Array Based Structure Loop Transformations For Cache Miss Reduction

Marian Stanca, Henk Corporaal, Sorin Cotofana, and Stamatis Vassiliadis
Electrical Engineering Department
Delft University of Technology, The Netherlands
{ akela, heco, sorin, stamatis}@cardit.et.tudelft.nl

Abstract

This paper is concerned with lowering the power consumption of an algorithm described in a high level language, ran on a processor. We are introducing an automated framework for array based loop transformations. A cache miss ratio power estimator is the main ingredient for the statically driven steering mechanism of transforming the algorithm. The experiments have shown decreases in cache miss rate up to 35%, and similar figures for the estimated energy required by the computation.

1. Introduction

Embedded systems can be characterized as, at least, mass produced elements of a larger system providing a dedicated, possibly time constrained, service to that system. Additionally, they have a priori known job characteristics, a small time to market window, and, not in the last place, they are very much cost sensitive. Until recently, the cost function has been "area efficient design respecting performance and design time". Currently, both performance and power dissipation have been added to the embedded system performance metric. It is commonly agreed that performance and low power issues can be addressed through optimizations at all design hierarchy levels, e.g., implementation, device, circuit, logic, architecture, algorithm, and system. In this paper we address the performance and power optimization issue at high level of abstraction. We assume that the application is described in a high level language and is meant to be executed on an architecture that includes a cache memory. Our underlying assumption is that reducing the cache miss ratio will be beneficial to both power consumption and performance. The underlining idea of our approach is to statically transform the application description (algorithm), targeting arrays within loops, and preserving the semantics. The transformation has as final goal the increase of the cache hit ratio. To evaluate the effectiveness of loop transformations in reducing power consumption we propose a framework which is based on an automatic transformation tool. Our framework includes a library of loop transformations, a cache miss rate based power estimator, and a transformation engine that is performing the steering mechanism.

The presentation is organized as follows. Section 2 provides some background informations on loop transformations. Section 3 describes the proposed framework for power reduction transformations. Section 4 consists of experiments and comparations and Section 5 presents some conclusions and intended future work.

2. Loop Transformation

A loop transformation can be applied to a given description of an algorithm as a program, P_{IN}, if a[some] loop[s], identified as candidates according to a given criterion, is[are] transformed in other loop[s]. The resulting program, P_{OUT} has to have the same functionality as the initial program. This is achieved with validity verifications for loop transformations. Loop transformations can operate on some of the elements of the loop set, L, considering as one loop a multiple nested loop. $card(L)$ is the number of loops present in a program. The iterator space is the topological reunion of the iterator domains [1], $\mathcal{I} = \left[{}^{T}i_1, \ldots, {}^{T}i_{dim(\mathcal{I})}\right]$, and $dim(\mathcal{I})$ is the depth of the loop nests. The effects of loop transformations may vary from affecting only the loop body to changing the iterator space, the number of loops, and the depth of the loop nests.

Loop transformations can be employed to decrease the traffic between the cache and off-chip memory or to increase the efficiency of the cache. Traditionally, the issue of loop transformations has been addressed either with a similar, power consumption related, goal [7] or with a different one, such as restructuring of a possibly sequential program to improve execution efficiency on parallel machines [9]. In [10] a technique that involves global data transformations and local loop transformations in order to

[1] The formalism allows as worst case, linear dependence of the iterators, $i_j = \sum_{k=0}^{j-1} \alpha_k i_k,\ j = \overline{2, dim(\mathcal{I})}$.

minimize overhead on distributed shared machines is introduced. The overhead is given by the need to have certain (expensive) temporary arrays copied and, as a result, communication and synchronization problems occur. This technique implies an algebraic transformation framework that has not fully investigated the mathematical properties of the transformation representation. Of those, the most important are the necessary validity tests for all the transformations. Verified by practical implementation in multimedia [1, 2], the semi-automated tools that are helping the design process, in yet another approach, seem to answer all the questions arose by the other approaches. The formalism has better coverage: non-rectangular iterator space, multiple loop nests, validity tests, and, especially, estimated complexity for most of the problems. The most important characteristic of this approach is that power consumption is not an independent metric that can be optimized, but power savings can be obtained via memory's area reduction [7].

