# Minimizing Memory Access By Improving Register Usage Through High-level Transformations

Shan Li
School of Computer Engineering
Nanyang Technological University
Nanyang Avenue, SINGAPORE 639798
Email: p144102711@ntu.edu.sg

Mohammed Javed Absar
STMicroelectronics Asia Pacific Pte. Ltd.
20 Science Park Road, SINGAPORE 117674
Email: javed@imec.be

***Abstract*—Multimedia signal processing software typically have to process large amounts of data. The algorithms often involve the handling of data arrays in the form of nested loops. Experiments show that for this kind of applications data transfer (memory access) operations consume much more power than data-path operations. Our objective is to reduce memory access related power consumption by reducing the number of data transfers between processor and memory, or between a higher (closer to processor) level of memory and a memory at a lower level using source program transformation. In this paper, we propose a source-to-source transformation method to improve register usage for multi-dimensional arrays in nested loops. Significant improvement is demonstrated through some benchmark programs.**

## I. Introduction

Power consumption has become an increasingly important cost factor in embedded system designs. One of the reasons is the proliferation of battery powered portable hand-held devices. Moreover, high power dissipation also means costly packaging and cooling requirements and lower reliability. Consequently, power-efficiency is a critical concern when designing an embedded system.

Our specific target of power reduction is multimedia applications. This kind of applications is data-intensive and access large amounts of data. Power consumption of these applications is largely contributed by data transfer and memory access operations [8], [9], [14], [10], [11]. In contrast, data-path operations consume much less power. Hence, reducing data memory access will lead to a reduction of power consumed by the application.

Generally, in a program data are stored as arrays and are often manipulated inside (nested) loops. Without any optimization to the source code, redundant access to the arrays often exist and should therefore be minimized. Our approach to this problem is to perform high-level source-to-source transformations so that temporally reused data can be stored in registers instead. However, temporal reuse can happen in various forms. In order to select and perform suitable high-level transformations for different kinds of temporal reuse automatically, a systematic method is required. This paper proposes such a method that improves the register allocation of subscripted array variables under normal compilation conditions, with compilers which use coloring-based register allocators. The major contribution of our work is that multiple-index subscripted variables are explored in our method. In contrast, the previous works [7], [6], [12], [13], [5] restrict array references to be single-index subscripts. This limits the maximum exploitation of temporal reuse.

## II. Multiple-Index Subscripted Variable

Subscripted variable is array reference inside a loop nest (e.g. $x[i][j]$). Multiple-index subscripted variable means that an array reference can have more than one loop index variables in each of its dimensions (e.g. $x[i+j][j]$ and $x[2i-j]$). A general form of

```
for (I1 = b1; I1 < (b1'+1); I1 += t1)
  for (I2 = b2; I2 < (b2'+1); I2 += t2)
      ...
    for (IM = bM; IM < (bM'+1); IM += tM)
      ...x[ f(I) ]...
```

$$\vec{f}(\vec{I}) = \vec{I}H + \vec{c} = [I_1 \quad I_2 \quad \dots \quad I_M] \begin{bmatrix} h_{11} & h_{12} & \dots & h_{1N} \\ h_{21} & h_{22} & \dots & h_{2N} \\ \dots & \dots & \dots & \dots \\ h_{M1} & h_{M2} & \dots & h_{MN} \end{bmatrix} + [c_1 \quad c_2 \quad \dots \quad c_N]$$

or

$$\vec{f}(\vec{I}) = [f_1(\vec{I}) \quad f_2(\vec{I}) \quad \dots \quad f_N(\vec{I})]$$
$$= [I_1h_{11} + I_2h_{21} + \dots + I_Mh_{M1} + c_1 \quad I_1h_{12} + I_2h_{22} + \dots + I_Mh_{M2} + c_2 \quad \dots \quad I_1h_{1N} + I_2h_{2N} + \dots + I_Mh_{MN} + c_N]$$

Fig. 1. General form of multiple-index subscripted variable 'x' in a nest loop

Multiple-index subscripted variable is illustrated in Fig. 1. The array reference $x[\vec{f}(\vec{I})]$ has $N$ dimensions inside a loop nest with a depth of $M$. In each dimension, a subscript expression $f_n(\vec{I})$ $(1 \leq n \leq N)$ can have maximal $M$ loop index variables $I_m$ $(1 \leq m \leq M)$. That is to say, the subscript value of the $nth$ dimension of the array is determined by more than one loop index variables. When it is the case, the array reference $x$ is said to be a multiple-index subscripted variable.

