Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization

Heiko Falk and Manish Verma

University of Dortmund, Computer Science 12, D-44221 Dortmund, Germany

Abstract. For mobile embedded systems, the energy consumption is a limiting factor because of today's battery capacities. Besides the processor, memory accesses consume a high amount of energy. The use of additional less power hungry memories like caches or scratchpads is thus common. This paper presents a combined approach for energy consumption minimization consisting of two complementary and phase-coupled optimizations, viz. data partitioning and loop nest splitting. In a first step, data partitioning partitions large arrays found in typical embedded software into smaller ones which are placed onto an on-chip scratchpad memory. Although being effective w. r. t. energy dissipation, this optimization adds overhead to the code since the correct part of a partitioned array has to be selected at runtime. Therefore, the control flow is optimized as a second step in our framework. In this phase, loop nests containing *if*-statements are split using genetic algorithms leading to minimized *if*-statement executions. However, loop nest splitting leads to an increase in code size and can potentially annul the program layout achieved by the first step. Consequently, the proposed approach iteratively applies these optimizations till a local optimum is found.
The proposed framework of combined memory and control flow optimization leads to considerable energy savings for a representative set of typical embedded software routines. Using an accurate energy model for the ARM7 processor, energy savings between 20.3% and 43.3% were measured.

1 Introduction

The emergence of portable or mobile computing and communication devices such as cellular phones, pagers, handheld video games etc. is probably the most important factor driving the need for low power design. Current battery technologies such as Li-Io have capacities of 90 Watt-hours / *kg* [1], meaning that 10 hours of operation for a device consuming 20 W of operating power would require a battery weight of around 2.2 *kg*. Thus, the cost and weight of the batteries become bottlenecks that prevent the reduction of system cost and weight unless efficient low power design techniques are adopted.

Since it has been shown that 50%–75% of the power consumption in embedded multimedia systems is caused by memory accesses [2, 3], the efficient utilization of memories is of major interest for the construction of low power devices. Main memory is the slowest and the most energy consuming memory type. On one hand, the high amount of main memory accesses is a reason for its high energy dissipation. On the other hand, the high latency during main memory accesses causes several wait states to a processor. To avoid these problems, a very effective way of energy reduction is to build up a memory hierarchy.

Additional memories are able to reduce the number of main memory accesses for frequently used instructions or variables. *Caches* are well known and included in many processor designs. Besides the data memory itself, they consist of an additional tag memory and of logic components enabling the fast comparison of addresses with the contents of the tag memory. The advantage of caches is their easy integration into a system since the detection of cache hits is done automatically by the hardware. For embedded systems, caches are often not well suited due to their inherently high energy consumption during tag memory access and comparison. Additionally, the accurate determination of worst case execution times often is difficult in a cache-based environment which is a critical issue for embedded real-time systems [4].

Recently, the utilization of *scratchpad memories* has become an important alternative to caches [5, 6]. A scratchpad is a small memory mapped into the processor's address space requiring only simple address decoders. The absence of logic components checking the validity of data is the reason for their low energy consumption. However, this property requires a careful mapping of instructions or data to the memory which has to be done by the programmer or the compiler.

This paper presents a novel combination of automated compiler optimizations for achieving an energy efficient utilization of scratchpad memories. For a given ARM7 based system architecture, a *data partitioning* step is performed first [7]. In this step, parts of a program and of its data are assigned optimally (i. e. least energy consuming) to the scratchpad. In particular, large arrays frequently found in data-dominated embedded software are split into several pieces. Data partitioning enables the storage of fragments of the original array in the scratchpad which was impossible before.

Since the software must take care of accessing the correct part of a split array, *if*-statements dynamically selecting the appropriate sub-array have to be inserted into the program's code. Although being beneficial w. r. t. energy consumption, data partitioning adds overhead to the software. The execution of the additional *if*-statements requires several CPU cycles and can lead to degraded pipeline performance due to pipeline stalls. To avoid this overhead, we apply a significantly improved loop splitting technique for control flow optimization (originating from [8, 9]) as the second step after data partitioning. This optimization minimizes *if*-statement executions so that the negative side-effects of data partitioning are largely eliminated. The improved control flow is achieved at the cost of increased code size. The size of the program and data objects placed on the scratchpad is bounded by the scratchpad size. Any increase in code size could potentially invalidate the program and data layout achieved in the previous step. Nevertheless, the size of frequently executed program objects is increased by a small amount and a hill-climbing approach can be utilized to obtain a locally optimal solution.

The main contribution of this paper is the successful exploration of the synergy effects of these two individual optimizations. The results clearly show that our control flow optimization is able to eliminate overheads caused by data partitioning. However, data partitioning enables loop nest splitting in our framework since loop nest splitting without data partitioning would be impossible. These two optimizations are related to each other in a phase-coupled sense and complement each other such that, when combined, they lead to considerable energy savings. Our proposed combined approach ensures that the finally generated code contains a minimum of control flow overhead but a

maximum of data stored in low energy scratchpads. Additionally, the recently presented algorithms for loop nest splitting are extended substantially. Using the techniques described in this paper, we are able to relax restrictions imposed by the original analysis algorithms [8, 9] so that more general classes of applications can be optimized.

A survey of work related to memory hierarchy exploitation and low power code generation is given in section 2. The phase-coupled algorithm for combined data partitioning and loop nest splitting is presented in section 3. Section 4 presents the techniques used for data partitioning, whereas loop nest splitting and its new extensions are described in section 5. Section 6 contains a detailed description of the measured experimental results, and section 7 summarizes and concludes this paper.