An analysis of these different approaches suggests that although mathematical models for formal description of loop transformations and possible transformations have been introduced, the mechanism that is steering the process according to a criterion is either missing or far from being formal. For the purpose of power aware loop transformations the techniques mentioned are not sufficient because either a relevant to power consumption cost function may not be introduced or the existing cost function is not introducing power consumption as an independent metric. Our approach is targeting a high level power aware loop transformation engine. Because of the power consumption term none of the previously mentioned approaches can be employed. Due to the fact that we want to keep account of the design time term into the embedded system cost function, we have decided to limit our research to statical analysis.

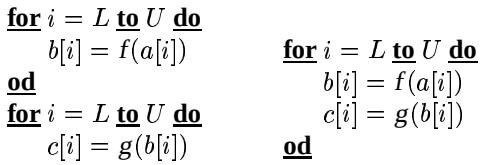

Figure 1. Loop Fusion

Loop transformations that reduce miss ratio, as our experiments suggest, can be classified into the following categories [2]:

- Inherently miss ratio reduction LT: loop fusion, loop interchange, loop tiling.
- Jointly applied LT: node splitting, loop fission, access normalization, wavefronting.
- Miss ratio reduction LT with special conditions: loop reversal, internalization.

This classification is not exhaustive as new types of applications may require new transformations to be added to every category. We will describe further the inherently miss ratio reduction loop transformations.

Although it is roughly depending on the statements involved in the loop nests the most power saving loop transformation is loop fusion. Loop fusion is decreasing the traffic between cache and external memory.

Definition 1 *Loop fusion is a loop transformation in which two different loops, having the same iterator space and the statements contained in the loops in flow dependence are transformed in a single loop. In this case* $dim(\mathcal{I}') = dim(\mathcal{I})$ *and* $card(L') = card(L) - 1$.

A loop fusion is depicted in Figure 1. If the variable that insures the flow dependence, i.e., $b[i]$ is alive after the second loop nest the power saving is given by the difference between an external memory reference and a register reference, saving $n = U - L + 1$ memory loads. Otherwise, $b[i]$ can be removed completely and replaced with a temporary register. This saves an extra n store operations. It has to be noted that for loops with unrelated statements (from the flow dependence point of view) fusion can still be performed. The gain in such a situation is much smaller, namely the one given by the creation and computation needed for the register that keeps the iterator. Very specific conditions can be found, e.g., a cache with the associativity of one and references in the same iteration at two different memory locations separated by a distance larger than the cache block size, due to which loop fusion applied to loops with unrelated statements can worsen the power consumption. In these cases loop tiling can be performed or the organization of the cache can be changed.

Another class of loop transformations is loop permutation, as exemplified in Figure 2. The gain for this class is caused by a better cache usage through improved data locality. Loop permutation is applied according to the way in which multidimensional arrays are stored in memory [3]. E.g., in C, the arrays are stored in memory by rows, so, for a better cache usage, successive iteration memory references have to stay in the same row. This is equivalent with the fact that the iterator space has its first iterator in the inner loop.

Definition 2 *Loop permutation is a loop transformation applied on loops with* $dim(\mathcal{I}) \geq 2$*, in which some of the loop nests are changed in respect to the lexicographical order. In this case* $dim(\mathcal{I}') = dim(\mathcal{I})$.

Loop permutation is legal if, and only if, the mapped distance vectors are lexicographically positive. This condition

[2] A different classification, serving a different purpose can be found in [8].

[3] The data layout is the most important factor for loop permutation.

