU Cycles)
SPM	–	1.2	1 cycle
Main Memory	1 byte	15.4	2 cycle
Main Memory	2 bytes	24.0	2 cycle
Main Memory	4 bytes	49.3	4 cycle

A. ARM7TDMI

ARM7TDMI is based on a simple RISC processor with a three-stage pipeline and is the most common processor used in low-power embedded devices. The processor is based upon Von Neumann architecture as it features a unified memory for storing both instructions and data. Our experiments are based upon an ARM7TDMI evaluation board (AT91EB01) [33], featuring an ARM7TDMI processor along with an on-chip 4-kB SPM. In addition, the board contains a 512-kB on-board SRAM which acts as the main memory and a Flash ROM for storing the startup code. Fig. 7 presents the top-level diagram of the evaluation board. Detailed energy measurements were performed to determine an instruction level energy model [31] with an accuracy of 98%. In this work, we use the processor simulator (ARMulator [34]) and the accurate energy model [31] for computing the energy consumption of the system. Table II presents the energy per access and the access times of the SPM and the main memory. The experiments in Section VI-A, compare the SPM allocation approaches on an energy consumption metric for different scratchpad sizes. However, the energy model [31] is valid for ARM7TDMI evaluation board with a 4-kB SPM. Therefore, we have used a single energy per access value for all SPMs sized between 64 bytes and 4 kB. This is justified because energy per access values for small SPMs are comparable. Moreover, in our setup the energy per access for the SPM (cf. Table II) is significantly smaller than those for the main memory.

B. M5 DSP

The M5 DSP was designed using a platform-based hardware/software codesign methodology introduced in [35]. The platform, depicted in Figs. 8 and 9 consists of a fixed control processing part (scalar engine) and a scalable signal processing part

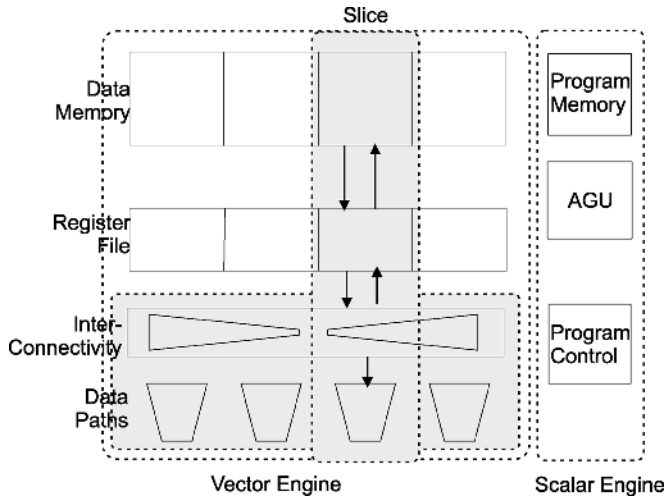

Fig. 8. M5 DSP.

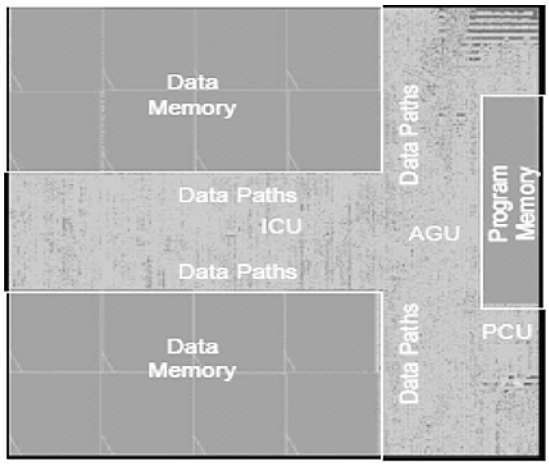

Fig. 9. Layout M5 DSP.

(vector engine). The functionality of the data paths in the vector engine can be tailored to suit the application.

The vector engine consists of P slices, where each slice comprises of a register file and a data path. The interconnectivity unit (ICU) connects the slices with each other and to the control part of the processor. All the slices are controlled using the single instruction multiple data (SIMD) paradigm and are connected to a 64-kB data memory featuring a read and a write port for each slice.

The scalar engine consists of program control unit (PCU), address generation unit (AGU) and a program memory. The PCU performs operations like jumps, branches, and loops. It also features a zero-overhead loop mechanism supporting two-level nested loops. The AGU generates addresses for accessing the data memory.

The processor was synthesized for a standard-cell library by Virtual Silicon for the 130-nm 8-layer-metal UMC process using Synopsys Design Compiler. The resulting layout of the M5 DSP is presented in Fig. 9. The total die size was found to be 9.7 mm^2 with data memory consuming 73% of the total die size.