2 Related Work

Code optimizations for caches include well-known techniques like *loop tiling*, *loop interchange* or *loop fusion* [10, 11]. Since applications spend most of their runtime in innermost loops, these optimizations concentrate on loop nests. The iteration space of loop nests is reordered in such a way that a higher locality of data accesses is achieved. The higher the spatial and temporal locality of data accesses, the fewer cache misses occur during program execution resulting in a more efficient cache utilization. Even nowadays, locality optimization is an area of ongoing research. In [12], loops are aligned so that the time between two successive accesses to the same memory location (*temporal locality*) is minimized. A graph based optimization strategy is used in [13] to cluster array references in loops with spatial or temporal reuse. This technique leads to average reductions of cache misses by 13.8%.

Array padding [10] is a good example of a data layout transformation. Here, unused data locations are inserted between columns of an array so as to reduce cache set conflicts and cache miss jamming. An approach for simultaneous generation of optimized data layouts and temporal locality improvement is presented in [14]. In this article, geometric models and algorithms are used to minimize TLB misses.

Panda et al. [15] presented an efficient use of a memory hierarchy by placing the most commonly used variables onto the scratchpad. The dynamic copying of array parts was studied in [5]. However, the preconditions imposed by this algorithm are very restrictive so that only a limited class of applications can be analyzed. An approach for statically placing both data and program parts onto a scratchpad memory basing on the analysis of instruction execution and variable accesses is presented in [16]. In that paper, only whole variables are considered at a time. Consequently, a large non-scalar variable (i. e. an array) could either be placed onto the scratchpad as a whole or not at all, potentially leading to a sub-optimal memory utilization.

A methodology of source code transformations for data access and storage management (*DTSE*) is presented in [17]. The described techniques can be applied to complex memory hierarchies consisting of multi-level caches and on-chip memories. However, the authors only focus on the optimization of data flow and neglect that the control flow gets very irregular since many additional *if*-statements are inserted. This impaired control flow has not yet been targeted by the authors. The results given in our paper clearly show the importance of combining data and control flow optimizations for the design

of low power embedded systems. In the following section, we present the combined approach for phase-coupled data partitioning and loop nest splitting.

3 Algorithm for Phase-Coupled Energy Minimization

As will be described in section 4, data partitioning improves scratchpad utilization but impairs the control flow due to the addition of *if*-statements. Loop nest splitting improves the control flow but increases the code size and can potentially annul the allocation of the scratchpad. Hence, it is obvious that both optimizations influence each other. As a consequence, data partitioning and loop nest splitting need to be applied in a phase coupled manner which is described in this section.

```
1 Algorithm(ΔSPLIT) {
2   Energy = INFINITY;
3   SPLIT = DataPartitioning();
4   do {
5     LoopNestSplitting(SPLIT);
6     OldEnergy = Energy;
7     if (SP-Objects <= ScratchpadSize)
8       Energy = ComputeEnergy(SPLIT);
9     else
10      Energy = INFINITY;
11    SPLIT -= ΔSPLIT;
12    ΔEnergy = OldEnergy - Energy;
13  } while(ΔEnergy ≥ 0);
14  return(OldEnergy); }
```

Fig. 1. Algorithm for energy minimization

The proposed algorithm (cf. fig. 1) implements a hill-climbing approach. It starts with an initial scratchpad allocation obtained after data partitioning (line 3) followed by loop nest splitting (line 5). Since data partitioning defines which basic blocks and array fragments are placed on the scratchpad, code size increases of individual basic blocks after loop nest splitting may lead to an invalid solution exceeding the scratchpad's capacity. This situation is checked after loop nest splitting (lines 7–10). For a valid allocation, the energy consumption is computed (line 8) and the change in energy dissipation caused by loop nest splitting is calculated (line 12). Since Δ`Energy` is positive in this case, the algorithm's loop steps into a second iteration. There, the algorithm must necessarily stop since no better valid solution can be obtained due to the optimality of the integer linear programming based approach of data partitioning.

However, if the initial allocation is invalid, the algorithm iteratively tries to obtain a valid solution. For this purpose, the splitting point `SPLIT` originally stemming from data partitioning is adjusted by a user defined offset Δ`SPLIT`, and loop nest splitting is re-applied again. The algorithm stops when it has ascertained a locally optimal solution. Since neither data partitioning nor loop nest splitting modify existing data dependencies between array elements, the algorithm of figure 1 does not need to do data dependence analysis.

4 Data Partitioning

Considering aggregate array variables as the candidates for placement onto the small on-chip memory is not the ideal decision, as this may lead to the under-utilization of the on-chip memory and to a high energy consumption by the application. The proposed data partitioning approach rectifies the aforesaid problem by partitioning an array present in the application into smaller array variables. The energy consumption of the application is then reduced by placing one of the smaller array variables onto the

on-chip memory, which is not large enough to contain the whole array variable. The data partitioning approach works in the following stepwise manner:

1. Whenever beneficial, the scratchpad is maximally filled with arrays entirely fitting into this memory [16].
2. Among all remaining arrays, one candidate array A is chosen for partitioning.
3. For A, a splitting point `SPLIT` is computed leading to the maximum reduction of energy consumption. If no splitting point exists leading to a reduction of energy dissipation, A is not split. It is proceeded to step 2 where another array is chosen.
4. Given the array A and the splitting point, the original application is transformed.

The array having the highest valence (i. e. energy consumption per element) which could not be placed on the scratchpad memory in its entirety by the algorithm presented in [16] is chosen in step 2. Step 3 is solved in a phase coupled manner using *integer linear programming (ILP)* (cf. section 4.1). The step of application transformation is described in section 4.2.

4.1 Integer Linear Program

The formulation of the integer linear program for data partitioning is based on the observation that the splitting of an array implicitly leads to changes of an application's code due to the selection of the correct fragment of a split array during runtime. If an array A is to be split, only those basic blocks of an application need to be modified which access A. These modified basic blocks which access a split array A are termed as *referencing basic blocks* in the following.