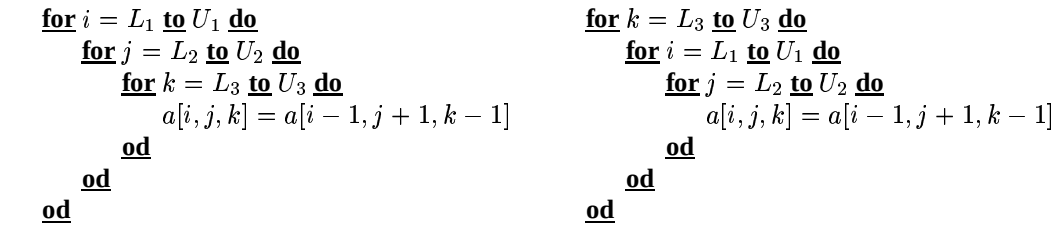

Figure 2. Loop Permutation for $dim(\mathcal{I}) = 3$

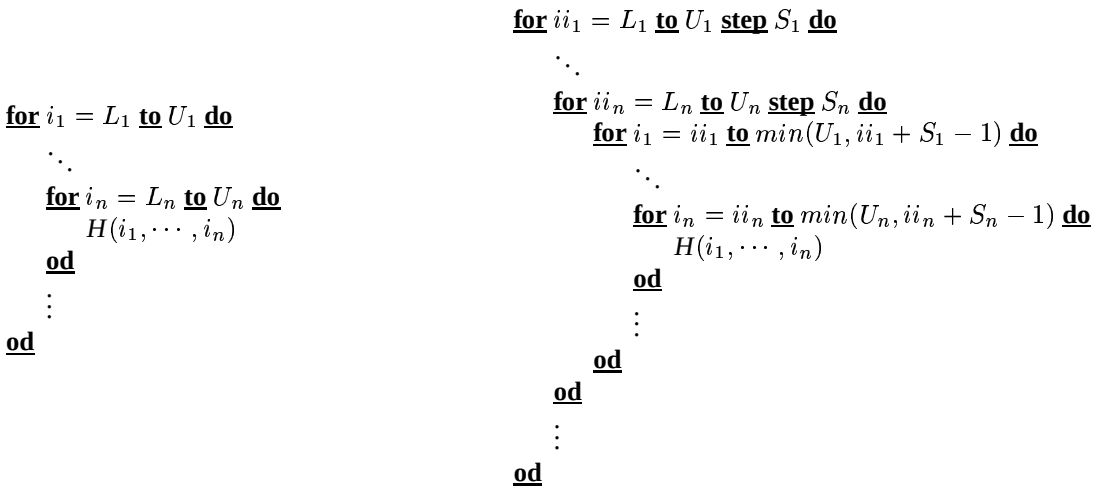

Figure 3. Loop Tiling

can be expressed formally with the following characterization lemma.

Lemma 1 *Loop permutation is valid if and only if the number of cycles in the decomposition of the dependence permutation, seen as the topological reunion of the dependence direction vector and dependence distance vector, is even.*

Loop permutation is an unimodular transformation $\mathbf{T} = \{t_{i,j} \mid t_{i,\overline{n-i}} = \pm 1, \text{ for } i = \overline{1,n}, \text{ and } t_{i,j} = 0 \text{ for } i \neq n - j,\ i, j = \overline{1,n}\}, \|\mathbf{T}\| = 1$. A loop permutation can be applied for $dim(\mathcal{I}) = 2$, in which case the transformation is called loop interchange. The transformation matrix is now $\mathbf{T} = \{t_{1,1} = t_{2,2} = 0, t_{1,2} = t_{2,1} = 1\}$, so $\|\mathbf{T}\| = 1$. Loop interchange is valid if the vectorial representation of dependences has, initially, positive value for the term representing the inner loop. The same type of power related gain appears, i.e. more efficient cache usage.

Another class of loop transformations is tiling, as depicted in Figure 3. The gain for this class cannot be predicted in a formal manner, but it consists of better cache performance, through increased data locality.

