Data Partitioning for Maximal Scratchpad Usage

Manish Verma
Computer Science XII
University of Dortmund
44221 Dortmund, Germany
Tel:+49-231-755-6325
Fax:+49-231-755-6116
verma@ls12.cs.uni-dortmund.de

Stefan Steinke
Computer Science XII
University of Dortmund
44221 Dortmund, Germany
Tel:+49-231-755-6133
Fax:+49-231-755-6116
steinke@ls12.cs.uni-dortmund.de

Peter Marwedel
Computer Science XII
University of Dortmund
44221 Dortmund, Germany
Tel:+49-231-755-6111
Fax:+49-231-755-6116
marwedel@ls12.cs.uni-dortmund.de

Abstract— The energy consumption for Mobile Embedded Systems is a limiting factor because of today's battery capacities. The memory subsystem consumes a large chunk of the energy, necessitating its efficient utilization. Energy efficient scratchpads are thus becoming common, though unlike caches they require to be explicitly utilized. In this paper, an algorithm integrated into a compiler is presented which analyzes the application, partitions an array variable whenever its beneficial, appropriately modifies the application and selects the best set of variables and program parts to be placed onto the scratchpad. Results show an energy improvement between 5.7% and 17.6% for a variety of applications against a previously known algorithm.

I. Introduction

The past decade has seen a considerable amount of growth in Embedded Systems. Computers are no longer confined to offices and home desks, they are finding use in every conceivable area. A large number of Embedded Systems are being modeled as mobile devices, for which dimension, weight and energy consumption are distinguishing features.

Improvements concerning the *battery capacity* have been made, but the rate is negligible compared to the rate at which energy consumption has increased. For example, the battery life of Ni-Cd batteries has increased by a factor of 2 in the last 30 years [19]. However, the power dissipation in the interior of a full featured notebook has increased by 90% over the last 3 years [6]. Moreover, the size of the laptops cannot be further reduced because of the surface area required for energy dissipation [6]. Hence, system architects have the utmost task of reducing energy consumption. There are a large number of options for reducing the energy/power consumption, a few examples of which are the following:

First and foremost, would be to decrease the feature size, as it directly effects the energy consumption. However, the lower the feature size, the higher is the cost and the complexity of the system. Another option is to decrease the switching activity of the transistor, as it is responsible for 70%-90% of the total transistor energy consumption [17]. The energy consumption depends on the number of switching operations and the load capacity of the attached nets. Nowadays, advanced micro-controllers come with power saving or idle modes and also with lower frequency modes. Thus the micro-controller can be made to sleep when idle or work at sufficiently lower frequency if meeting all constraints.

A different energy optimizing approach would be a prudent selection of system components by creating a memory hierarchy, using onchip and offchip memories. The onchip memory consumes less energy because of its small size and the proximity to the core, access times are low and less bus drivers need to be activated. Memory organization consisting of caches or scratchpads [1] or both can be used for an optimal energy reduction.

Caches provide copying of *relevant/irrelevant* data to/from the offchip memory respectively. If the memory content is valid then the current access to the offchip memory is saved. The software code can be used without any modification for different cache memory organizations. However, caches are notorious for their unpredictable behavior. An application for a certain cache configuration may consume equal or more energy because of *thrashing* [5], than the one without cache. In real-time embedded systems, a number of time constraints have to be fullfilled and a certain response time has to be guaranteed. Worst Case Execution Time (WCET) behavior of cache-based systems is very difficult to predict exactly. A simple and imprecise upper bound on WCET can be calculated assuming a cache miss for every memory access. Tight upper bounds [9] can be computed, but not for all applications. Consequently, the presence of cache does not greatly reduce the WCET. On the other hand, scratchpad memories consume much less energy per access, as they are devoid of tag arrays and comparators [3]. Scratchpads do not cause a miss, hence can be used to improve the WCET. However, they depend upon software for their utilization.

In this work, we propose a compiler extension which uses the scratchpad to the maximal value, partitioning program and data into smaller segments whenever beneficial. The best set

[1] Small onchip RAM's statically mapped into the processor's address space are called Scratchpads.

of program and data values are identified using Integer Linear Programming [10] and the selected objects are placed onto the scratchpad.

