Dynamic Overlay of Scratchpad Memory for Energy Minimization

Manish Verma, Lars Wehmeyer, Peter Marwedel
Department of Computer Science XII
University of Dortmund
44225 Dortmund, Germany
{Manish.Verma, Lars.Wehmeyer, Peter.Marwedel}@uni-dortmund.de

ABSTRACT

The memory subsystem accounts for a significant portion of the aggregate energy budget of contemporary embedded systems. Moreover, there exists a large potential for optimizing the energy consumption of the memory subsystem. Consequently, novel memories as well as novel algorithms for their efficient utilization are being designed. Scratchpads are known to perform better than 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 an allocation technique which analyzes the application and inserts instructions to dynamically copy both code segments and variables onto the scratchpad at runtime. We demonstrate that the problem of dynamically overlaying scratchpad is an extension of the Global Register Allocation problem. The overlay problem is solved optimally using ILP formulation techniques. Our approach improves upon the only previously known allocation technique for statically allocating both variables and code segments onto the scratchpad. Experiments report an average reduction of 34% and 18% in the energy consumption and the runtime of the applications, respectively. A minimal increase in code size is also reported.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – Compilers, Memory Management

General Terms: Design Measurement Performance.

Keywords: scratchpad, overlay, dynamic allocation.

1. INTRODUCTION

The design of embedded systems is very much driven by applications. It is expected that future applications will be derived from the multimedia domain and will require significantly more processing power. As a result, powerful processors and large memories will have to be used in future embedded systems. It is evident that these processors and memories are exteremely power-hungry, whereas the electrical energy available in embedded systems (especially in portable systems) is strictly limited. Consequently, a significant research on low power techniques is being performed.

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

Several researchers [10, 14] have identified the memory subsystem as the energy bottleneck of the entire system. Memory hierarchies are being constructed to reduce the memory subsystem's energy dissipation. Caches and scratchpad memories represent two contrasting memory architectures. Caches improve the performance by exploiting the temporal and spatial locality present in the program. However, for embedded systems, the overheads associated with caches often negate their benefits. Moreover, caches are notorious for their unpredictable behavior [9].

On the other hand, a scratchpad memory (SPM) consists of just a memory array and address decoding circuitry. Due to the absence of the tag memory and the comparators, scratchpad memories require much less energy per access than a cache. In addition, they require less onchip area and allow tighter bounds on WCET prediction of the system. However, they require complex program analysis and explicit support from the compiler.

Previous work on SPM usage has mainly focused on static allocation of the SPM. Unlike a cache which adapts its contents according to the program behavior, a statically allocated SPM does not modify its contents during the runtime of the program. The static allocation technique may lead to under utilization of the SPM. Consequently, we present a profile based approach which on the basis of live ranges of both variables and code segments, replenishes the contents of the SPM. The technique identifies the points in the program to insert instructions to copy the contents on and off the SPM. The points are optimally chosen in order to cause the least overhead. The technique also computes addresses within the SPM address range where variables and code segments are to be copied. These addresses are computed such that a large number of variables and code segments share the same SPM space.

The rest of the paper is structured as follows: After the presentation of related work, the algorithm for solving the scratchpad overlay problem is described in Section 3. This is followed by a presentation of the experimental workflow. A discussion on the experimental results is presented in Section 5. The paper ends with a conclusion and future work.

2. RELATED WORK

The research on scratchpad utilization can be classified into two broad categories *viz.* static allocation and dynamic allocation techniques. In the former, the SPM is loaded once at the start and its contents remain invariant during the entire execution of the application. On the other hand, dynamic allocation techniques change the SPM contents during the execution to reflect the dynamic behavior of the application.

Most of the research [3, 14] on static allocation of the SPM has focused on assigning data elements. The authors [14] were

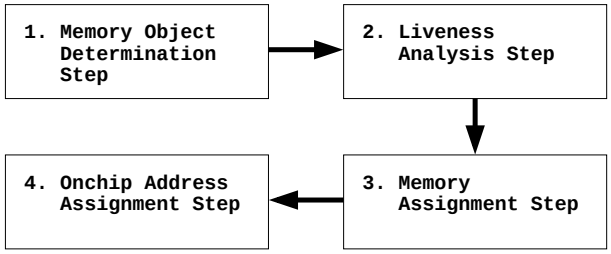

Figure 1: Workflow of the Scratchpad Overlay Algorithms

the first-ones to demonstrate the effectiveness of the SPMs. They achieved commendable reductions in the runtime of the application by mapping frequently accessed data elements onto the SPM. Authors in [3] proposed an optimal algorithm for assigning data elements onto the SPM. The algorithm is based upon profiling the application and solving a system of binary linear equations. In both the approaches, data elements (e.g. arrays) that are too large to fit in the SPM are kept in the main memory.