Definition 3 *Assuming $1 \leq k \leq n$ the number of dimension tiled, tiling is a loop transformation in which a nested loop with $dim(\mathcal{I}) = n$ is transformed in a nested loop with $dim(\mathcal{I}') = n + k$ with a smaller and less complex iterator space in the inner loop[s].*

A necessary condition for this type of transformation to be applied is that the loop nests that have to be tiled are fully permutable. Therefore, there have to be only positive terms in the matrix representation of the dependences, on the position corresponding to the loop nests that have to be tiled. The dimension of the tiles must be established so that the dimension of the new iterator space is related with the dimension of the cache blocks[4]. Figure 3 depicts the most general case in which all the dimensions are tiled. Such a general type of tiling is possible but not always needed. Usually 1 or 2 dimensions are tiled for better taking advantage of the cache organization.

Loop transformations with special conditions are those transformations that are speculating architectural details. In some cases and under certain conditions some other transformations may offer savings, e.g., if an address bus generator contains a counter, then is beneficial to have the iterator space transformed -with access normalization or loop reversal, for instance- in order to take advantage of it. It is also conceivable that transformations like scaling and rotation may be convenient when the changed iterator space is facilitating increased spatial and temporal locality. The cost, in terms of power consumption, of type changing for the iterating variable and subsequent operations cannot be compared with the power consumption reduction due to better cache usage, at this level of abstraction, therefore we

[4]The cache organization is the most important factor for loop tiling.

neglect such cases.

3. LT Based Design Framework

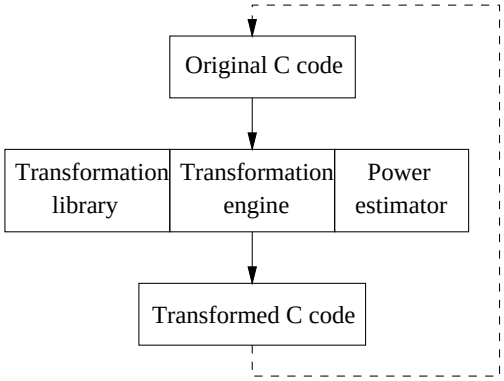

Figure 4. Power Reduction Framework

Our design framework is illustrated in Figure 4. The result is that the original description in C of an algorithm is transformed in a less power consuming one. The framework consists of a power estimator, a loop transformations library, and a transformation engine. The library contains an enumeration of possible loop transformations, with applying conditions. The transformation engine is based on a steering mechanism, in order to selectively apply the needed transformation for power consumption reduction. The steering mechanism is taking the decision of applying a given transformation. Such decisions are taken through a less than exact ruled based method. The rules are inferred from experiments and are a reflection of the type of programs that are targeted for power consumption reduction. The most important characteristic of this framework is that due to the steering mechanism, at a given time, the transformation to be applied is only a function of the current state of the code. Rules for loop transformations may look as described further.

1. If two consecutive loops have the same iterator space and there is flow dependence between the arrays involved in the loops statements then loop fusion may be applied; if there are no dependencies between the statements then loop fusion may still be applied with lesser improvement.

2. If a loop has it's statements addressing arrays in a non-lexicographical manner and validity check holds then loop interchange may be applied.

3. If a loop has it's statements addressing arrays that are larger than the cache line the tiling may be applied.

4. If a loop is reading arrays with different references to the iterator space node splitting may be applied.

5. If consecutive loop nests have their some sub- statements in flow dependence and loop fusion can not be applied, and the arrays are larger than the cache lines, wavefront may be applied.

This is not an exhaustive rule enumeration, as studying different types of benchmarks may extend both the rule list and the transformations list. For further improvement, exploiting weaker conditions than those described, may be possible. For example, if two consecutive loop nests have their statements in flow dependence and the iterator space is different, enabling transformations may still be applied in order to obtain an identical iterator space. Many other types of loop transformations, e.g. loop unrolling, are increasing the cache hit ratio while not modifying the power consumed by the application. It has to be mentioned that our framework can be extended in applying any type of loop transformation with no repercussions to the power consumption.