In the next section we describe related research work followed by a detailed description of the algorithm. In section IV we describe the experimental setup, followed by the results of the experiments performed on the ARM7T RISC processor. The paper ends with a conclusion and future work.

II. Related Work

Optimizing the energy consumed by the application program has been a research topic since the last decade. Tiwari et al. [18, 19] were among the first to propose an energy model. This instruction level model consists of a constant energy cost for each instruction ("base cost") plus an overhead cost ("inter-instruction cost") for switching to the next instruction. The sum over all instructions denotes the total amount of energy consumed by the observed application. This model is useful for compilers which could generate the most energy efficient code. However, the model fails to take system components into consideration. For low power processors, energy consumption due to memory accesses can not be neglected. Otherwise, the generated code would be optimal for processor energy but not for the whole system including memory.

The instruction level model by Simunic et al. [14] incorporates the effect of memories, but since all values are taken from the vendor's datasheet, there is no distinction between different instructions. The combination of these two approaches and a further extension for treating the bit patterns on buses was presented by Steinke et al. [16]. This model, having a precision of 1.7%, is chosen for the current research work.

Panda et al. [11, 12] presented an efficient use of the memory hierarchy by placing the most commonly used variables onto the scratchpad. A further approach by Sjödin et al. [15] places variables onto scratchpad by static analysis and shows that this is sufficiently precise and no dynamic analysis is needed. The dynamic copying of array parts was studied by Kandemir et al. [7]. However, the algorithm is applicable under the simplifying constraints [21] [7] (*i.e.* perfectly nested loops, exactly known loop bounds and array subscripts being affine function of the all loop indices along with additional constraints). Also, it applies only when the aggregate overhead cost of moving data to and from onchip memory is less than the cost of accessing the data accessing data from offchip memory. Consequently, the applicability of the algorithm is restricted to data-intensive DSP applications.

Steinke et al. [8] moved both data and program parts onto the scratchpad, based on the static analysis of execution of instructions and accesses of variables. The approach shows an improvement of 12%-43% in energy consumption against caches. However, it only considers whole variables at a particular time. Thus, a large non-scalar variable (e.g. array variable) could either be put on to the onchip as a whole or not. This could potentially lead to a sub-optimal utilization of the scratchpad.

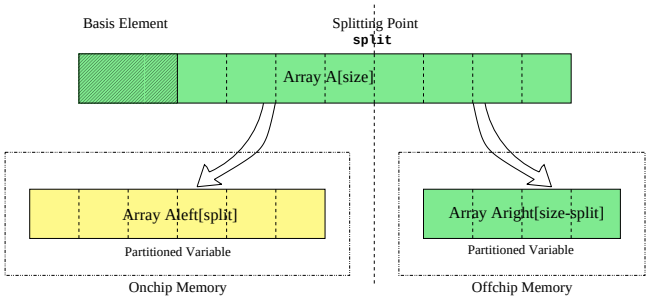

Fig. 1. Partitioning of Array

Based on the cache model CACTI developed by Wilton et al. [20], Banakar et al. [3] presented a detailed model for the energy/area consumption of scratchpad memories and is also used in our research work.

Observing, the above approaches on usage of the scratchpad memory, we find that there is a void in the research that has taken place. Thus, we propose an approach which is an extension to Steinke's [8] approach. In this approach, we partition an array variable for the maximal usage of the scratchpad memory. The next section presents the algorithm in detail, followed by a description of the workflow and the results.

III. Algorithm

Steinke considers Program Memory Objects PMO (basic blocks and functions) and Data Memory Objects DMO (variables) as possible candidates for moving onto the scratchpad. He associate a profit (energy saving) with each memory object (PMO and DMO) and formulates a knapsack problem [13]. Our approach works in the following stepwise manner.