For a program containing m basic blocks referencing an array A of length n, the integer linear program used for data partitioning is basically formulated using a set $V := V_{BB} \cup V_{RBB} \cup DD \cup V_A$ of binary decision variables. V_{BB} and V_{RBB} each consist of m decision variables. A variable v^i_{BB} equal to 1 ($1 \leq i \leq m$) (v^i_{RBB} resp.) denotes the case where basic block i (referencing basic block i, resp.) is placed on the scratchpad. DD is a single binary variable which is equal to 1 only if array A is partitioned. A variable $v^j_A \in V_A$ equal to 1 ($1 \leq j \leq n$) models the splitting of array A at position j.

Using these decision variables, data partitioning is formulated as a knapsack problem. To each basic block, referencing basic block and split parts of array A, their corresponding sizes are attached. The integer linear program contains constraints in order to ensure that the size of all objects placed on the scratchpad does not exceed the scratchpad's capacity. Additional constraints are included guaranteeing the consistency of an actual assignment of values to the decision variables (e. g. if a referencing basic block RBB_i is placed on the scratchpad, its original counterpart BB_i must not be considered).

The objective function to be maximized during data partitioning models the energy savings achieved by an assignment of values to the decision variables, compared to the energy consumption of an unpartitioned application totally stored in background main memory. The objective function considers all energy related aspects of data partitioning: access frequencies of basic blocks and array elements, the savings due to the placement of objects on the scratchpad and the overhead due to the more complex code of the referencing basic blocks in order to access the correct part of a split array at runtime. For more details, the interested reader is referred to the original publication [7].

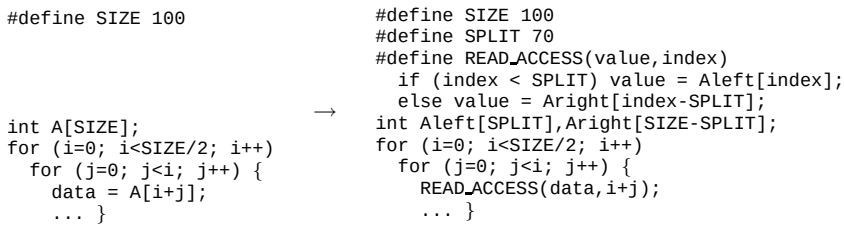

Fig. 2. A typical code fragment before and after data partitioning

4.2 Application Transformation

The application transformation step takes an application code, a chosen array *A* and a splitting point `SPLIT` as inputs and outputs a transformed application. Specifically, *A* is replaced by two smaller arrays `Aleft` and `Aright` (cf. figure 2) which are generated according to `SPLIT`. The application code is modified in such a way that all accesses to the array *A* are replaced by an *access macro*. The access macro determines which of the two smaller arrays are being referenced on the basis of the index expression and the splitting point `SPLIT`. Figure 2 displays a typical example of the original and the modified application codes. The access macro in the figure 2 represents a read access to the array *A*. A similar access macro can also be constructed for the write accesses as well, though it is omitted for the sake of simplicity of the code examples.

As can be seen from figure 2, the proposed way of data partitioning leads to a bi-partitioning of arrays. Generally, the integer linear program can easily be rewritten so that *n*-way partitions are supported. In the context of the ARM7 based system studied in section 6, it turned out that the generation of partitions larger than 2 is disadvantageous since the resulting complex if-then-else structures over-compensate any savings. Similarly, an e. g. trapezoidal array partitioning is not beneficial due to the increased control flow overhead.

5 Loop Nest Splitting for Control Flow Optimization

As described in the previous section, preprocessor macros containing *if*-statements are inserted in a program's code for selecting the correct partition of a split array during runtime. Given that array references typically occur in the innermost loops of embedded software, these *if*-statements constitute an overhead w. r. t. runtime and energy consumption which should not be neglected. We propose to apply a substantially improved variant of *loop nest splitting* [8] for the optimization of these *if*-statements.

The transformation presented in our paper determines ranges of loop iterations where all *if*-statements in the loop nest are provably satisfied. Using this information, the loop nest is rewritten so that no *if*-statement is executed for these iteration ranges. In figure 3, the example code shown in figure 2 is depicted before and after loop nest splitting (note that the access macro shown in figure 2 is expanded now). Loop nest splitting detects that the outer `i` loop iterates from 0 to 49, while the inner `j` loop steps from 0 to the actual value of `i`. Considering the condition `i+j<70` inserted by data partitioning, it is recognized that this condition must necessarily be true for $\texttt{i} \leq 35$. Using

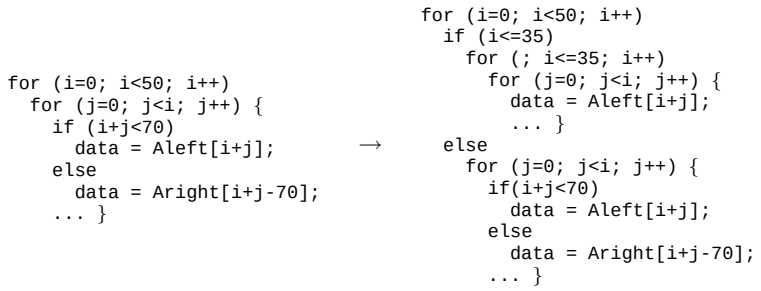

Fig. 3. A typical code fragment before and after loop nest splitting

this information, a new *if*-statement (the *splitting-if*) is inserted in the loop nest exactly checking this condition. Since `i<=35` implies that `i+j<70` is true, the *then*-part of the splitting-if consists of the body of the `i` loop without any *if*-statements. To minimize executions of the splitting-if, a second `i` loop is inserted in the *then*-part counting to the corresponding bound of the iteration ranges (i. e. this loop ensures that all iterations of the `i` loop up to `i <= 35` are executed without any further evaluation of the splitting-if). The *else*-part of the splitting-if is an exact copy of the original loop body. Using this code transformation, a reduction of *if*-statement executions from 1,225 down to 610 can be achieved for the codes depicted in figure 3.