A recent approach [20] utilizes the SPM to store instructions and models an instruction cache present in the system as a conflict graph. Instructions are mapped to the SPM in order to minimize the conflict cache misses and the energy consumption. Authors in [17, 19] demonstrated the benefits that can be achieved by assigning both instructions and data elements to the SPM.

In contrast, dynamic utilization of the SPM is still a field of ongoing research. Authors of [10] use Presburger formulas to dynamically copy parts of arrays at runtime. Dynamic copying of instructions is proposed by authors in [15]. Certain infrequently executed points (*e.g.* entry point of a function or a loop) are considered as copy-points. At these copy-points, copying of the following basic blocks is considered in order to minimize the energy consumption.

Global register allocation is one of the most researched and fundamental topics in code optimization and compiler construction [9]. 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*. Those symbolic registers which cannot be assigned a real register are kept in the main memory. *Spill code* is inserted to bring these symbolic registers to and from the main memory. *Global register allocation* attempts to find an assignment of the symbolic registers to the processor's real registers such that the overhead due to the generated spill code is minimized. The allocation problem was proven to be NP-complete [7]. Consequently, most of the allocators use a graph coloring [5] based heuristic. These allocators perform near-optimal allocation for regular architectures with a large number of processor registers. However, they fail considerably for irregular architectures which invariably come with a smaller number of registers. In the recent past, approaches for optimal register allocation [1, 8, 11] have been proposed. The register allocation problem is formulated as an integer linear program (ILP) and solved using a general purpose ILP solver [6]. Although global register allocation is NP-complete, authors [8] have empirically demonstrated that it takes $O(n^3)$ time to compute the optimal register allocation for real-life benchmarks. The runtime was reduced to $O(n^{1.3})$ when the problem was relaxed to compute near-optimal solutions [1].

3. SCRATCHPAD OVERLAY (SO)

The problem of scratchpad overlay attempts to share the SPM between memory objects such that the memory objects are copied onto the SPM when required and copied off when not required. The scratchpad overlay problem is a weighted version of the *global register allocation* problem for CISC architectures, as memory objects can have different sizes. Unlike RISCs, CISCs are relaxed to have one or more operand(s) of an instruction as a memory location. This reduces the pressure on the register file and the number of unnecessary load and store instructions. However, the high cost of accessing memory motivates toward an optimal utilization of the register file. The global register allocation problem for CISCs is also NP-complete [1] and as a result the scratchpad overlay problem can be proven to be NP-complete. Briefly, the proof argues that if the scratchpad overlay problem is restricted to have equal sized memory objects then the problem transforms to the global register allocation problem. The NP-completeness of the allocation problem ensures that 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, as shown below both the problems are NP-complete. Nevertheless, we present optimal approaches for both the first and the second problem.

The scratchpad overlay problem is solved using the four-step approach shown in figure 1. In the first step, variables and code segments from the application code are identified as memory objects for scratchpad overlay. Liveness analysis is performed in the following step to determine the live range of these memory objects. The third step involves the optimal assignment of memory objects to the SPM or to the main memory. In the final step, onchip addresses of the memory objects assigned to the SPM are computed.

3.1 Memory Objects

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

- Global variables (both scalar and non-scalar)
- Non-scalar local variables
- Code segments called *traces*.

All global variables are considered as they occupy a space in the data memory. Only non-scalar local variables are considered as they consume a space on the stack and are generally not assigned to the register file. We expect that the frequently accessed scalar local variables will be assigned to the registers during the register allocation step. Therefore, they are not considered for allocation onto the SPM. Frequently executed code segments called *traces* are identified using the *Trace Generation* technique. A trace is a frequently executed straight-line path consisting of basic blocks connected by fall-through edges. Our traces are similar to the ones described in [20]. Traces improve the processor performance by enhancing the spatial locality present in the program code. Moreover, due to the fact that traces always end with an unconditional jump [18], they form an atomic unit of instructions which can be placed anywhere in the memory without modifying other traces. The above set of candidates is termed *Memory Objects* (MO). In the following subsection, we compute the live range of each memory object.