The design scenario assumes that an input program, i.e., the application, is given, in which patterns of loops are searched. If such a pattern is found, the necessary rule triggering conditions are checked. In case more than a rule may be applied at a given time, the rule with the biggest priority is applied. Assigning different priorities uncovered the fact that some classes of loop transformations are orthogonal, being insensitive to the order in which were applied. The process is iterative until no further power consumption reduction transformations, at this level of abstraction, can be performed to the code.

Comparing manual inspection with the automated one suggested that the number of steps in which the minimum of power consumption can be achieved is minimal. Due to the fact that the benchmarks were either “nicely written” or the result of an automated transforming tool, the comparation suggested that the number of loops that can be transformed, with respect to power consumption, automatically is very close to number of loops that can be transformed manually. Such a good performance, in respect to identify possible transformations, may be explained through the fact that the rule based method is miming the design decisions taken by the human expert. In the next section, in an attempt to evaluate the performance, we will discuss the experimental results, when our framework is applied.

4 Experiments and Comparations

In order to verify the effectiveness of our approach a set of experiments has been performed. Two examples of loop transformations are presented. These examples were chosen to exemplify that power savings are not obtained with only power saving loop transformations. Although at some point in the design space the power consumption resulted from a transformation is at maximum, the final result

```
for k = 0 to 1 do
    for j = M + 1 to 1 step −1 do
        for i = M + 1 to 1 step −1 do
            EL[2*i − 1, 2*j, k] = (EL[2*i, 2*j, k] + EL[2*i − 2, 2*j, k])/2
            EL[2*j, 2*i − 1, k] = (EL[2*j, 2*i, k] + EL[2*j, 2*i − 2, k])/2
        od
    od
od

for k = 0 to 1 do
    for j = M + 1 to 1 step −1 do
        for i = M + 1 to 1 step −1 do
            EL[2*i − 1, 2*j, k] = (EL[2*i, 2*j, k] + EL[2*i − 2, 2*j, k])/2
            EL[2*i, 2*j − 1, k] = (EL[2*i, 2*j, k] + EL[2*i, 2*j − 2, k])/2
        od
    od
od
```

Figure 5. Combined Transformation Applied to WANAL

is lowering the power consumption. WANAL is a wave equation solver, part of the RICEPS benchmark suite. In Figure 5 the original and transformed loop, respectively, are presented. In order to get good locality it is necessary to interchange j and i in the second statement but not in the first. To achieve this we can perform loop distribution (valid because there is no flow dependence between sub-statements between iteration), loop interchange in the second loop and, finally, loop fusion. The steering mechanism works as follows:

1. Assuming row major allocation, the second sub-statement is not optimal (verifiable with the dependence distance matrix), but the first is.
2. There is no flow dependence between the two sub-statements (verifiable with dependence direction) so we can do loop fission.
3. Loop fission (now valid)
4. Loop interchange (verifiable with the dependence vector) - sub-statement 2 is now optimal.
5. Further improvement can be gained with loop fusion (valid because of the same iterator space and no flow dependence between sub-statements).