The techniques for loop splitting presented previously [8] are limited such that only restricted classes of loops can be analyzed. Since all loop bounds are required to be constant, a loop nest as depicted in figure 3 could not be optimized. Section 5.1 briefly reflects the basic concepts of loop nest splitting. In section 5.2, the techniques of [8] are significantly extended enabling the optimization of more general classes of applications.

5.1 Analysis Techniques for Loop Nest Splitting

This section gives a brief summary of the recently published concepts for splitting loop nests [8]. For a given loop nest $\Lambda = \{L_1, \ldots, L_N\}$ of depth N, L_l denotes a single loop l with its index variable i_l and the lower and upper bounds, lb_l and ub_l, respectively. Every loop L_l can contain one or more *if*-statement whose conditions depend on the index variables of Λ. Such conditions are said to be loop-dependent. The *if*-statements must have the format `if (`$C_1 \oplus C_2 \oplus \ldots$`)` where C_x are loop-dependent conditions that are combined using logical operators $\oplus \in \{\wedge, \vee\}$. Every single loop-dependent condition C of an *if*-statement has to be an affine expression of the index variables i_l and can thus be represented as $C = \sum_{l=1}^{N} (c_l * i_l) + c \geq 0$ for constants $c_l, c \in \mathbb{Z}$.

The analysis for loop nest splitting consists of three different stages. In the first step called **Condition Optimization**, all loop-dependent conditions C are analyzed separately using a genetic algorithm (*GA*). GAs are employed since ILP is not suitable due to the non-linear objective function. For every condition C and every loop L_l, two values $lb'_{C,l}$ and $ub'_{C,l}$ are determined during this step. These values represent ranges of iterations of the loop nest where condition C is satisfied for all index variables i_l with $lb'_{C,l} \leq i_l \leq ub'_{C,l}$. These values are chosen in such a way that a loop nest splitting using

$lb'_{C,l}$ and $ub'_{C,l}$ minimizes the total execution of *if*-statements. For this purpose, the fitness function of the GA computes the total number of executed *if*-statements after loop nest splitting for a given set of $lb'_{C,l}$ and $ub'_{C,l}$ values. These computations are done very efficiently in linear time using a set of numerical formulas. The fewer the number of executed *if*-statements, the higher is the fitness of an individual.

Since this GA gives optimized results only for a single condition of an *if*-statement, these partial results need to be combined which is done during **Global Search Space Construction**. Here, for every condition C and its associated values $lb'_{C,l}$ and $ub'_{C,l}$, a polyhedron [18] P_C is generated. In general, a polyhedron P is a set of points in an N-dimensional geometric space which is defined by linear inequalities: $P = \{x \in \mathbb{Z}^N \mid Ax \geq b\}$ for an integer matrix A and a constant vector b. For the construction of P_C, constraints of the format $i_l \geq lb'_{C,l}$ and $ub'_{C,l} \geq i_l$ are used. Furthermore, appropriate constraints for all lower and upper loop bounds are needed: $i_l \geq lb_l$ and $ub_l \geq i_l$. If two conditions C_x and C_y are connected using the && operator in an *if*-statement, the corresponding polyhedra P_x and P_y are intersected. For the || operator, the union of polyhedra is used. This way, polyhedra can be built representing those iterations of the loop nest where a single *if*-statement is satisfied. Since all *if*-statements in a loop nest need to be fulfilled for loop nest splitting, all these polyhedra representing a single *if*-statement need to be combined using the intersection. The resulting polyhedron G called the *global search space* represents those loop nest iterations where all *if*-statements are satisfied.

Due to the nature of the union operator, G is a finite union of polyhedra: $G = R_1 \cup R_2 \cup \ldots \cup R_M$. Each polyhedron R_r of G defines a region where all *if*-statements in a loop nest are satisfied. But it should be avoided to use all such regions of G for loop nest splitting since this would lead to an increased number of executions of *if*-statements. Therefore, a second GA is applied to G for a **Global Search Space Exploration**. The goal of this second GA is to select only those regions R_r of G leading to a total minimization of *if*-statement executions. After the termination of the GA, only the constraints of the selected regions R_r are considered for generating the conditions of the splitting if-statement.

5.2 Modeling and Analysis of Loops with Non-Constant Bounds

Although the techniques summarized in the previous section already lead to large improvements [8], they are not applicable in the area of efficient memory hierarchy exploitation. As previously mentioned, all loop bounds lb_l and ub_l are explicitly required to be constant. This section presents extensions and improvements of [8] which eliminate this restriction and allow loop nest splitting to be applied to more general classes of embedded software, including popular sorting algorithms and DSP filter routines.

The only advantage of constant loop bounds as required in [8] is the simplicity of the GA for condition optimization. For values $lb'_{C,l}$ and $ub'_{C,l}$ generated by the GA, the fitness function only evaluates some formulas consisting of sum-of-products of the constant loop bounds and $lb'_{C,l}$ resp. $ub'_{C,l}$. This way, the exact number of *if*-statement executions can be computed and minimized. The restriction to constant loop bounds is not used elsewhere during loop nest splitting. In general, the basic structure of loop nest splitting as previously summarized is able to treat non-constant loop bounds, since the employed polyhedral models support complex iteration spaces.