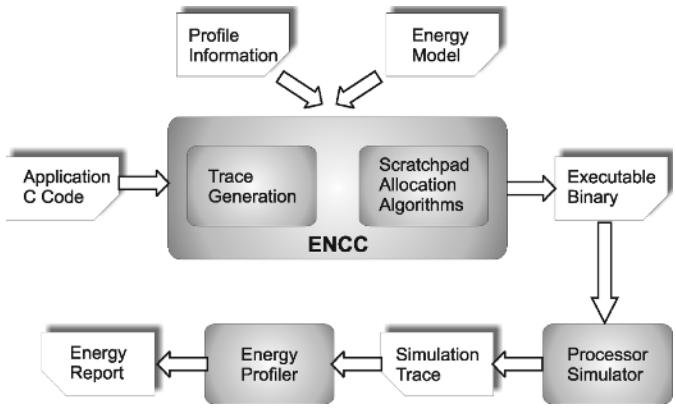

Fig. 10. Experimental workflow.

In this work, we insert a small SPM in between the large data memory and the register file. The SPM is used to store only data elements from data arrays used in the application. The energy consumption of the entire system could not be computed as the instruction-level energy model for the system is currently unavailable. We used a memory energy model from UMC to compute the energy consumption of the data memory subsystem. However, due to copyright reasons, we are forbidden from reporting exact energy values. Consequently, we will report normalized energy values for the data memory subsystem of the M5 DSP. Moreover, the accesses to the SPM and the data memory require only one CPU cycle without any wait states.

C. Experimental Workflow

The experiments for the ARM7-based systems were conducted according to the workflow presented in Fig. 10. In the first step, the benchmark programs are compiled using an energy optimizing C compiler [27]. Within the compiler backend, first the I-cache optimization technique called trace generation [29] is applied. Then, one of the scratchpad allocation approaches (i.e., SA [18], Opt. SO, and Near-Opt. SO) is used to assign memory objects to the SPM. The C compiler generates the executable binary which contains instructions to utilize the SPM. The executable binary is then fed into the processor simulator (ARMulator [34]) to obtain a sequence of executed instructions. Finally, the energy consumption of the system is computed using the instruction sequence and the energy models [31]. The energy consumption of systems with cache-based memory hierarchy is computed using the same workflow.

The experiments for the M5 DSP-based systems were conducted according to the workflow similar to the one presented in Fig. 10. A research compiler without the trace generation step is used to generate the executable binary of the application. The executable binary is fed into the processor simulator to obtain the application execution statistics. The statistics include the number and type of accesses to various memories and the total execution time of the application. Finally, the data memory energy consumption is computed using the memory energy model from UMC.

TABLE III
BENCHMARK PROGRAM FOR ARM7-BASED SYSTEMS

Benchmark	Code Size (bytes)	Data Size (bytes)
adpcm	804	4996
edge detection	908	7792
histogram	704	133156
mpeg4	1524	58048
multisort	636	2020
dsp	2784	61272
media	3280	75672

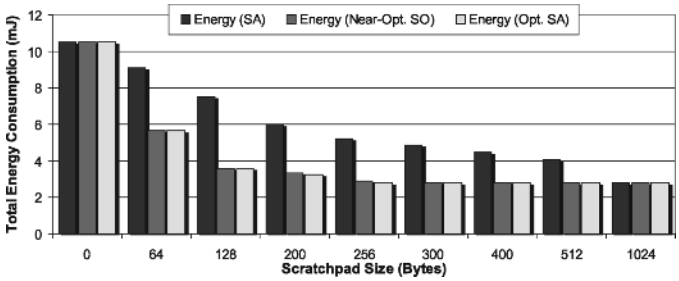

Fig. 11. Edge Detection: Energy comparison of scratchpad allocation approaches.

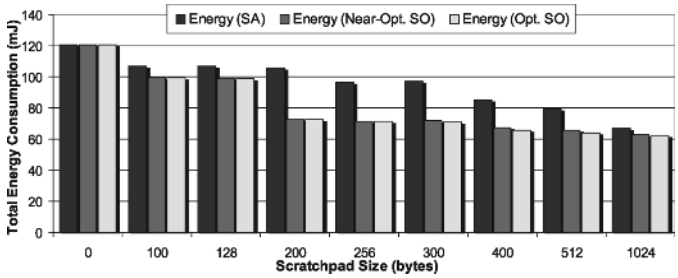

Fig. 12. DSP: Energy comparison of scratchpad allocation approaches.

VI. EXPERIMENTAL RESULTS

The proposed techniques for ARM7-based systems were evaluated for an assorted set of benchmarks from the Mediabench [36] and the UTDSP [37] benchmark suite. In addition, large applications were formed by combining several independent small routines as we believe that the real-life applications are also constructed using the small specialized routines. Table III presents the set of benchmarks used to evaluate the scratchpad overlay techniques. The table also presents the code and data sizes of the applications. For M5 DSP, we used DSP routines from the DSPStone [38] benchmark suite. For the experiments, all memory objects are by default assigned to the main memory unless they are assigned to the SPM by the scratchpad allocation approaches. In that case, the memory objects are copied from the main memory to the SPM. In the following subsection, we present the evaluation of the scratchpad overlay techniques for the ARM7-based system.