3.2 Liveness Analysis

Liveness analysis is performed on the control flow graph (CFG) $G(N,E)$ of every function. The node set N of the CFG is the set of basic blocks present within the function code and the edges belonging to the edge set E represent the possible flow of control during the execution of the function. The concept of *DEF-USE* chains [13]

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 classified as a **USE**.

The nodes of the CFG are attributed with *DEF-MOD-USE* information in order to compute the live range of memory objects. A combination of both static and profiling based analysis method is used to determine basic blocks containing references to variables and also the type of those references. The static method is used to determine basic blocks containing references while dynamic profiling is used to differentiate between *DEF* and *MOD* references. Traces, inspite of being a set of basic blocks, are considered similar to variables. However, they cannot have a *DEF* or a *MOD* reference, as instructions are never defined or modified but are always executed. Consequently, all basic blocks are assigned a *USE* reference to the corresponding trace. A fixed point iterative algorithm is then used to compute the live range of every memory object.

3.3 Memory Assignment Problem

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 CFG. The edge based formulation enables the efficient determination of the optimal points for the spill code insertion. We define the following static attributes ($Attrib_{STATIC}$) for every memory object on each edge of the CFG.

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

where the *DEF* attribute is defined on every edge originating from a node with a *DEF* attribute. In contrast, the *MOD* or *USE* attribute is 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 when an edge can be assigned more than one static attributes for a memory object, the following priority order is used to determine the appropriate attribute.

$$DEF > MOD > USE > CONT$$

The priority order defined above guarantees that no DEF-USE chains are broken. Additionally, spill attributes ($Attrib_{SPILL}$) are defined on edges to model appropriate spilling of memory objects.

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

The *LOAD* attribute defined on an edge implies that the corresponding memory object can be spill-loaded from the main memory onto the SPM on the current edge. On the other hand, the *STORE* attribute implies that the memory object can be spill stored from the SPM onto the main memory. The *LOAD* attribute is defined on edges which have the *MOD*, *USE* or *CONT* attribute defined or which originate from a *diverge node*. Similarly, *STORE* attribute is defined on edges which have *DEF* attribute defined or which enter a *merge node*. A diverge (merge) node is a node whose out-degree (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.

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

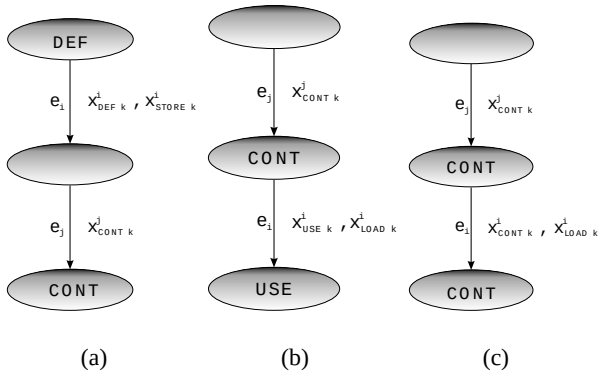

Figure 2: Flow Constraints: (a) DEF, (b) USE and (c) CONT constraint

where $e_i \in E$, $at_j \in Attrib_{STATIC} \cup Attrib_{SPILL}$ and $mo_k \in MO$. We first describe the objective function and then the constraints for the proposed ILP formulation. The objective function represents the energy savings that could be achieved by scratchpad overlay. The energy savings (objective function) need to be maximized in order to minimize the energy consumption of the system.

$$E = \sum_i \sum_k \left\{ \begin{array}{lcl} & E_{profit}(i,j,mo_k) & * \; x^i_{j\,k} \\ - & E_{load\_cost}(i,mo_k) & * \; x^i_{LOAD\,k} \\ - & E_{store\_cost}(i,mo_k) & * \; x^i_{STORE\,k} \end{array} \right\} \quad (2)$$

where $E_{profit}(i,j,mo_k)$ is the energy savings obtained by assigning memory object mo_k to the SPM at edge e_i. $E_{load\_cost}(i,mo_k)$ and $E_{store\_cost}(i,mo_k)$ 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 refrain from explaining the computation of E_{profit}, E_{load_cost} and E_{store_cost}, computed using an accurate energy model [16].

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

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

Figure 2(a) represents the DEF-constraint, where edge e_i 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, a *MOD-constraint* or *USE-constraint* is added for edges with *MOD* or *USE* attribute.

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

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

As shown in the figure 2(b), 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

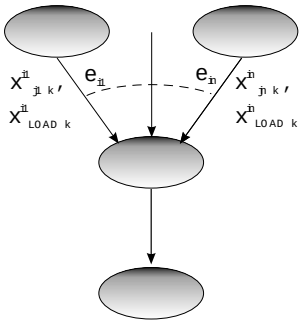

Figure 3: Merge-Node Const.