Hence, we decided to maintain the basic structure of the analysis algorithms for loop nest splitting consisting of condition optimization, global search space construction and search space exploration. The problem formulation for loop nest splitting is given by:

Definition 1 (Loop Nest Splitting). *Let $\Lambda = \{L_1, \ldots, L_N\}$ be a loop nest of depth N. For every loop $L_l \in \Lambda$, a pair (lb'_l, ub'_l) is computed defining an interval of the index variable i_l. For $lb'_l \leq i_l \leq ub'_l$, all loop-variant* if*-statements in Λ are satisfied. (lb'_l, ub'_l) is chosen such that a minimization of* if*-statement executions is achieved.*

Since this is similar to the satisfiability of constraints with simultaneous minimization of an objective function, definition 1 describes an NP complete problem. The use of GAs here is motivated by their ability to find high-quality solutions especially for such complex optimization problems [19]. Furthermore, other well-known optimization strategies (e. g. ILP) can not be used due to the non-linearity of the objective function.

As already mentioned, polyhedral models are an integral part of the analysis. Since they base on linear inequalities, we allow loop bounds to be affine expressions:

Definition 2 (Affine Loop Bounds). *Let $\Lambda = \{L_1, \ldots, L_N\}$ be a loop nest of depth N.*

1. *For the outermost loop L_1, the bounds lb_1 and ub_1 are constant values.*
2. *For any other loop $L_l \in \{L_2, \ldots, L_N\}$, the loop bounds are affine expressions of the surrounding variables $i_1, \ldots, i_{l-1}$. Hence, the index variable i_l iterates between*
$$lb_l = \sum_{j=1}^{l-1} (c'_j * i_j) + c' \leq i_l \leq \sum_{j=1}^{l-1} (c''_j * i_j) + c'' = ub_l \qquad (c'_j, c', c''_j, c'' \in \mathbb{Z} \text{ const.})$$

The constant outermost loop bounds ensure that Λ is still fully analyzable at compile time due to the absence of data dependencies. The variable inner bounds imply that the number of *if*-statement executions subject to condition optimization not only depends on the values $(lb'_{C,l}, ub'_{C,l})$ generated by the GA. Now, implicit dependencies on a variable i_j exist if the bounds of an inner loop L_l depend on i_j. Hence, the formerly used formulas computing the *if*-statement executions are invalid. Instead, the GA's fitness function is restructured such that it models Λ after splitting using a set of $(lb'_{C,l}, ub'_{C,l})$ values. For this purpose, the following chromosomal encoding is employed:

Definition 3 (Chromosomal Encoding). *Let $\Lambda = \{L_1, \ldots, L_N\}$ be a loop nest.*

1. *A chromosome C is an array of integer values of length $2 * N + 1$.*
2. *For $l \in \{1, \ldots, N\}$, gene $C[2 * l - 1]$ ($C[2 * l]$ resp.) denotes $lb'_{C,l}$ ($ub'_{C,l}$ resp.). This way, the GA defines the regions of iterations with satisfied condition C.*
3. *Gene $C[2 * N + 1]$ stores the* innermost loop λ *for loop nest splitting. This gene states that loop nest Λ is split at loop L_λ, i. e. the splitting-if is placed in loop L_λ.*

The pseudo-code of the fitness function for optimizing a condition C is depicted in figure 4. The main goal of this function is to update two counters accurately. The first one (`if_count`) stores the number of *if*-statement executions for a given chromosome C. `penalty` counts how many times condition C is not satisfied when it is supposed by C that it should be satisfied. Using `penalty`, illegal individuals generated by the GA are detected.

Principally, this fitness function contains the entire loop nest Λ as can be seen from lines 3 and 17 of figure 4 (dots denote the omitted loops $L_2, \ldots, L_{N-1}$). For every loop,

code is required to model Λ after a potential splitting (lines 4–15 resp. 18–25). When entering a loop L_l, it is first checked whether L_l contains the splitting-if (cf. lines 4 and 18). If this is not the case, the algorithm proceeds to loop L_{l+1} (line 15).

Otherwise, L_l contains the splitting-if (lines 6 and 20) whose execution requires to increment `if_count` (lines 5 and 19). The splitting-if for loop L_l checks genes $C[1],\ldots,C[2*l]$ and verifies that the index variables $i_1,\ldots,i_l$ actually are within the ranges specified by these genes. If the splitting-if is true, the duplicated loop L_l (cf. the second `i` loop in figure 3) counting to the new upper bound $ub'_{C,l}$ is executed next (lines 7 and 21). Within this loop, the remaining loop nest $L_{l+1},\ldots,L_N$ can be found (lines 8–9). Since the splitting-if is true when executing this code, it is assumed that condition C is also true so that counter `if_count` is not altered in lines 7–10 resp. 21–22. But since the GA can generate illegal individuals for which condition C is false, care has to be taken to detect these situations. Lines 10 and 22 check whether the condition C is true or not. If it is false, an illegal individual is detected and counter `penalty` is incremented.

```
1 double fitness(lb'_{C,1},ub'_{C,1},...,lb'_{C,N},ub'_{C,N},λ) {
2  int if_count=0, penalty=0;
3  for (lb_1 ≤ i_1 ≤ ub_1) {
4   if (λ==1) {
5    if_count++;
6    if (i_1 ≥ lb'_{C,1} && i_1 ≤ ub'_{C,1}) {
7     for (i_1 ≤ ub'_{C,1})
8      ...
9       for (lb_N ≤ i_N ≤ ub_N)
10       if (!satisfied(C)) penalty++;
11   } else
12    ...
13     for (lb_N ≤ i_N ≤ ub_N)
14      if_count++;
15  } else
16   ...
17    for (lb_N ≤ i_N ≤ ub_N) {
18     if (λ==N) {
19      if_count++;
20      if (i_1 ≥ lb'_{C,1} && i_1 ≤ ub'_{C,1} && ... &&
            i_N ≥ lb'_{C,N} && i_N ≤ ub'_{C,N}) {
21       for (i_N ≤ ub'_{C,N})
22        if (!satisfied(C)) penalty++;
23      } else
24       if_count++;
25     } ... }
26 if (penalty==0)
27  return((double) (1/if_count));
28 else
29  return((double) (1/(penalty+ERR))); }
```