A multiple-index subscripted variable generates temporal reuse if it accesses the same datum in different iterations; this kind of variables is exploitable in improving register allocation. Those that fail to generate temporal reuses will not be considered during register allocation. Whether multiple-index subscripted variable can generate temporal reuse is determined by both its subscript expressions and the boundary values of the loop nest. This is explained in more detail in the section III-A.

Temporal reuse generated by a multiple-index subscripted variable is referred to self-temporal reuse. However, this is different from the self-temporal reuse described in [15]. In [15] the self-temporal reuse of an array reference is generated by invariant loop index variable only, whereas, the self-temporal reuse here is generated by multiple loop indices. Fig. 2 gives an example of the two cases. Both reference $x[i]$ in Fig. 2(a) and reference $x[i+j]$ in Fig. 2(b) generate temporal reuses. $x[i]$ is invariant to loop index $j$. With the same value of $i$, $x[i]$ accesses the same datum throughout the entire loop $j$. In this example, $x[i]$ has 10 instances from $x[0]$ to $x[9]$. Every instance is accessed 10 times which is the boundary value of loop $j$. In contrast, $x[i+j]$ does not have any invariant loop. Its subscript is a function of both loop indices $i$ and $j$. The temporal reuse generated by $x[i+j]$ results from both loop indices $i$ and $j$. For example, $x[i+j]$ accesses the instance $x[1]$ twice for $(i = 1 \ j = 0)$ and

```
for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    ... = x[i] ...
```

(a)