A. Experimental Results (ARM7)

A comparison of the scratchpad allocation techniques *viz.* SA, Opt. SO, and Near-Opt. SO for edge detection and DSP benchmarks is presented in Figs. 11 and 12, respectively. The figures present the total energy consumption of the ARM7-based systems when the SPM present in the memory hierarchy is allocated using the scratchpad allocation techniques. From Figs. 11 and 12, we make the following observations:

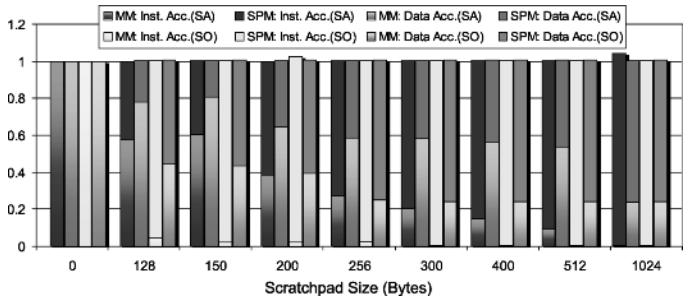

Fig. 13. Edge Detection: Comparison of SA approach versus Near-Opt. SO approach for memory accesses.

First, from both the figures, we observe that the energy consumption values for SA approach monotonically decreases with the increase in scratchpad size. The energy consumption values for Opt. SO and Near-Opt. SO approach decrease faster than those for SA approach. The energy values, for edge detection benchmark (cf. Fig. 11), reach a threshold value at 256 bytes and, thereafter, remain constant. For the SA approach, larger scratchpad size implies that more memory objects can be statically allocated onto the SPM and lower will be the energy consumption. This justifies the monotonically decreasing energy values for the SA approach. Whereas, the scratchpad overlay approaches share the SPM among memory objects with nonconflicting life-times and result in lower energy values than for the SA approach. The energy consumption values become constant when no additional memory objects can be overlayed on the SPM.

Second, we observe that the energy values for Near-Opt. SO and Opt. SO approaches (cf. Fig. 11) at 256-bytes SPM is equal to that for SA approach at 1024 bytes. Similarly, for DSP benchmark (refer to Fig. 12) the energy consumption for the scratchpad overlay approaches at 512 bytes of SPM is equal to that for the SA approach at 1024 bytes of SPM. We can, therefore, conclude that the scratchpad overlay approaches efficiently utilize the SPM. Finally, from both the figures we observe that the energy consumption values for the Near-Opt. SO are very close to those for the Opt. SO. This implies that the proposed near-optimal scratchpad overlay approach performs fairly close to the optimal approach. Next, we present a fine-grained comparison of the SA approach and the Near-Opt. SO approach.

Next, we compare SA and Near-Opt. SO approach in terms of memory accesses for the edge detection benchmark. Fig. 13 presents the comparison for the two scratchpad allocation approaches. The memory accesses for each allocation approach is classified into four categories, depending upon the access type and accessed memory. The memory accesses of a particular access type are normalized according to the main memory accesses of the corresponding access type for a system without an SPM. The series labels presented the legend box of Fig. 13 are generated according to the following regular expression:

$$\{\text{Memory}\} : \{\text{Access Type}\}(\{\text{Allocation Approach}\}). \quad (22)$$

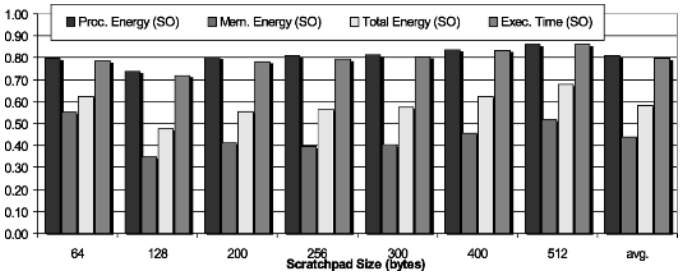

Fig. 14. Edge Detection: Comparison of SA approach versus Near-Opt. SO approach.

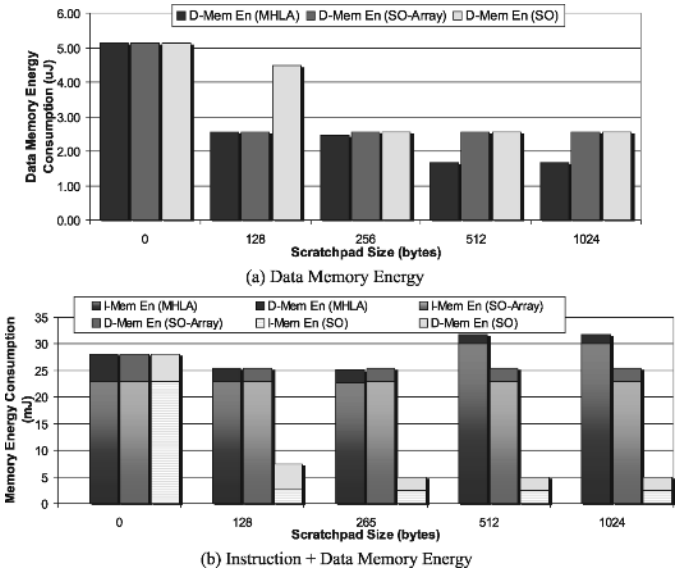

Fig. 15. Edge Detection: Comparison of SO approach versus MHLA approach.

For example, the series label MM: Inst. Acc.(SA) implies that the series represent normalized instruction accesses to the main memory when SA approach is used to allocate the SPM present in the system. The bars in the figure denote the total instruction or data accesses made during the execution of edge detection benchmark. The stacked bars also present the decomposition of the accesses into the accesses to the main memory and to the SPM. From the above figure, we make the following observations.