1. Chooses a candidate array *A* among all the possible arrays for partitioning.
2. Decides whether to partition the array *A* or not. Decides whether partitioning will result in reduced energy consumption of the application.
3. If the array *A* is partitioned then decides the splitting point `split` of the array which leads to maximum reduction in the energy consumption.
4. Given the partitioning decision and the splitting point, modifies the original application according to the splitting point.

The array *A* chosen in step 1 is the highest valence (*i.e.* energy consumption per element) array which could not be moved onto the scratchpad by the Steinke's algorithm. Step 2 and step 3 are executed simultaneously while solving the formulated ILP. We describe the application transformation in the following subsection followed by the partitioning decision and the formal description of the problem.

A. Application Transformation

Application Transformation is an independent phase, which takes an application code, a chosen array *A* and a splitting

point `split` as inputs and outputs a transformed application. Specifically, it partitions the array *A* (refer Fig. 1) according to the splitting point to generate the two smaller arrays. Then it replaces the array *A* with the smaller arrays and modifies the application such that all the accesses to the array variable are replaced with an Access macro. The Access macro on the basis of the index variable and the splitting point `split` dynamically determines which of the two partitioned arrays are being referenced.

The underlying principle can be explained using a highlevel example. A simple reference to the array A

```
int A[size];
data = A[i];
```

is replaced by the following set of lines.

```
int Aleft[split], Aright[size-split];
ACCESS(data,i);
```

where ACCESS macro is defined as

```
#define ACCESS(data, index)
if    (index < split)
      data = Aleft[index];
else
      data = Aright[index-split];
```

B. Partitioning Decision

For the partitioning decision, we need to compare the energy consumption of the original application and the modified application. However, the application will be modified only when we prove that it leads to a more energy efficient solution. From subsection A we observe that only those program memory objects get modified which contain a reference to the array, while all the others remain invariant. Corresponding to every program memory object which contains atleast one reference to the array *A* we add an additional program memory object. This alternative program memory object will either be named Referenced Basic Block *RBB* or Referenced Function *RF* if it corresponds to either a basic block or a function respectively. Now we compare only the referenced functions and referenced basic blocks with their corresponding functions and basic blocks. Consequently, the number of additional variables in the problem is proportional to PMOs of the original application. We finally include a binary decision variable *DD* into the ILP representing the decision of array partitioning.

C. Splitting Point

First of all, as shown in Fig. 1 we combine a constant number *bsize* of adjacent array elements of the array *A* to form a Basis Element *b*. Now, the array instead of having *size* elements has $\lceil size/bsize \rceil$ elements. This reduces the complexity of the problem as well as the accuracy of the solution against the solution where each array element is a basis element(*i.e.* $bsize = 1$). Corresponding to all $\lceil size/bsize \rceil - 1$ splitting points (refer Fig. 2) we generate $2 * (\lceil size/bsize \rceil - 1)$ partitioned variables

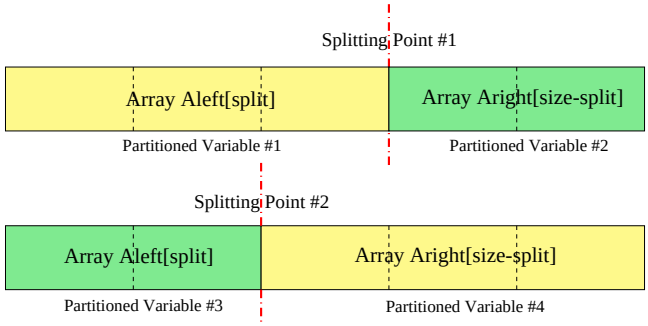

Fig. 2. splitting points and partitioned variables

pv as data memory objects. A partitioned variable *pv* consists of consecutive basis elements *b* such that only one comparison is required in the Access macro for accessing the partitioned variable. Two possible splitting points and their corresponding partitioned variables are shown in Fig. 2.

D. Integer Linear Problem

To decide between the original application and the modified application one approach is to compare their energy consumption and choose the one with lower energy consumption. The other approach is to consider their energy difference against a high energy value, name this as *energy saving* and choose the one with higher energy saving. We choose the latter approach as it aids the problem formulation. The high energy value corresponds to the system with only offchip memory. The partitioning decision is also dependent upon the choice of the splitting point as it determines the energy savings due to the partitioned arrays. The splitting point is in turn depends upon the size of the onchip scratchpad.