Fig. 4. Fitness function for condition optimization

Finally, some code is required for a false splitting-if in loop L_l. In analogy to figure 3, the remaining loop nest $L_{l+1},\ldots,L_N$ is copied into the *else*-part of the splitting-if (see lines 11–13). In loop L_N, the counter `if_count` is incremented (lines 14 and 24) since an *if*-statement checking condition C would be executed.

The fitness function terminates by returning the fitness of an individual based on the counters `if_count` and `penalty`. If an individual is valid, counter `penalty` is zero (line 26) and the inverse of `if_count` is returned to the GA. This way, individuals implying few *if*-statement executions have a high fitness. For an invalid individual (line 29), a large constant *ERR* is added to `penalty`, and the inverse of this sum is passed to the GA. This way, illegal individuals can never have a better fitness than legal ones.

As can be seen from figure 4, this fitness function has exponential complexity. Its runtime now depends on the depth N of a loop nest and the actual loop bounds. Affine loop bounds (cf. definition 2) ensure that the following analysis steps of global search space construction and search space exploration do not need to be modified since the affine bounds can directly be modeled by corresponding polyhedral constraints.

Memory Type	#Cycles
Scratchpad	1 Cyc
Main mem. (16 bit)	1 Cyc + 1 wait state
Main mem. (32 bit)	1 Cyc + 3 wait states

Table 1. CPU cycles for memories

Memory Type	Size	Energy [nJ]	Memory Type	Size	Energy [nJ]
Scratchpad	128 B	0.53	Scratchpad	256 B	0.61
Scratchpad	512 B	0.69	Scratchpad	1 kB	0.82
Scratchpad	2 kB	1.07	Scratchpad	4 kB	1.21
Scratchpad	8 kB	2.07	Main (16 bit)	512 kB	24.0
Main (32 bit)	512 kB	49.3			

Table 2. Energy consumption of memories

6 Benchmarking Results

The data partitioning and loop nest splitting techniques presented in this paper are fully implemented. To demonstrate the efficacy of our combined approach, typical embedded system benchmarks were passed through the algorithm described in section 3. The unchanged original source codes and the finally generated codes of the benchmarks (i. e. the code after data partitioning and loop nest splitting) were fed into an energy-aware research compiler [20] for the ARM7 architecture. During compilation, all optimizations were enabled to explore the maximum optimization potential. The generated assembly outputs of the compiler were finally processed by a simulator and an energy profiler. Both the compiler and the energy profiler make use of an instruction-level energy model [21] for the ARM7 having a very high accuracy of 98.3%. This way, the energy consumption and runtimes of our benchmarks can be computed reliably. The key characteristics of this energy model are briefly summarized in tables 1 and 2. Since data partitioning and loop nest splitting are performed at the level of C source codes as also illustrated by the code examples given in sections 4 and 5, our framework can easily be ported to other processors by simply providing an appropriate energy model.

For the experiments, benchmarks from different domains were selected. First, we used a 40 order *FIR* filter as a typical embedded DSP algorithm. Second, the sorting algorithms *insertion sort (INS)* and *selection sort (SELS)* were analyzed. Finally, a complete MPEG4 motion estimation routine *(ME)* was studied. The relevance of the extensions for loop nest splitting presented this paper is clearly demonstrated by the fact that a splitting of the FIR, INS and SELS benchmarks is impossible using the restricted techniques originally presented in [8]. The runtimes of our implemented tools are very low, not more than 39.8 CPU seconds are required to execute the algorithms described in sections 4 and 5 on a Sun Blade 1000 running at 750 MHz. The maximum contribution of loop nest splitting to these runtimes only amounts to 5.18 CPU seconds. Experiments were conducted by varying the scratchpad sizes. The following figures 5 and 6 show the impact of our optimization methodology for scratchpad sizes which are individually tuned for every benchmark, but kept fixed during all measurements. In these cases, a memory size of 1.8 kB was used for FIR, and 1.3 kB were used for INS and SELS. The ME routine with its large video frames was analyzed using a memory size of 119 kB. In order to demonstrate the stability of the proposed optimization methodology, detailed results for a large variety of scratchpad sizes are given in figure 7 using the SELS benchmark.

Figure 5 shows the effects of the combination of data partitioning and loop nest splitting on the energy consumption of the benchmarks. All results are shown as a percentage of the original unoptimized benchmark codes denoted as 100%. For both data partitioning and loop nest splitting, the relative energy consumed by the memory sys-

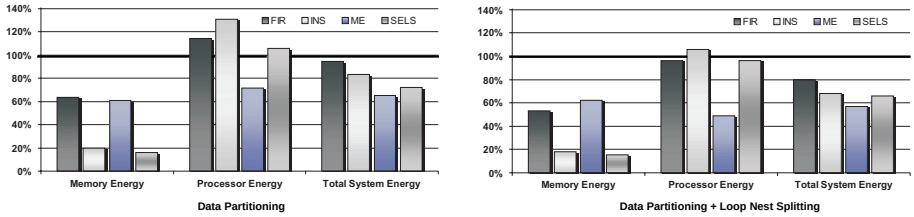

Fig. 5. Relative energy consumption after data partitioning and loop nest splitting

tem (i. e. background main memory and on-chip scratchpad), by the ARM7 processor and by the total system (i. e. processor plus memories) is depicted.