First, the stacked bars for instruction and data access, representing normalized memory accesses, are larger than the unit value. This is because additional instruction and data accesses are required for copying the memory objects on and off the SPM. Second, the SO approach efficiently utilizes the SPM for caching instruction segments. For example, a 128-byte SPM allocated using the SO approach is able to cache 98% of all instruction accesses. Whereas, the same sized SPM allocated through SA approach is able to cache only 50% of the total instruction accesses. Third, the SPM allocated using the SO approach can also cache a higher percentage of data accesses than that by the SPM allocated using SA approach. However, the percentage of cached data accesses is lower than cached instruction accesses for both the allocation approaches. This is due to the fact that instruction segments (i.e., traces and functions) are smaller and more frequently accessed than data variables. Therefore, we conclude that the SO approach more efficiently utilizes the SPM than the SA approach.

Fig. 14 presents the normalized energy consumption values and performance values of the edge detection benchmark allocated using the Near-Opt. SO approach. The components of total energy consumption, process, and memory energy consumption are also presented in the figure. The energy and performance values for the benchmark allocated using the SA approach are denoted as the unit valued baseline. From Fig. 14, we observe that both the energy values and the performance values for the Near-Opt. SO approach are less than the corresponding values for the SA approach. Inspite the fact that the processor executes additional instructions for copying memory objects on and off the SPM, the overlay approaches are able to save both energy and execution time. The accesses to the SPM are cheaper than the main memory both in terms of access time and energy per access. This enables the Near-Opt. SO approach to over-compensate for the copying overhead and result in significant savings as compared to the SA approach. The Near-Opt. SO approach leads to a maximum reduction of 65% in the memory energy consumption for a 128-bytes SPM. The total energy consumption, being the sum of the processor energy and the memory energy, shows an average reduction of 42%. The application on an average requires 21% less CPU cycles for execution.

Fig. 15 compares the SO approach against the MHLA approach [21] for edge detection benchmark. As discussed in Section II, the MHLA approach overlays array tiles onto the SPM. Due to the unavailability of a freely available DTSE tool, the algorithm of the MHLA approach was manually performed by the authors according to the best of their understanding. The figures also present the energy values for SO-Array approach which is the restricted version of SO approach restricted to overlay only array variables. The readers are requested to refer to Table I in Section III, as the following two paragraphs use the values presented in the table for explaining the above figures.

Fig. 15(a) compares data memory energy values of the overlay approaches. We observe that MHLA approach [cf. Fig. 15(a)] leads to the least data memory energy consumption values for all SPM sizes as it overlays tiles of image arrays present in the benchmark. For 128 bytes, the SO-Array approach assigns small arrays (*viz.* `Gauss`, `x_offset`, `y_offset`) onto the SPM. Thereafter, data memory energy values remain constant as the remaining arrays (*viz.* `in_image`, `gb_image`, `ed_image`, `out_image`, and `tmp_image`) are larger than the SPM size. At 128-bytes SPM, the data memory energy value for the SO approach is higher than that for the SO-Array approach, as the SO approach assigned an instruction segment instead of a data array onto the SPM.

Fig. 15(b) compares the total memory energy, sum of the instruction and data memory energy, of the SO approaches and MHLA approach. The stacked bars of the figure distinguish between instruction and data memory energy values. The first observation that we make is that for zero bytes of SPM the instruction memory energy is much larger than the data memory energy. This is because (cf. Table I) the number of instruction memory accesses (2939787) are an order of magnitude larger

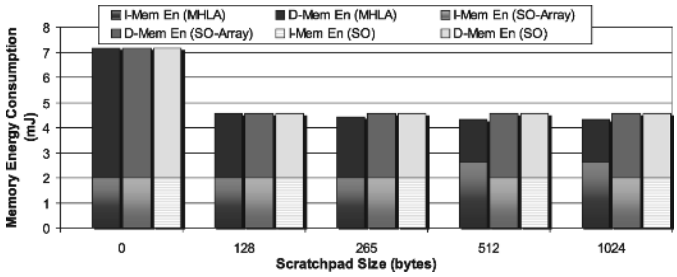

Fig. 16. Edge Detection: Comparison of SO approaches versus MHLA approach for total memory energy (code preassigned to on-chip SRAM).

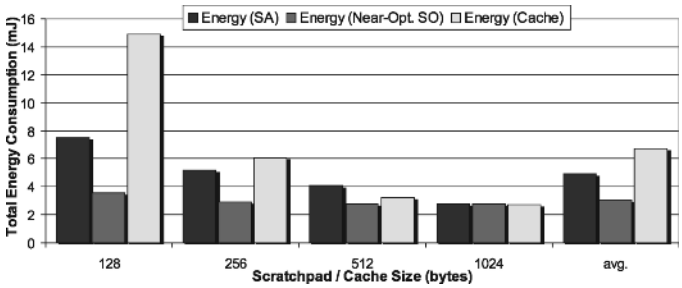

Fig. 17. Edge Detection: Comparison of cache versus SA and Near-Opt. SO approaches.