It has to be noted that steps 1 to 4 are driven by the fact that the steering mechanism has been triggered. While these steps are equivalent to one traversal of the itinerary described in Figure 1, so is step 5 alone. With row major allocation, the second statement in the initial nested loop is addressing memory locations that are in different rows. For the case in which the dimension of the cache is smaller than $2(2M+3)^2$ then a cache miss occurs in every iteration. This inconvenience is eliminated in the transformed version of the loop [8]. Even if the cache dimension is bigger than the array dimension, then, for a block-buffering scheme, the gain of increased spatial locality is evident; also the transformed version offers the advantage of reading the same operands twice in a short period of time, ensuring good temporal locality too. Following, a study case is presented. Let us assume that the dimension of the cache lines is l and that there is no border effect. Expressing the lack of border effect is done with $\frac{M+1}{l} \in \mathrm{N}$. Replacing the previous factor with $\left[\frac{M+1}{l}\right] + 1$ in all the equations does treating cases where this is not true. For the original loop nest, for the first statement, a cache read miss appears due to cold start every time decreases. There are no cache write misses. For the second statement, two cache read misses and one write miss appears every iteration, except for the two cases in which the cache line is already accessed within the same iteration. Both LRU and min [3] policy for replacing cache lines when the cache is filled ensure that the line needed for the first statement during the next iteration would not be purged, avoiding a supplementary miss. In order to be able to show qualitative results without going into too much low level detailing, we will consider that the miss penalty is the same for both the cases in which the cache is already filled or it is not filled yet. Doing so we discard the effect of bus contention between main memory and on chip memory. The limited cache dimension effect can be taken into account, from the energy consumption point of view, by considering different miss penalties. This also allows greater flexibility due to the fact that differently organized caches can be studied. All the effects are considered in Equation (1), with $P_{R,W}$ being power penalty

```
for j = 2 to n do
    for i = 2 to n do
        a[i,j] = a[i-1,j] + a[i,j-1] + a[i-1,j-1]
        +b[i-1,j] + b[i,j-1] + b[i-1,j-1]
        +c[i-1,j] + c[i,j-1] + c[i-1,j-1]
    od
od

for i = 2 to n do
    a[i,2] = a[i,1] + a[i-1,1] + b[i,1] + b[i-1,1] + c[i,1] + c[i-1,1]
od
for j = 2 to n-1 do
    for i = 2 to n do
        a[i,j] = a[i,j] + a[i-1,j] + b[i-1,j] + c[i-1,j]
        a[i,j+1] = a[i,j] + a[i-1,j] + b[i,j] + b[i-1,j] + c[i,j] + c[i-1,j]
    od
od
for i = 2 to n do
    a[i,n] = a[i,n] + a[i-1,n] + b[i-1,n] + c[i-1,n]
od
```

Figure 6. Combined Transformation Applied to SLIA

for a cache miss.

$$PP_{WN_I} = 2(M+1)\left(\frac{M+1}{l}P_R + 2(M-1)P_R + MP_W\right) \quad (1)$$

For the modified loop nest, for both statements, two cache misses at reading appears due to cold start and every time $\left[\frac{i}{l}\right]$ decreases. Supplementary cache misses are avoided using LRU or min replacement technique. The result is presented in Equation (2)

$$PP_{WN_F} = 2(M+1)\left(\frac{M+1}{l}\right)(2P_R + P_W) \quad (2)$$

In Equations (1), (2) if $P_R = P_W = 1$ then the right term becomes number of cache misses. With this assumption, the equations have been verified with Simple Scalar [5] for some numerical examples. This normalization allows the illustration of a very interesting effect: the local decrease of power consumption is given only by $\frac{PP_{WN_I}}{PP_{WN_F}} \sim l$.

SLIA is a synthetic loop presented in Figure 6, in the original and the transformed version. LT address data alignment constraints, the number of operations needed, and the possibility of pipelining. The steering mechanism performs in a similar manner. The study case is resembling the WANAL study case.. The effects are as written in Equations (3),(4).

$$PP_{SL_I} = \frac{n-1}{l}6(n-1)P_R \quad (3)$$

$$PP_{SL_F} = \frac{n-1}{l}(3(n+1)P_R + nP_W) \quad (4)$$

Simplifying with $P_R = P_W = 1$ offers the conclusion: the local decrease of power consumption is given only by $\frac{PP_{SL_I}}{PP_{SL_F}} \sim 1.5$. Taking advantage better of the cache can be obtained if the modified version of the loop is scheduled in a wavefront. A more complicated description would illustrate further reduction of the cache miss ratio.

To asses the performance and have a non-local image of the decreased cache miss ratio, and consequently power consumed, our design framework is applied on two sets of benchmarks. The Simple Scalar[5] tool set contains a number of simulators that may be used to find profiling informations about programs and their characteristics when ran on various architectures. Instrumenting the executables in order to determine the traces of memory references was performed a retargeted gcc compiler, provided with the tool set. All the experiments assume 8KBytes data cache size, an associativity of one, and a line size of 8Bytes.