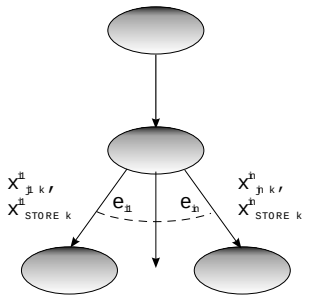

Figure 4: Diverge-Node Const.

flow constraint, represented in figure 2(c), is added for edges with *CONT* attribute.

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

The CONT-constraint (refer figure 2(c)) informally implies 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 (eqn. 7, 8 and 9, 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 [8]. The following *merge-node constraints* are added for all the merge nodes.

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

$$x^{i1}_{j1\,k} = \ldots = x^{in}_{jn\,k} \quad s.t. \;\; at_{j1} \ldots at_{jn} \in Attrib_{STATIC} \qquad (8)$$

In the above constraints, edges $e_{i1} \ldots e_{in}$ (refer figure 3) constitute all the edges entering a merge node. The merge node constraint 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 or spill loaded (*LOAD*) on each of the remaining edges. Similarly, for all the diverge nodes the following constraints (*diverge-node constraint*) are added.

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

$$x^{i1}_{j1\,k} = \ldots = x^{in}_{jn\,k} \quad s.t. \;\; at_{j1} \ldots at_{jn} \in Attrib_{STATIC} \qquad (10)$$

As shown in figure 4, edges $e_{i1} \ldots e_{in}$ represent 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* for all edges which ensures that the aggregate size of all memory objects assigned to the SPM on each edge should be less than the SPM size.

$$\sum_k x^i_{j\,k} * Size(mo_k) \leq ScratchpadSize \;\; \forall e_i \in E \qquad (11)$$

A commercial ILP solver [6] 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 maximum of two binary variables, each corresponding to the static and the spill attributes. Consequently, the total number of variables in the ILP formulation is $O(|MO| * |E|)$. However, the maximum runtime of ILP solver was found be less than 16.99 CPU seconds on a Sun Sparc 1300Mhz compute machine. We still have to determine the addresses of the memory objects assigned to the SPM, in order to solve the scratchpad overlay problem. An approach to compute the addresses is presented in the following subsection.

3.4 Onchip Address Assignment Problem

In the previous step, an implicit assumption was made while formulating the memory assignment problem as an ILP. 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 onchip 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 onchip addresses, despite the scratchpad size constraint being satisfied. The problem of onchip 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 [7]. 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:

$$0 \leq \quad O^i_j \quad \leq ScratchpadSize - Size(mo_j) \qquad (12)$$

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.

$$O^i_j - O^i_k \quad \geq \quad Size(mo_k) \quad \textbf{XOR} \qquad (13)$$

$$O^i_k - O^i_j \quad \geq \quad Size(mo_j) \qquad (14)$$

The first constraint (eqn. 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 ($O^i_k + Size(mo_k)$) of memory object mo_k. The second constraint (eqn. 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.

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

The following is the linearized form of the above set of constraints with **L** being a sufficiently large constant.

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

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

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.

$$O^i_k - O^j_k \quad = \quad 0 \qquad (18)$$

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.

$$v^{ij}_k = \begin{cases} 1 & \text{if } O^i_k \neq O^j_k \\ 0 & \text{otherwise} \end{cases} \qquad (19)$$

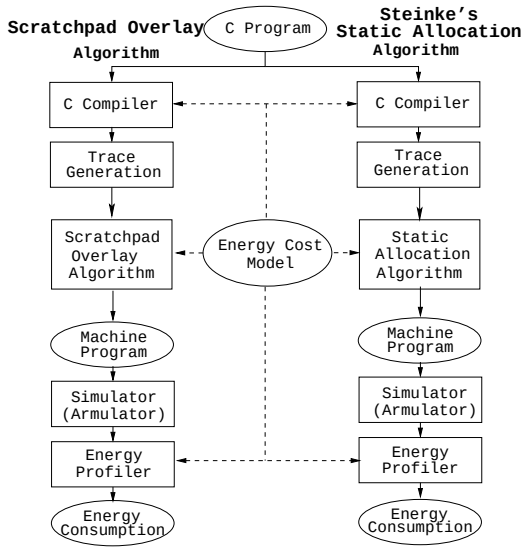

Figure 5: Experimental Workflow

The unit value of the variable v_k^{ij} 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_k^{ij}.