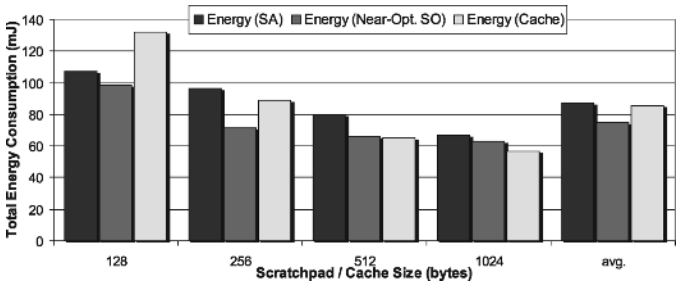

Fig. 18. DSP: Comparison of cache versus SA and Near-Opt. SO approaches.

than data memory accesses (203694). Moreover, in our setup all memory objects reside in the main memory unless they are assigned to the SPM. The SO approach, which assigns both data arrays and instruction segments onto the SPM, therefore, achieves the least memory energy consumption values. For 512 and 1024 bytes, the total memory energy for MHLA approach is larger than those for the SO approaches. The reason being that additional instructions are required for copying array tiles onto the SPM which decreases the data memory energy but increases the instruction memory energy.

In the above comparisons, code segments are more beneficial to overlay onto the SPM due to the fact that ARM7TMDI processor in our setup, is a RISC processor with a von Neumann architecture. However, many ARM processors with separate instruction and data caches are also available. Moreover, many older embedded systems also store their application code in a ROM. Therefore, Fig. 16 presents a comparison for the scratchpad overlay approaches when all code segments are present in an on-chip SRAM memory. This is the best case scenario for data-array optimizations as the instruction memory energy is reduced to the minimum. From Fig. 16, we observe that even in this scenario, the total memory energy for the MHLA approach is not much smaller than that for the SO approaches. Therefore, we believe that for real-life system architectures the energy consumption values, due to the SO approaches, will be a little better than those due to the MHLA approach.

The operation of a scratchpad memory under the control of the scratchpad overlay approaches is similar to that of a cache. Hence, it would be appropriate to present a comparison between a scratchpad memory and a cache. Figs. 17 and 18 compare caches against scratchpads allocated using the SA and the Near-Opt. SO approaches. The cache experiments were conducted for a four-way set associative unified cache of varying sizes. From both the figures, we find that the energy consumption of the system with a 128-bytes cache is much worse than those with a 128-bytes SPM. This is because a small cache causes a high number of misses which results in a high number of expensive main memory accesses. In contrast, the best set of memory objects is assigned to the SPM, reducing the number of main memory accesses. Consequently, the small SPMs irrespective of the allocation approaches, are significantly better than a small cache. The energy consumption for the cache-based systems improves significantly for larger sizes. For 1024 bytes, the energy consumption of the cache-based system is a bit better than that of the SPM-based systems. Though, we should note that for the edge detection benchmark, the energy value for a 1024-bytes SPM allocated using Near-Opt. SO approach is the same as that for a 256-bytes SPM. Hence, we conclude that the energy values for a small SPM-based system with Near-Opt. SO approach is comparable to that for a large cache-based system.

Now, we would evaluate the behavior of the SO approaches for all benchmarks. First, we study the number of instruction and data accesses (cf. Table IV) serviced by the SPM allocated using Near-Opt. SO approach. The size of SPM for each benchmark is different and is chosen to be 20% of aggregate benchmark size, rounded to the nearest two-exponential value. Table IV classifies instruction and data accesses according to the accessed memory for systems with or without a scratchpad memory. It also presents the percentage of instruction and data accesses serviced by the SPM compared against those serviced by the main memory for a system without an SPM. The higher the percentage, the better the utilization of the SPM. From the table, we observe that for all benchmarks the percentage of instruction accesses are higher than that of data accesses serviced by the SPM. This is because, in our setup, instruction accesses are far more beneficial in reducing the energy consumption than data accesses. The percentage of instruction accesses serviced by SPM range between 101.9% and 75.6%. Whereas, data access percentages range between 0.0% and 76.6%. For the multisort benchmark, the Near-Opt. SO approach did not overlay any of the data arrays onto the SPM, as instruction traces provided higher energy reductions. For benchmarks with a high number of large arrays, *viz.* dsp, media and mpeg4, the Near-Opt. SO approach serviced from the SPM a high percentage of 44.5%, 67.0%, and 65.4% data accesses, respectively.