The problem can be viewed as a decision problem, to choose between referenced basic blocks, referenced functions and a partitioned array on one side and original basic blocks, functions and the unpartitioned array on the other side, under the objective to maximize energy savings. This has to be done such that the combined size of the chosen memory objects does not exceed the scratchpad size. Thus the problem can be viewed as a variant of the knapsack problem [13], which can be solved by Integer Linear Programming. The following sections formally describe the ILP formulation.

D1. Program Memory Objects

The energy saving by moving a Function or a Referenced Function *i* onto the scratchpad is the sum of the product of the number of executions m_k of each instruction *k* within the function *i*, with the energy saving of a single instruction fetch E_{instr_fetch}

$$E(F_i) = \sum_k m_k * E_{instr\_fetch}$$

where E_{instr_fetch} is the difference between the energy consumption in fetching an instruction from the offchip memory and the scratchpad memory.

$$E_{instr\_fetch} = E_{instr\_fetch\_main} - E_{instr\_fetch\_sp}$$

The energy saving due to a Basic Block or a Referenced Basic Block j is the number of executions of the Basic Block n_j multiplied with the product of the number of instructions in the Basic Block m with the difference of energy consumption of an Instruction Fetch E_{instr_fetch}. We need to subtract the energy of u jumps from the main memory to the scratchpad $E_{jump\_main2sp}$ and w jumps from the scratchpad to the main memory $E_{jump\_sp2main}$. Since the energies of these jumps are different, they have to be handled separately.

$$\begin{aligned} E(BB_j) &= m * n_j * E_{instr\_fetch} - \\ &\quad u * E_{jump\_main2sp} - \\ &\quad w * E_{jump\_sp2main} \end{aligned}$$

D2. Data Memory Objects

The energy saving by moving a variable v onto the scratchpad is the product of the number of accesses $acc(v)$ to the variable with the energy cost E_{data}

$$E(v) = acc(v) * E_{data}$$

where E_{data} is the difference in energy consumption of the memory access in a load or store instruction when the variable v is in the offchip or the scratchpad memory.

$$E_{data} = E_{data\_main} - E_{data\_sp}$$

The accesses $acc(pv)$ to a partitioned variable pv is the sum of the accesses $acc(b_i)$ to the basis elements b_i belonging to the partitioned variable.

$$acc(pv) = \sum_{b_i \in pv} acc(b_i)$$

The energy saving by a partitioned variable pv_n is the product of the number of accesses to the partitioned variable $acc(pv_n)$, with the energy difference E_{data}.

$$E(pv_n) = acc(pv_n) * E_{data}$$

D3. Objective Function

Our formulation of the problem uses the following definitions for moving functions F, referenced functions RF, basic blocks BB, referenced basic blocks RBB, variables v, and partitioned variables pv with $x \in F \cup RF \cup BB \cup RBB \cup v \cup pv$

$$\begin{aligned} E(x) &= \text{saved energy consumption for } x \\ S(x) &= \text{size of } x \end{aligned}$$

$$m(x) = \begin{cases} 1, & \text{if } x \text{ is moved to the scratchpad} \\ 0, & \text{otherwise} \end{cases}$$

We also need to decide whether it is beneficial to partition the array variable or not. For this purpose we include the decision variable DD.

$$DD = \begin{cases} 1, & \text{if the array variable } A \text{ is partitioned} \\ 0, & \text{otherwise} \end{cases}$$

To optimize for energy the following objective function of the knapsack problem needs to be maximized:

$$\begin{aligned} sav &= \sum_{i \in I} m(F_i) * E(F_i) + \sum_{l \in L} m(RF_l) * E(RF_l) + \\ &\quad \sum_{j \in J} m(BB_j) * E(BB_j) + \\ &\quad \sum_{m \in M} m(RBB_m) * E(RBB_m) + \\ &\quad \sum_{k \in K} m(v_k) * E(v_k) + \sum_{n \in N} m(pv_n) * E(pv_n) - \\ &\quad DD * E_{overhead} \end{aligned}$$

Index sets I, J, K, L, M and N correspond to index values for functions, basic blocks, variables, referenced functions, referenced basic blocks and partitioned variables respectively.