$$O_k^i - O_k^j - \mathbf{L} * v_k^{ij} \quad = \quad 0 \forall e_i, e_j \in E \tag{20}$$

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_k^{ij} for all valid pairs of edges and for all memory objects is denoted as the objective function of the ILP formulation.

$$\sum_i \sum_j \sum_k v_k^{ij} \tag{21}$$

For a valid solution the value of the objective function should be zero which is achieved by minimizing the objective function. The ILP formulation is a Mixed Integer Linear Programming (MIP) problem, as it consists of both binary and integer variables. The number of integer variables 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 [6], which can take substantial time for certain problem instances having a large number of variables.

4. WORKFLOW

The experimental setup consists of an ARM7T processor core, an onchip SPM and an offchip main memory. We compared the energy consumption of the system when the onchip SPM is allocated using the scratchpad overlay technique against the static allocation technique by Steinke et. al [17]. The energy consumption of the system is based upon accurate energy models proposed by [4, 16], where [16] has an accuracy of 98% for our experimental setup.

The experiments were conducted according to the workflow presented in figure 5. In the first step, the benchmarks programs are compiled using an energy optimizing C compiler. The I-cache optimization technique called trace generation [18] is applied in the following step. This is followed by the application of the proposed scratchpad overlay technique. The generated machine code

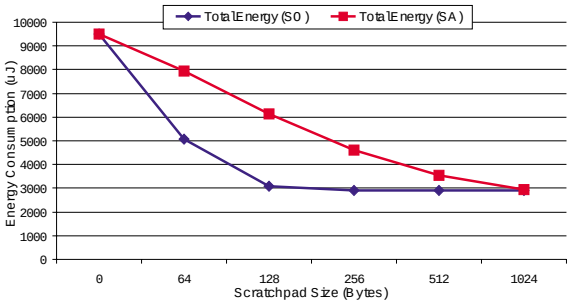

Figure 6: Energy Consumption of Edge Detection using SO vs. SA

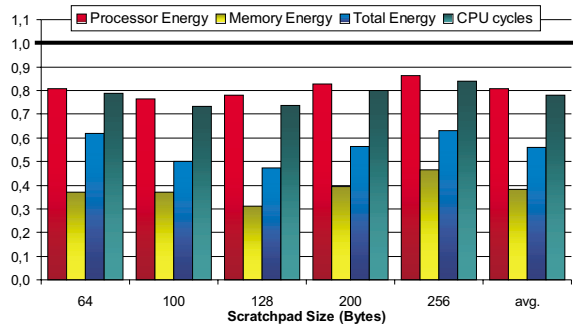

Figure 7: Edge Detection: Energy and Performance comparison of SO vs. SA algorithm

is then fed into the ARMulator [2] to obtain a sequence of executed instructions. Finally, the energy consumption of the system is computed using the instruction sequence and the energy models [4, 16]. A similar workflow is followed to compute the energy consumption when the SPM is allocated using the static allocation (SA) approach [17]. In the following section, we discuss the results obtained using the scratchpad overlay approach.

5. EXPERIMENTAL RESULTS

The proposed technique is evaluated for an assorted set of benchmarks from MediabenchII [12] and UTDSP benchmark suite. Moreover, for a fair comparison, a benchmark consisting of the sorting routines presented in [17] is also included. Energy consumption was computed by varying the size of the SPM, allocated with the static approach [17] or with the scratchpad overlay (SO) approach.

Figure 6 compares the energy consumption of the edge detection benchmark when the SPM is allocated using the proposed overlay technique against the static allocation (SA) technique [17]. The energy curve for SA algorithm monotonically decreases with the increase in the SPM size, as the technique can allocate additional memory objects for larger sizes. In contrast, the energy curve for the SO algorithm declines faster to reach a threshold value at 256B of the SPM and thereafter remains constant for larger sizes. The justification for the behavior is that the SO algorithm is able to share the SPM among many memory objects. The energy consumption becomes constant when no additional memory objects can be overlayed on the SPM. Moreover, the energy consumption of a system with 256B SPM allocated with SO algorithm is equal to that of a system with 1024B SPM allocated with SA algorithm. This implies that by sharing a small SPM, equal or higher energy savings can be obtained than by statically using a large SPM. The

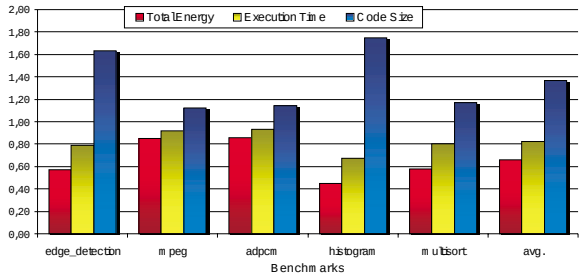

Figure 8: Energy, Performance and Code Size comparison using SO vs. SA algorithm

trend of energy consumption values remains the same for all the benchmarks. Consequently, a comparison of the two allocation algorithms is presented for a smaller and finer range of SPM sizes.