Fig. 19 presents the comparison of the scratchpad overlay approaches against the static allocation approaches across all

TABLE IV
OVERALL MEMORY ACCESSES FOR NEAR-OPT. SO APPROACH

benchmark	SPM size (bytes)	No SPM System		SPM system (Near-Opt. SO)				%access to SPM	
		Inst. Acc.	Data Acc.	Inst. Acc.		Data Acc.			
		MM	MM	MM	SPM	MM	SPM	Inst. Acc.	Data Acc.
adpcm	1024	158926	5264	537	161897	2312	2952	101.9	56.1
edge detection	2048	237484	14093	1007	238411	3298	10795	100.0	76.6
histogram	32k	785046	115456	8816	776796	53248	62208	98.9	53.9
mpeg4	16k	2396856	160444	588340	1811358	55483	104961	75.6	65.4
multisort	512	664038	27230	503	664067	27230	0	100.0	0.0
dsp	16k	2465959	131603	514200	1955487	73005	58598	79.3	44.5
media	16k	3124385	190301	590948	2532806	62823	127478	81.1	67.0

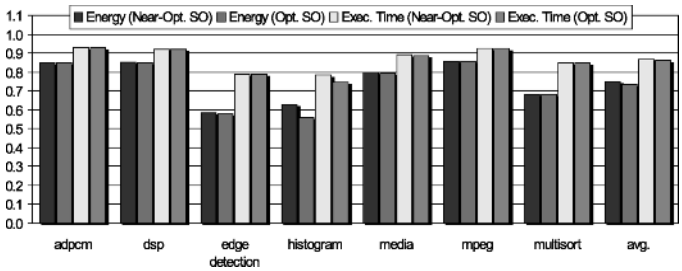

Fig. 19. Overall comparison of scratchpad overlay approaches against the static allocation approach.

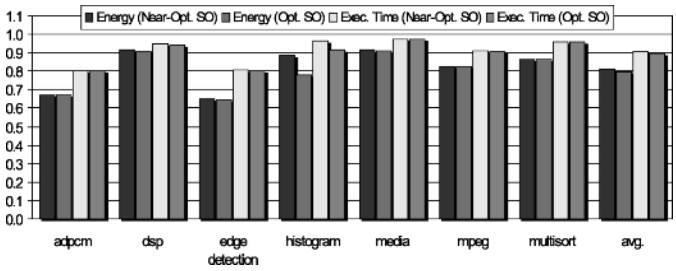

Fig. 20. Overall comparison of scratchpad overlay approaches against the cache.

benchmarks. The total energy consumption and execution time values for the overlay approaches are averaged across all the SPM sizes and are normalized against the corresponding average energy and execution time values for the SA approach. Moreover, the average energy consumption and execution time values across all SPM sizes and across all benchmarks are normalized and are presented as the last set of bars in the figure. The minimum energy and execution time savings are observed for the adpcm benchmark. This is justified as the adpcm benchmark consists of only one encoder and decoder routines and provides little opportunity for overlay. For the histogram benchmark, the maximum average energy savings of 44% are reported due to the the Opt. SO approach. Maximum performance improvements of 22% and 26%, due to the Near-Opt. SO and the Opt. SO approaches, respectively, are reported for the same benchmark. The histogram benchmark comprises of several hotspots and data arrays with nonconflicting life-times. As a consequence, the overlay approaches achieve the maximum savings for the benchmark. For the same benchmark, we observe a noticeable difference between the energy consumption and execution time values for the Near-Opt. SO and the Opt. SO approach. Otherwise, the Near-Opt. SO approach achieved very close to the optimal results. Finally, the overall average energy consumption and execution time savings for the Near-Opt. SO approach are reported to be 25% and 13%, respectively. The overall average energy and execution time savings for the Opt. SO approach are merely 1% better than those for the Near-Opt. SO approach.

Fig. 20 presents a similar comparison of the scratchpad overlay approaches for SPM-based systems against cache-based systems. The SPM allocated using the scratchpad overlay approaches clearly outperform the cache for all benchmarks. However, the total energy and execution time savings due to overlay approaches are lower than those presented in Fig. 19. The maximum energy savings of 35% each due to the optimal and near-optimal overlay approaches, are reported for the edge detection benchmark. The overall average energy savings of 19% and 20% are reported due to the Near-Opt. SO and the Opt. SO approaches, respectively. Moderate average performance improvements of 9% and 10% are reported against a cache-based approach. In the following subsection, we present the evaluation of the scratchpad overlay approaches for the M5 DSP.

B. Experimental Results (M5 DSP)

The experiments for the M5 DSP were conducted by assuming that the L1 group memory or the scratchpad memory of varying sizes can be synthesized. In addition, we assumed that only data elements can be assigned to the scratchpad. Either the SA or the scratchpad overlay approach can be utilized for assigning data elements onto the scratchpad memory. We used the Opt. SO algorithm for the utilization of the scratchpad memory. The Opt. SO approach required minimal computation time as the number of memory objects for the DSP routines were fairly small. We did not consider using the Near-Opt. SO approach due to the low computation time of the Opt. SO approach. Array slicing [30] approach was used to create small slices of the data arrays present in the applications.