The overhead energy $E_{overhead}$ of partitioning the array variable A is subtracted from the energy savings. The overhead energy $E_{overhead}$ is the product of the total number of accesses $acc(A)$ to A, with the sum of the extra energy of executing $E_{exec}(ACCESS)$ the Access macro.

$$E_{overhead} = acc(A) * E_{exec}(ACCESS)$$

D4. Constraints

The size constraint can be modeled as follows:

$$\begin{aligned} &\sum_{i \in I} m(F_i) * S(F_i) + \sum_{j \in J} m(BB_j) * S(BB_j) + \\ &\sum_{k \in K} m(v_k) * S(v_k) + \sum_{l \in L} m(RF_l) * S(RF_l) + \\ &\sum_{m \in M} m(RBB_m) * S(RBB_m) + \sum_{n \in N} m(pv_n) * S(pv_n) \\ &\leq scratchpadsize \end{aligned}$$

We extend the constraints to be able to handle the following situations:

To ensure that the referenced variables (RBB and RF) and partitioned variables pv are not chosen when A is not partitioned (i.e. DD = 0):

$$\begin{aligned} \sum_{l \in L} m(RF_l) &+ \sum_{m \in M} m(RBB_m) + \\ \sum_{n \in N} m(pv_n) &- C * DD \leq 0 \end{aligned}$$

To ensure that the original variables (BB and F) and the array variable A are not chosen when it is partitioned (i.e. DD = 1):

$$\begin{aligned} \sum_{i \in I} m(F_i) &+ \sum_{j \in J} m(BB_j) + \\ m(A) &- C * (1 - DD) \leq 0 \end{aligned}$$

where C is any sufficiently large constant such that the above equations always remain less than or equal to zero.

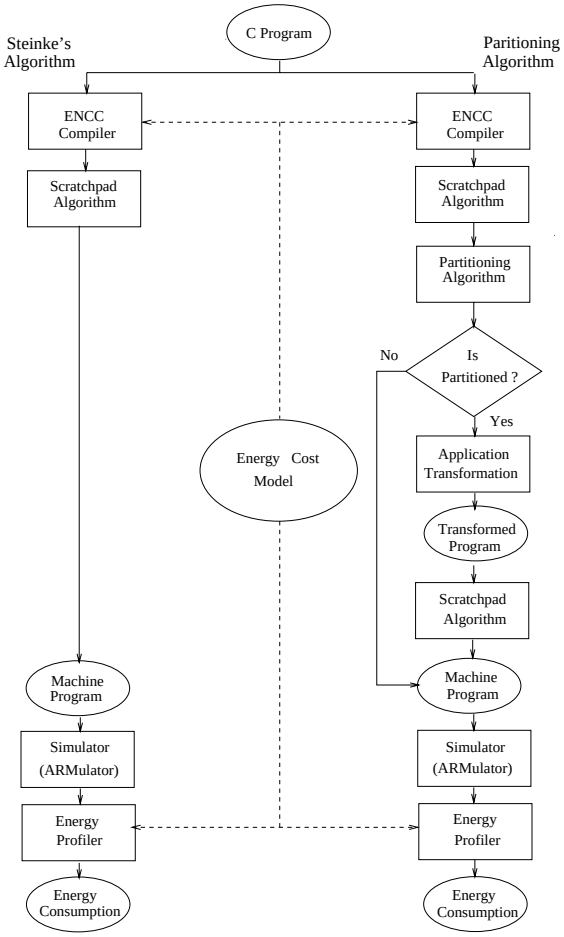

Fig. 3. Experimental Workflow

To enforce that only one variant of the partitioned array is selected to be moved to scratchpad:

$$\sum_{n \in N} m(pv_n) \leq 1$$

To regulate that either a Basic Block (Function) or its corresponding Referenced Basic Block (Referenced Function) is selected:

$$\begin{aligned} \forall l \in L: \quad & m(RF_l) + m(F_l) \quad \leq 1 \\ \forall m \in M: \quad & m(RBB_m) + m(BB_m) \quad \leq 1 \end{aligned}$$