The left diagram of figure 5 clearly shows that data partitioning is a highly effective optimization w. r. t. the memory system. From column Memory Energy, it can be seen that the partitioning of arrays and the placement of parts of arrays onto a scratchpad leads to energy savings between 36.7% (FIR) and 84.2% (SELS). Due to the impaired control flow after data partitioning, the energy consumed by the ARM7 processor generally increases when compared to the original code version. Column Processor Energy shows additional energy consumptions between 6% (SELS) and 30.8% (INS). In the case of the ME benchmark, an energy reduction of 28.6% was measured. This is due to the fact that ME is very data-intensive and needs to access memory very frequently. Since the ARM7 CPU accesses an on-chip memory much faster than the main memory, the processor does not execute as much energy consuming wait states as before data partitioning. For the entire system (column System Energy), the techniques described in section 4 lead to total energy savings between 5.7% (FIR) and 34.7% (ME) with an average improvement of 21.3%.

The right diagram of figure 5 illustrates the relative energy consumption of the benchmarks after combined data partitioning and loop nest splitting. As can be seen by comparing the columns Memory Energy of both charts of figure 5, loop nest splitting conserves the energy savings for the memory system achieved by data partitioning. In the case of the FIR benchmark, additional savings of memory energy by 9.9% were measured. The notably less *if*-statement executions for this benchmark imply less instruction fetches from the memories leading to this result. Column Processor Energy of figure 5 clearly shows that the techniques presented in chapter 5 are able to eliminate the penalties introduced by data partitioning completely. After loop nest splitting, the energy consumption of the ARM7 for the FIR and SELS benchmarks is better than the original unoptimized code. In the case of the INS benchmark, the ARM7 consumes only 5.8% more energy than before any optimization. But also in this case, loop nest splitting has proven to be highly effective, since it reduces the energy dissipation by 25%. For the ME benchmark, loop nest splitting leads to an energy reduction for the ARM7 processor of 22.3% compared to the measurements immediately after data partitioning. Column System Energy illustrates the total savings achieved by the methodology proposed in this paper. It can be seen that the combined energy dissipation of the ARM7 and its memories drops between 20.3% (FIR) and 43.3% (ME) with an average saving of 32.3% compared to the unoptimized benchmarks.

With respect to the runtimes of the benchmarks, the combination of data parti-

tioning and loop nest splitting also is quite beneficial. Figure 6 illustrates the relative runtimes of the benchmarks after each optimization step. Again, the 100% base line denotes the runtime of the original benchmark versions before any optimization. Due to the fact that *if*-statements are inserted in the code of the benchmarks during data partitioning (compare section 4), the execution times of almost all benchmarks increase by 8.7% (SELS) up to 36.7% (INS). Only in the case of the data-intensive ME benchmark, a speed-up of 28.2% was measured which is due to the reduction of wait states as explained above. Loop nest splitting is able to eliminate the negative effects of data partitioning nearly totally. Compared to the runtimes after data partitioning, speed-ups between 9.3% (SELS) and 31.1% (ME) were measured. In total, we are able to improve the runtimes of two benchmarks slightly (FIR: 2.5%, SELS: 1.5%) after the application of both optimizations while simultaneously achieving high gains w. r. t. energy dissipation. For the INS benchmark, a moderate total runtime degradation of 9.7% was still measured after loop nest splitting, whereas the ME benchmark was accelerated by 50.6%.

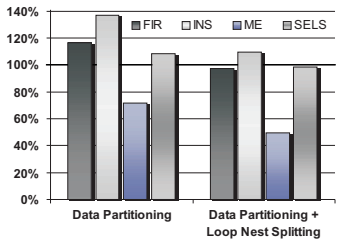

Fig. 6. Relative runtimes after data partitioning and loop nest splitting

Finally, the influence of varying scratchpad sizes on energy consumption and runtimes is depicted in figure 7. The left diagram of this figure illustrates the total energy consumption of all code versions of the SELS benchmark for eleven different scratchpad sizes. It is not surprising that the original unoptimized benchmark consumes the same high amount of energy for all scratchpad sizes. This is due to the fact that no data can be placed onto the scratchpad memory at all due to the large size of the occurring arrays. In contrast, data partitioning is effective in energy consumption minimization already for very small memory sizes. In the case of a 256 bytes memory, only negligible improvements were measured which are not visible due to the resolution of figure 7. But already for 512 bytes, visible improvements were observed. With larger scratchpad sizes, data partitioning achieves higher gains due to the fact that less costly accesses to the main memory are performed. This way, a monotonically decreasing curve has been obtained. The same holds for loop nest splitting applied after data partitioning. Here, loop nest splitting is able to reduce the energy consumption considerably for scratchpads larger than 600 bytes. Again, a monotonic regression can be observed clearly demonstrating the stability of our combined optimization methodology.