Figure 7 denotes the normalized energy consumption and CPU cycles (execution time) values of the SO algorithm. The energy and performance values of the SA algorithm are denoted as the unit valued baseline. The efficient utilization of the SPM by the SO algorithm leads to reductions of upto 65% in memory energy consumption. Both the processor energy and the execution time are reduced as accessing the onchip SPM requires less CPU cycles than the offchip main memory. The total energy consumption, being the sum of the processor energy and the memory energy, shows an average reduction of 43%. The application on average requires 21% less CPU cycles for execution.

The comparison of SO and SA algorithms across all benchmarks are presented in figure 8. The normalized energy values, the execution time values and the code sizes are the averages over all scratchpad sizes. The SO algorithm achieves reductions of 15%, 14%, 55% and 42% in the total energy consumption for mpeg, adpcm, histogram and multisort respectively. Average reductions of 8%, 7%, 32% and 20% in the execution time of the applications are also reported. The average reductions in energy consumption and execution time across all benchmarks are 34% and 18%, respectively. The SO approach inserts spill code for loading and storing of memory objects which is reflected by the increase in the code size of the application. The average increase in code size of varies from 12% to 75% for our set of benchmarks. However, the increase in code size becomes negligible (less than 1%) when the size of the whole application consisting of both data and code is considered.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a liveness analysis based technique for dynamic utilization of the scratchpad memory. The problem of dynamically overlaying both data and instructions onto the SPM was shown to be an NP-complete problem. A technique which breaks the scratchpad overlay problem into two smaller problems and obtains an optimal solution for both the problems was presented. The presented technique enables efficient utilization of the SPM and results in reduced energy consumption of the system against a published algorithm. The average reductions in energy consumption and execution time are 34% and 18%, respectively. In the future, we intend to extend the approach to handle inter-procedural overlay of the SPM and would also like to explore the possibility of obtaining near-optimal results using heuristic approaches.

7. REFERENCES

[1] A. W. Appel and L. George. Optimal spilling for cisc machines with few registers. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, pages 243–253, Snowbird, Utah, USA, 2001.

[2] ARM. *Advanced RISC Machines Ltd.* http://www.arm.com.

[3] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

[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. of 10th International Symposium on Hardware/Software Codesign*, Colorado, USA, May 2002.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–101, Boston, Massachusetts, USA, 1982.

[6] CPLEX. *CPLEX limited*. http://www.cplex.com.

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

[8] D. W. Goodwin and K. D. Wilken. Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. *Software-Practice and Experience*, 26(8):929–965, August 1996.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach; second edition*. Morgan Kaufmann, 1996.

[10] M. Kandemir, I. Kadayif, and U. Sezer. Exploiting Scratch-Pad Memory Using Presburger Formulas. In *Proc. of the 14th International Symposium on System Synthesis (ISSS)*, pages 7–12, Montreal, P.Q., Canada, 2001.

[11] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *Proc. of the 31st annual ACM/IEEE International Symposium on Microarchitecture*, Dallas, TX, USA, 1998.

[12] MediabenchII. *Benchmark Suite for Multimedia and Communication Systems*. http://cares.icsl.ucla.edu/MediaBenchII/.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1. edition, 1997.

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

[15] 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. of the 15th International Symposium on System Synthesis (ISSS)*, Kyoto Japan, October 2002.

[16] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proc. of International Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, Yverdon-Les-Bains, Switzerland, Sep. 2001.

[17] S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. of Design Automation and Test in Europe (DATE)*, Paris France, March 2002.

[18] H. Tomiyama and H. Yasuura. Optimal code placement of embedded software for instruction caches. In *Proc. of the 9th European Design and Test Conference*, Paris France, March 1996. ET&TC.

[19] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*, page 77, January 2003.

[20] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware Scratchpad Allocation Algorihm. In *Proc. of Design, Automation and Test in Europe (DATE)*, February 2004.