Based on the above inequations, an ILP solver [10] can find the optimal solution for the given cost function. From the value of the decision variable *DD* and the size of the partitioned variable moved onto scratchpad, the result regarding partitioning of the array and the splitting point can be determined respectively. Consequently, if the array is partitioned, the application is modified appropriately and the chosen memory objects can then be placed onto the scratchpad memory using Steinke's Algorithm.

IV. Work Flow

The experiments were carried out following the workflow described in Fig. 3. In the first step the benchmark programs are compiled using the energy aware compiler ENCC [4]. For partitioning the array variable and use of the scratchpad, the algorithm in the previous section is executed. If the algorithm proposes to partition the array, the application is modified as described in subsection A. The modified application is again compiled and the machine code generated is simulated using the ARMulator (the simulator from ARM Ltd [1]). Based on the instruction trace, the energy profiler calculates the total amount of energy consumed for the different processor instructions and memory accesses taking into account all the inserted instructions for accessing the partitioned variables.

TABLE I
Processor Cycles for Scratchpad System

Access Type	number of cycles
scratchpad	1 cycle
main memory 16 bit	1 cycle + 1 wait state
main memory 32 bit	1 cycle + 3 wait states

According to the experimental setup the processor requires only 1 clock cycle and no wait states to access the scratchpad memory. On the other hand accessing main memory, requires 1 clock cycle and 1 or 3 wait states depending upon the data width (c.f. table I).

Energy consumption of the scratchpad memory is calculated using the CACTI model. However, the energy consumption of the main memory is measured from the evaluation board [2]. The energy values in the table II show a difference of 1 or 2 orders of magnitude for accessing scratchpad memory and main memory.

V. Results

For the experiments, benchmarks from different domains were selected. First, the *FFT* and *FIR* applications were studied, having a completely specified array variable access pattern. Second, the sorting algorithms *bubble sort*, *insertion sort* and *selection sort* which have a semi-specified array variable access pattern were chosen. Finally, a compression algorithm *ArithCoder* using Arithmetic Coding [22], having a completely undefined array variable access pattern was studied.

The results in Fig. 4 depict the comparison of the partitioning algorithm against Steinke's algorithm for *Selection Sort*. Partitioning is not done when the onchip scratchpad size is not

TABLE II
Energy Consumption of Memories

Memory Type	Memory Size	Energy (nJ)
scratchpad	128 Bytes	0.53
scratchpad	256 Bytes	0.61
scratchpad	512 Bytes	0.69
scratchpad	1024 Bytes	0.82
scratchpad	2048 Bytes	1.07
scratchpad	4096 Bytes	1.21
scratchpad	8192 Bytes	2.07
main memory 16 bit	512 KBytes	24.0
main memory 32 bit	512 KBytes	49.3

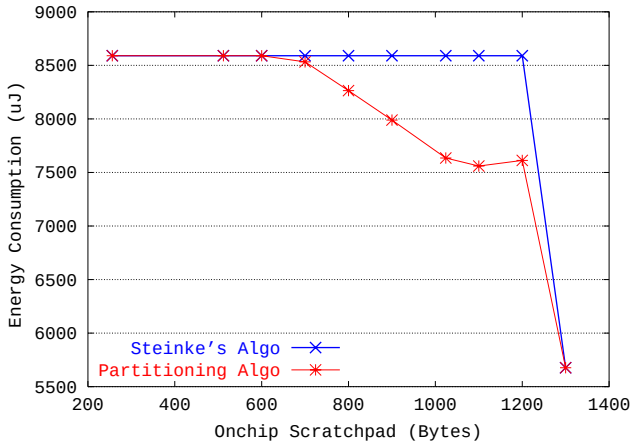

Fig. 4. Energy Comparison (Selection Sort)