With respect to the runtimes of the SELS benchmark (cf. right diagram of figure 7), a similar behavior of combined data partitioning and loop nest splitting for various scratchpad sizes has been measured. Again, the unoptimized benchmark requires constant execution times for all scratchpad sizes. Starting from considerably increased runtimes for small scratchpads, the overhead of data partitioning gets smaller the larger the on-chip memory becomes. This behavior is due to the high latencies imposed by main memory accesses which are minimized most effectively by data partitioning for larger scratchpads. In contrast, the benefits of loop nest splitting are already visible for scratchpad sizes starting from 512 bytes. From this point on, the impact of loop nest splitting on the performance of the benchmark becomes larger as the speed-ups

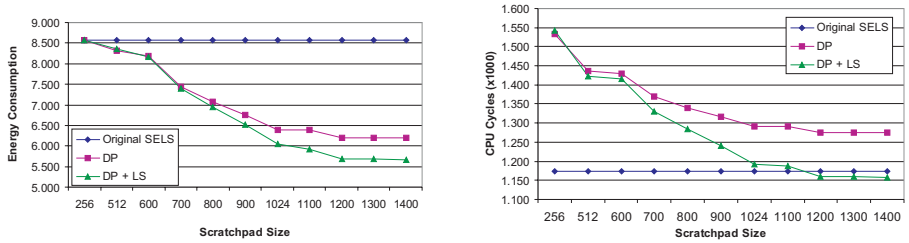

Fig. 7. Energy and performance comparison (Selection Sort)

increase. For a scratchpad size of 1,024 bytes, a runtime nearly equal to the one of the original code has been obtained. For 1,200 bytes, the code generated by the proposed optimization methodology is faster than the original code so that loop nest splitting is able to over-compensate the overheads of data partitioning.

7 Conclusions

This paper presents a new approach for energy dissipation minimization of embedded software forming a homogeneous framework for low power code generation. In a first step, a data layout optimization is performed by partitioning large arrays into smaller pieces which can be put on low-energy scratchpad memories. Motivated by the introduction of a severe control flow overhead during this step, we propose to apply a control flow optimization step afterwards. Loop nest splitting has proven to be highly effective in generating a very regular control flow in the hot-spots of applications making it suitable to eliminate the negative effects of array partitioning.

Besides this entirely new combination of optimizations, the second major contribution of this paper is a significant extension of the analysis algorithms for loop nest splitting. Using these extended techniques, tight restrictions of the original algorithms can be relaxed so that more general classes of applications are transformed. Without these extensions, the optimization of three benchmarks analyzed in this work would have been impossible justifying the efforts spent on the loop nest splitting algorithms.

The results presented in this paper demonstrate that our combined optimization methodology is highly beneficial. The partitioning of arrays and placement of data onto a scratchpad leads to average reductions of energy dissipation of 21.3% for an actual ARM7 based system. In contrast, the runtimes of the benchmarks increase in almost all cases due to the control flow overhead. Loop nest splitting removes this overhead leading to improved runtimes for all benchmarks. Furthermore, loop nest splitting achieves an additional reduction of energy consumption. In total, the successive application of data partitioning and loop nest splitting as proposed in this paper leads to energy savings of up to 43.3% with an average gain of 32.3%.

Since all techniques presented in this paper are implemented such that the optimizations are performed at the level of C source codes before any assembly code generation for the ARM7 processor, our framework is inherently portable to other embedded processors. For this purpose, basically only an accurate energy model needs to be provided.

References

1. Wahlström, J.: Energy Storage Technology for Electric and Hybrid Vehicles – Matching Technology to Design Requirements. KFB Kommunikationsforskningsberedningen, Stockholm, Sweden (1999)
2. Stan, M.R., Burleson, W.P.: Bus-invert coding for low-power i/o. IEEE Transactions on VLSI Systems **3** (1995)
3. Wuytack, S., Catthoor, F., Nachtergaele, L., et al.: Power exploration for data dominated video applications. In: Proc. of ISLPED, Monterey (1996)
4. Marwedel, P., Wehmeyer, L., Verma, M., Steinke, S., Helmig, U.: Fast, predictable and low energy memory references through architecture-aware compilation. In: Proc. of ASP-DAC, Yokohama (2004)
5. Kandemir, M., Ramanujam, J., Irwin, M.J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: Proc. of DAC, Las Vegas (2001)
6. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In: Proc. of CODES, Estes Park (2002)
7. Verma, M., Steinke, S., Marwedel, P.: Data partitioning for maximal scratchpad usage. In: Proc. of ASP-DAC, Kitakyushu (2003)
8. Falk, H., Marwedel, P.: Control flow driven splitting of loop nests at the source code level. In: Proc. of DATE, Munich (2003)
9. Falk, H., Marwedel, P., Catthoor, F.: Chapter 17. In: Control Flow driven Splitting of Loop Nests at the Source Code Level. Volume Embedded Software for SOC. Kluwer Academic Publishers, Boston (2003) 215–229
10. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys **26** (1994)
11. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
12. Fraboulet, A., Huard, G., Mignotte, A.: Loop alignment for memory accesses optimization. In: Proc. of ISSS, San Jose (1999)
13. Kandemir, M.: A compiler-based approach for improving intra-iteration data reuse. In: Proc. of DATE, Paris (2002)
14. Loechner, V., Meister, B., Clauss, P.: Precise data locality optimization of nested loops. The Journal of Supercomputing **21** (2002) 37–76
15. Panda, P.R., Dutt, N., Nicolau, A.: Memory Issues in Embedded Systems-On-Chip. Kluwer Academic Publishers, Massachusetts (1999)
16. Steinke, S., Wehmeyer, L., Lee, B.S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Proc. of DATE, Paris (2002)
17. Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P.G., van Achteren, T., Omnes, T.: Data Access and Storage Management for Embedded Programmable Processors. Kluwer Academic Publishers, Massachusetts (2002)
18. Wilde, D.K.: A library for doing polyhedral operations. Technical Report 785, IRISA Rennes, France (1993)
19. Bäck, T.: Evolutionary Algorithms in Theory and Practice. Oxford University Press (1996)
20. Steinke, S., Wehmeyer, L., et al.: The *encc* energy aware c compiler homepage. http://ls12-www.cs.uni-dortmund.de/research/encc/ (2002)
21. Steinke, S., Knauer, M., Wehmeyer, L., Marwedel, P.: An accurate and fine grain instruction-level energy model supporting software optimizations. In: Proc. of PATMOS, Yverdon-Les-Bains (2001)