A comparison of the SA and the Opt. SO approaches for FIR2DIM, complex multiply, and FIR routines is presented in Fig. 21(a)–(c), respectively. Normalized energy values of the data memory subsystem of the M5 DSP are shown in the figure. The energy consumption value of the data memory system without a scratchpad memory is considered as the unit valued threshold. From the figures, we make a few observations. First,

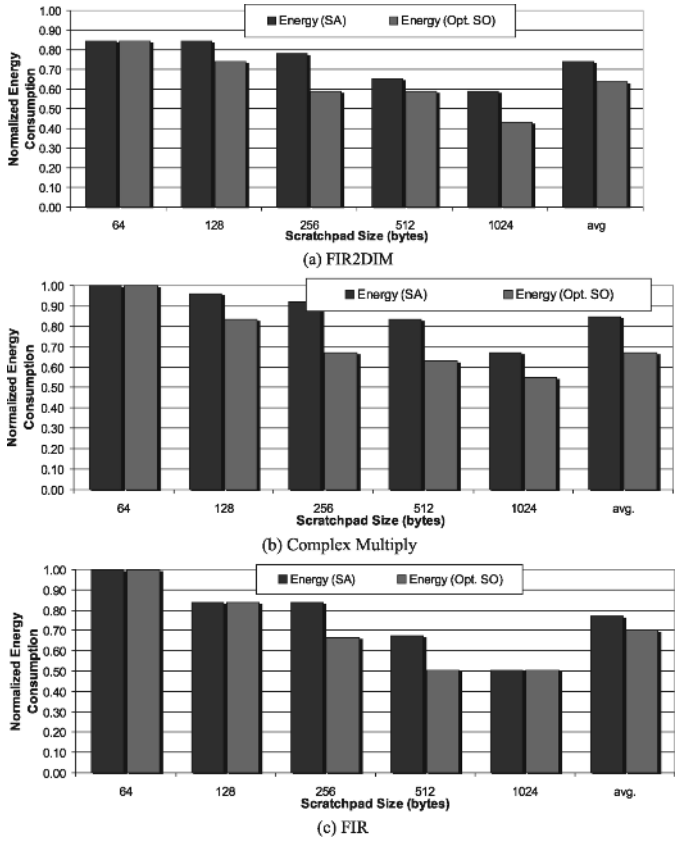

Fig. 21. Energy comparison of scratchpad allocation approaches for M5 DSP.

we observe that the energy values for the Opt. SO approach are always better than or equal to those for the SA approach. Second, we observe that the insertion of a small scratchpad into the data memory hierarchy decreases its energy consumption. Energy reductions in between 40% and 50% can be observed for a system with a 512-bytes scratchpad memory allocated using Opt. SO approach. Third, the Opt. SO approach results in the maximum energy savings of 54%, 44%, and 50% for FIR2DIM, complex multiply, and FIR benchmarks, respectively. Last, The SA approach results in an average energy savings of 28%, 18%, and 23% for FIR2DIM, complex multiply, and FIR benchmarks, respectively. Even higher average energy savings of 31%, 32%, and 30% for FIR2DIM, complex multiply, and FIR benchmarks, respectively, are reported for the Opt. SO approach.

VII. Conclusion

In this work, we presented overlay-based techniques for the utilization of the scratchpad memory. The problem of overlaying both data and instructions onto the SPM was shown to be similar to the problem of global register allocation. A couple of techniques to obtain optimal and near-optimal solutions for the overlay problem were presented. The techniques solve the problem in a two-step process. In the first step, the approaches assign memory objects to the scratchpad memory and also determine optimal locations to insert spill instructions. In the second step, the approaches compute addresses for the memory objects assigned to the scratchpad memory. The optimal approach uses an ILP formulation to determine optimal solutions, whereas the near-optimal approach uses first-fit heuristic-based approach. The proposed approaches result in reduced energy consumption of the system against a published algorithm and against a cache-based system. The average energy reductions of 24% and 20% are reported against the previous approach and the cache-based system, respectively. Additional experiments for a low-power DSP report average savings of 31% in the energy consumption of the data memory hierarchy. In the future, we would like to improve our handling of arrays. Specifically, we would like to overlay both array slices and instructions on to the scratchpad.

References

[1] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," in *Proc. Des. Autom. Conf. (DAC)*, 2002, pp. 628–633.

[2] S. Wuytack, F. Cathoor, L. Nachtergaele, and H. D. Man, "Power exploration for data dominated video applications," in *Proc. Int. Symp. Low-Power Electron. Des. (ISLPED)*, 1996, pp. 359–364.

[3] L. Wehmeyer and P. Marwedel, "Influence of memory hierarchies on predictability for time constrained embedded software," in *Proc. Des. Autom. Test in Europe (DATE)*, 2005, pp. 600–605.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proc. IEEE Int. Symp. Codes. (CODES)*, 2002, pp. 73–78.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1996.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

[7] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 16, no. 3, pp. 428–455, May 1994.

[8] M. D. Smith, N. Ramsey, and G. Holloway, "A generalized algorithm for graph-coloring register allocation," in *Proc. Conf. Program. Lang. Des. Implementation (PLDI)*, 2004, pp. 277–288.

[9] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proc. SIGPLAN Symp. Compiler Construction*, 1982, pp. 98–101.

[10] C. Fu and K. D. Wilken, "A faster optimal register allocator," in *Proc. 31st Int. Microarch. Conf. (MICRO)*, 2002, pp. 245–256.

[11] D. W. Goodwin and K. D. Wilken, "Optimal and near-optimal global register allocation using 0–1 integer programming," *Software-Practice and Experience*, vol. 26, no. 8, pp. 929–965, Aug. 1996.

[12] O. Coudert, "Exact coloring of real-life graphs is easy," in *Proc. Des. Autom. Conf. (DAC)*, 1997, pp. 121–126.

[13] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 3rd ed. Boston, MA: Addison-Wesley, 1973, vol. 2.