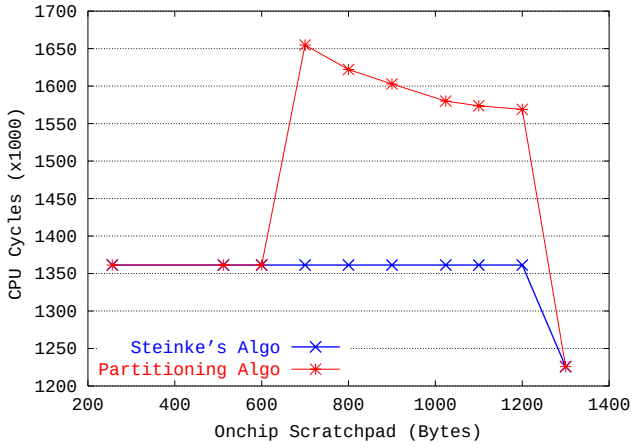

Fig. 5. Performance Comparison (Selection Sort)

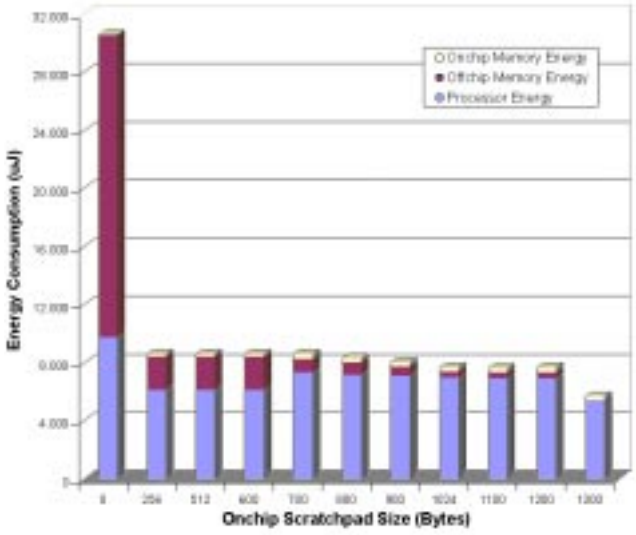

Fig. 6. Energy Distribution (Selection Sort)

TABLE III
Energy Steinke's Algorithm vs. Partitioning Algorithm [uJ]

benchmark	Steinke Algo.	Part. Algo.	% imp.
FFT	16,604.979	15,331.800	7.66%
FIR	5,669.055	4,671.960	17.59%
BubbleSort	15,719.219	13,877.833	11.71%
SelectionSort	8,590.022	7,612.881	11.37%
InsertionSort	6,301.395	5,710.620	9.37%
ArithCoder	16,945.211	15,972.100	5.74%
average			10.57%

large enough to optimize energy or when the scratchpad size is large enough to keep the whole array variable onchip. At these points (at 600 Bytes and 1300 Bytes in Fig. 5), our results are exactly the same as Steinke's Algorithm.

Since we are replacing a single statement to access the array with multiple statements, we are actually decreasing the performance of the application in terms of the CPU cycles. The results in Fig. 5 confirm the fact. Consequently, we prove that *"The application is optimized for energy and not for performance"* which is not a common phenomenon in the energy optimizations. The maximum energy saving that is achieved in the above example (Selection Sort) is 11.37% and with a performance degradation is 15.25% for the onchip scratchpad size of 1200 Bytes. Fig. 6 presents the distribution of the total energy as consumed by processor, onchip scratchpad memory and offchip memory. We observe a considerable drop in the offchip memory as well as total energy when we introduce an onchip scratchpad memory into the system, which then continuously decrease with increasing scratchpad size. We also observe an increase in the processor energy just at the point (700 Bytes), when we decide to partition the array.

The number of memory objects is linear to the number of the basic blocks and variables in the application. In our benchmarks, the size of basis element b was 80 Bytes and the average runtime for solving ILP inequations was observed to be less than 4s. The application transformation algorithm being a pattern matching algorithm was linear to the size of the application program and had minimal runtime for the benchmarks. Finally, the maximum energy savings which could be achieved against Steinke's algorithm are shown in table III and performance penalties in table IV.