DSP type of benchmarks targeting image handling (edge, flatten, smooth) allow all enumerated types of loop transformations to be applied. The same phenomenon may be observed for mathematical related programs [6], e.g., equation solvers, which are usually manipulating multi-dimensional matrixes. We have chosen the data set to be much bigger than the cache size and a small line size, in order to observe the relevance of each type of loop transformation that we have applied. Therefore, the miss rate for the modified applications may still be improved. The instruction count of the retargeted modified code is decreasing as a consequence of applying loop transformations. The results are depicted in Table 1.

name	miss rate initial	miss rate final	instr. count initial	instr. count final
edge	49.78	37.12	101,376	89,739
flatten	48.19	41.77	110,116	96,127
smooth	46.20	19.63	90,641	83,282
matmul	44.12	23.76	898,253	816,980
nasi	76.37	41.52	487,367,565	474,877,114
nmc	62.91	37.48	2,213,146	2,181,533
tper	65.20	44.73	3,336,268	3,331,151
ccm	60.25	36.32	1,656,187	1,575,131
hydro	66.43	38.04	2,598,311	2,368,551
wave	58.56	29.81	1,109,423	996,394

Table 1. DSP and Matrix Manipulation Benchmarks

Separate experiments with digital circuitry synthesis tools, targeting a 0.5 micron technology, have provided us with the means of linking the cache miss ratio, along with the instruction count (as only parameters available at high level of abstraction) to the power consumption. Test inputs have been applied in order to simulate application activity on an architecture. With profiling information, an underlying assumption that energy saving scheme are available for idle time, and suppositions on instruction latency and functional block through-output, we were able to average $10^{-15}J$ per instruction. Moreover $10^{-14}J$ per bus transaction, $5x10^{-9}J$ per cache read/write, and $8x10^{-7}J$ per main memory access. It has to be noted that a separate set of experiments with a different implementation library offered similar figures for energy per instruction and energy per bus transaction. And very different results for memory type of accesses. The power estimator showed energy savings in the range of 9-12% for image processing applications and 19-27% for matrix manipulation benchmarks.

5. Conclusions and Future Work

A novel technique has been introduced in order to build a power aware, at high level of abstraction, loop transformations engine, employed statically on array structures. Although in an opened to improvement stage, the principle on which the engine is based already suggested important conclusions for DSP and matrix manipulation benchmarks.

Directions for future work will include, at least, the following issues. Studying new types of benchmarks will suggest new jointly applied power savings transformations and new rules to be included in the steering mechanism. The power estimator can further detail the penalties as a function of architectural characteristics, opening the possibility for our approach to function, hierarchically, on different levels of abstraction. When necessary and with the reserve arisen by the increased cost in design time, dynamic analysis will be employed to extend the class of benchmarks on which loop transformations can be applied. Dynamic analysis is characterized by the lack of formal verification of transformation legality due to impossibility of finding all the dependences between the memory references, but may prove beneficial when application relying heavily on pointers are to be optimized.

References

[1] F. Balasa, F. Catthoor, and H. D. Man. Practical solutions for counting scalars and dependences in ATOMIUM - a memory management system for multi-dimensional signal processing. *IEEE Transactions on Computer-aided Design*, 16(2):133–145, 1997.

[2] F. Balasa, F. H. M. Franssen, F. V. M. Catthoor, and H. J. De Man. Transformation of nested loops with modulo indexing to affine recurrences. *Parallel Processing Letters*, 4(3):271–280, Sept. 1994.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 78–89, New York, May 22–24 1006. ACM Press.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[6] J. Castellanos and F. Carmouche. Riceps benchmarks.

[7] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergale, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration Of Memory Organization For Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[8] D. Kulkarni and M. Stumm. Loop and data transformations: A tutorial. Internal document, a tutorial guide., 1993.

[9] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. In *International Conference on Parallel Processing, Vol.2: Software*, pages 19–28, Boca Raton, USA, Aug. 1995. CRC Press.

[10] M. F. P. O'Boyle and P. M. W. Knijnenburg. Integrating loop and data transformations for global optimization. *PACT*, 1998.