[14] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?," in *Proc. 1st Int. Symp. Memory Managem. (ISMM)*, 1998, pp. 26–36.

[15] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," in *Proc. Workshop Compiler Arch. Support Embedded Comput. Syst. (CASES)*, 2003, pp. 318–326.

[16] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embedded Comput. (TECS)*, vol. 1, no. 1, pp. 6–26, Nov. 2002.

[17] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-On-Chip*. Norwell, MA: Kluwer, 1999.

[18] S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. Des. Autom. Test Eur. (DATE)*, 2002, pp. 409–415.

[19] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-aware scratchpad allocation algorihm," in *Proc. Des. Autom. Test Eur. (DATE)*, 2004, pp. 1264–1269.

[20] F. Angiolini, M. Francesco, F. Alberto, L. Benini, and M. Olivieri, "A post-compiler approach to scratchpad mapping of code," in *Proc. Workshop Compiler Arch. Support Embedded Comput. Syst. (CASES)*, 2004, pp. 259–267.

[21] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Cathoor, "Layer assignment techniques for low energy in multi-layered memory organizations," in *Proc. Des. Autom. Test Eur. (DATE)*, 2003, pp. 1070–1075.

[22] O. Ozturk, M. Kandemir, I. Demikiran, G. Chen, and M. Irwin, "Data compression for improving SPM behavior," in *Proc. Des. Autom. Conf. (DAC)*, 2004, pp. 401–406.
[23] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown, "Compiler managed dynamic instruction placement in a low-power code cache," in *Proc. Int. Symp. Code Generation Optimization (CGO)*, 2005, pp. 179–190.
[24] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, "Reducing energy consumption by dynamic copying of instructions onto onchip memory," in *Proc. 15th Int. Symp. Syst. Syn. (ISSS)*, 2002, pp. 213–218.
[25] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," in *Proc. Conf. Hardw./Softw. Codes. Syst. Syn. (CODES+ISSS)*, 2004, pp. 104–109.
[26] A. Appel and L. George, "Optimal spilling for CISC machines with few registers," in *Proc. Conf. Program. Lang. Des. Implementation (PLDI)*, 2001, pp. 243–253.
[27] S. Steinke, L. Wehmeyer, M. Verma, and P. Marwedel, "Energy Aware C Compiler," Dept. Comput. Sci. XII, Univ. Dortmund, Dortmund, Germany, 2006. [Online]. Available: http://ls12-www.cs.uni-dortmund.de/research/encc/
[28] S. Wuytack, J.-P. Diguet, and F. Catthoor, "Formalized methodology for data reuse exploration for low power hierarchical memory mappings," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 6, no. 6, pp. 529–537, Dec. 1998.
[29] H. Tomiyama and H. Yasuura, "Optimal code placement of embedded software for instruction caches," in *Proc. 9th Eur. Des. Test Conf. (ED&TC)*, 1996, pp. 96–101.
[30] S. S. Muchnick, *Advanced Compiler Design and Implementation*, 1st ed. San Mateo, CA: Morgan Kaufmann, 1997.
[31] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," presented at the Int. Workshop Power Timing Modeling, Optimiz. Simul. (PATMOS) Yverdon-Les-Bains, Switzerland, 2001.
[32] ILOG Inc., Gentilly, France, 2006. [Online]. Available: http://www.ilog.com/products/cplex/
[33] Atmel corporation, San Jose, CA, 2006. [Online]. Available: http://www.atmel.com/
[34] Advanced RISC machines ltd., Cambridge, U.K., 2006. [Online]. Available: http://www.arm.com/
[35] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Synchronous transfer architecture (STA)," in *Proc. Int. Workshop Syst. Arch. Modeling Simul. (SAMOS)*, 2004, pp. 343–352.
[36] *"Benchmark Suite for Multimedia and Communication Systems,"* Cares Lab., Univ. California, Los Angeles, CA, , , 2006 [Online]. Available: http://cares.icsl.ucla.edu/MediaBench/
[37] C. Lee, DSP Benchmark Suite Univ. Toronto, Toronto, ON, Canada, 2005 [Online]. Available: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html
[38] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," presented at the Signal Process. Appl. Technol. (SPAT) Dallas, TX, 1994.

Manish Verma (S'06) recieved the B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi, India, in 2001. He is currently pursuing the Ph.D. degree in the Embedded Systems Research Group at the University of Dortmund, Dortmund, Germany.

His research focus is on optimizing compilers and memory hierarchy optimization of uni- and multiprocessor embedded systems.

Peter Marwedel (SM'05) received the Ph.D. in physics and the Dr.Habil. in computer science from the University of Kiel, Kiel, Germany, in 1974 and 1987.

He is currently a Professor at the University of Dortmund, Dortmund, Germany, since 1989, and he is also the head of the local technology transfer centre ICD. His research interests include high-level synthesis and embedded systems, in particular code generation for embedded systems. His early work includes the design of the MIMOLA high-level synthesis system. Currently, his focus is on generation techniques for efficient embedded code.

Dr. Marwedel is a member of the Association for Computing Machinery and a member of the Gesellschaft für Informatik. He is also a member of the ARTIST European network of excellence on embedded and real-time systems.