TABLE IV
Performance Steinke's Algorithm vs. Partitioning Algorithm [cycles]

benchmark	Steinke Algo.	Part. Algo.	% imp.
FFT	1,344,537	1,283,481	+4.54%
FIR	690,852	764,948	-10.72%
BubbleSort	1,986,192	2,912,574	-46.64%
SelectionSort	1,361,253	1,568,885	-15.25%
InsertionSort	892,073	1,113,301	-24.79%
ArithCoder	2,192,162	2,208,645	-0.75%
average			-15.60%

VI. Conclusion & Future Work

It is evident from the results that the partitioning algorithm shows a reasonable improvement of 5.7% to 17.6% for applications from all domains including those with irregular array access pattern. The results from the partitioning algorithm are always better than or equal to Steinke's Algorithm. With this algorithm a further improvement is gained against commonly used Cache systems. Moreover the algorithm can be integrated into any compiler used for the scratchpad systems.

We would like to enhance this approach in the future by reducing the overhead due to the Access macro, by combining it with other optimizations *e.g. loop splitting*.

References

[1] ARM. *Advanced RISC Machines Ltd.* www.arm.com.

[2] AT91M40400 *ATMEL Corporation.* www.atmel.com.

[3] 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 the 10th International Symposium on Hardware/Software Codesign*, Estes Park,CO, May. 2002.

[4] encc. University of Dortmund, Computer Science Dep., ls12-www.cs.uni-dortmund.de/research/encc.

[5] J. L. Hennessy and D. A. Patterson. Computer Architecture A Quantative Approach, Morgan Kaufmann Publisher Inc., CA. 1990

[6] Intel. Mobile Power Guidelines 2000. Technical Report 1.0, Intel Corporation, Dec. 1998.

[7] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratchpad memory Space. In *Proc. of 38th DAC 01*, pages 690–695, Las Vegas, NV, Jun. 2001.

[8] S. Steinke, L. Wehmeyer, B. S. Lee and P. Marwedel, Assigning program and data objects to scratchpad for energy reduction. In *Proc. of the DATE Conference 2002* Paris, France, Mar. 2002.

[9] C. Ferdinand, F. Martin, R. Wilhelm, Applying compiler techniques to cache behaviour prediction. In *Proc. of Workshop on Languages, Compilers and Tools for Real-Time Systems* Las Vegas, Nevada, page 37-46, Jun. 1997

[10] G. L. Nehmhauser and L. A. Wolsey. *Integer And Combinatorial Optimization*. John Wiley and Sons, New York, NY, 1988.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proc. of European Design and Test Conference*, Paris, France, Mar. 1997.

[12] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues In Embedded Systems-on-chip*. Kluwer Academic Publishers, Norwell, MA, 1999.

[13] R. Sedgewick. *Algorithms*. Addison Wesley, Massachusetts, 1988.

[14] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. of the 36th DAC 99*, pages 867–872, New Orleans, LA, Jun. 1999.

[15] J. Sjödin, B. Fröderberg, and T. Lindgren. Allocation of global data objects in on-Chip RAM. In *Proc. Workshop on Compiler and Architectural Support for Embedded Computer Systems*, Washington DC, Dec. 1998.

[16] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *PATMOS 01*, Yverdon-Les-Bains, Switzerland, Sep. 2001.

[17] Synopsys. *Power products reference manual V3.5*. synopsys, 1996.

[18] V. Tiwari. *Logic and system design for low power consumption*. PhD Thesis, Princeton University, Princeton, NJ, 1996.

[19] V. Tiwari, S. Malik, and A. Wolfe. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing Systems*, 13(3):223–238 Aug. 1996.

[20] S. J. E. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[21] M. E. Wolf and M. S. Lam A loop transformation theory and an algorithm to maximise parallelism In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing Aug. 1990*

[22] A. Moffat, R. Neal and I. H. Witten Arithmetic coding revised *ACM Transactions on Information Systems, 16(3):256-294, 1998.*