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
    ... = x[i+j] ...
```

(b)

Fig. 2. Examples of two kinds of self-temporal reuse

($i = 0$ $j = 1$). Furthermore, $x[i + j]$ accesses each instance with different number of times. $x[0]$ is accessed once for ($i = j = 0$), whereas, $x[9]$ is accessed 10 times for different value combinations of $i$ and $j$. This example shows that multiple-index subscripted variable generates self-temporal reuse in a different way from loop invariant reference. Therefore, the self-temporal reuse described in this paper complements the original concept in [15]. In a more general case like Fig. 1, the situation will be more complicated. This requires a systematic approach to detect and exploit multiple-index subscripted variable.

## III. Method

Conceptually, the exploitation method is simple. It tries to minimize memory access by replacing the frequently used multiple-index subscripted variables to temporary scalar variables. This data transformation is called scalar replacement [5]. To enable this transformation, an systematic approach is developed. The flow of the overall approach is shown in Fig. 3. The steps in the approach are discussed in the following sub-sections.

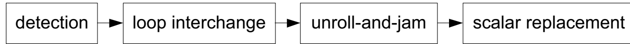


Fig. 3. The flow of the approach exploiting Multiple-index subscripted variable

### A. Detection Process

Given a program, the compiler should find out not only the multiple-index subscripted variables but also their ability of generating temporal reuses. This process starts from the subscript expressions of an array reference. The subscript expressions can be written in the form of a matrix called access matrix, $H$ (defined in Fig. 1).

For a loop invariant reference, the access matrix, $H$, contains one or more rows of zeros, meaning a loop index is not in any subscript expression of the reference. This evidence of temporal reuses is obvious to detect. In contrast, the situation for a more general form of array reference including multiple-index subscripted variable is not that direct. For a general form of array reference, its ability of generating temporal reuses can be found by comparing the rank of its access matrix with the depth of the loop nest it resides. For a multiple-index subscripted variable, the rank of its access matrix is computed by applying Echelon reduction [3]. This rank is used to compare with the loop nest's depth less the number of invariant loops, since a multiple-index subscripted variable can also be a loop invariant reference. Assuming the number of loops invariant to a multiple-index subscripted variable is $p$, the results of the comparison are shown below.

- When $rank(H) = M - p$, there is no temporal reuse is generated by the multiple indices.
- When $rank(H) > M - p$, this situation should not exist in a correctly written program. There is no reuse in this case.
- When $rank(H) < M - p$, the reference might generates temporal reuses with its loop indices. It should be noticed that, these temporal reuses are exploitable only when they occur within the loop bounds $\vec{b}$ and $\vec{b}'$.

When it is found that a reference has $rank(H) < M - p$, the *distance vector* should be computed using Equation 1. Distance vector measures the distance between multiple accesses to a datum in terms of loop iterations. Its principle is that self-temporal reuse occurs when any two instances, $x[\vec{f}(\vec{i})]$ and $x[\vec{f}(\vec{j})]$, of an array reference $x[\vec{f}(\vec{I})]$ access the same datum at different iterations $\vec{i}$ and $\vec{j}$ ($\vec{i} \neq \vec{j}$).

$$\begin{aligned} \because\ & \vec{f}(\vec{i}) = \vec{f}(\vec{j}) \text{ and } \vec{i} \neq \vec{j} \\ & \vec{i}H + \vec{c} = \vec{j}H + \vec{c} \\ & (\vec{i} - \vec{j})H = \vec{0} \\ & \vec{d} = \vec{i} - \vec{j} \\ \therefore\ & \vec{d}H = \vec{0} \end{aligned} \qquad (1)$$

The distance vectors for $x[i]$ in Fig. 2(a) and $x[i+j]$ in Fig. 2(b) are $[0, 1]$ and $[1, -1]$ respectively. From the distance vector of $x[i + j]$, it can be seen that there is a coupling effect between the values of the two loop indices. That is to say, whenever loop index $i$ increases by 1, loop index $j$ should decrease by 1. This coupling effect of the indices generates temporal reuses for exploitation.

### B. Loop Interchange And Related Tests

Loop interchange is a loop transformation that permutes the loops inside a loop nest [2], [3]. It is performs before unroll-and-jam in the approach. Loop interchange itself is not helpful to reveal reuses from multiple loop indices. Instead, it is performed for loop invariant references. Loop invariant reference is exploitable only when its invariant loop is the innermost loop. Although this paper is to exploit multiple-index subscripted variable, the occurrence of loop invariant reference makes loop interchange necessary.

To determine whether loop interchange is necessary, the method does the followings. It finds out all the invariant loops, which are legal to shift to the innermost loop, in a loop nest. Then it selects one of the invariant loops. By assuming that loop as the innermost loop, it computes the amount of exploitable reuses. The invariant loop that generates maximum number of reuses will be chosen to be the innermost loop if it is not yet the innermost. If invariant loop does not exist in the loop nest, no loop interchange and related tests will be performed.

### C. Unroll-And-Jam

Unroll-and-jam is a kind of loop transformation technique that unrolls a loop body several times [6], [13]. It can be used in conjunction with scalar replacement to improve register allocation [5], since it reveals the reuses carried by the loop indices. As mention in section III-A, the distance vector of reuse generated by multiplex-index subscripted variables reveals coupling effects between loop indices. Thus, when applying unroll-and-jam, multiple loops should be unrolled to expose this kind of reuses. Although unrolling multiple loops does not cause any legality problem, jamming the unrolled loop body into the innermost loop is not always legal. For an outer loop, unroll-and-jam is allowed when the outer loop can be interchange to the innermost loop.

The most important problem in unroll-and-jam is to determine the unrolling vector of the loop nest. For loop nest with depth $M$, the

For every loop carrying legal temporal reuses, its unrolling factor $v$ is initialized to the corresponding $t_m$. The minimal ratio $R^{min}$ is initialized to 1. The optimum unrolling factor $v^{opt}$ is initialized to $t_m$.

Assuming there are L loops carrying legal temporal reuses,

```
for m=1, to L
    for v_m = t_m to v_m^max
        compute the number of reuses, Ne.
        compute the number of registers, Nr.
        if (Nr < the number of registers available)
            break;
        endif.
        compute the total number of memory accesses ,Na.
        compute the number of memory access ratio R.
        if (R < R^opt || ((R==R^opt) && ( (v_1 × v_2 ... × v_m ... × v_L) < (v_1^opt × v_2^opt ... × v_m^opt ... × v_L^opt) )))
            R^opt = R.
            Optimal unrolling vector is assigned with the current unrolling vector.
        endif
    endfor
endfor
```

Fig. 4. Iterative algorithm of searching optimal unrolling vector

```
for (i = 0; i < 9; i += 3)
{
    for (j = 0; j < 9; j += 3)
    {
        … = x[i + j] …;
        … = x[i + j + 1] …;
        … = x[i + j + 2] …;
        … = x[i + j + 1] …;
        … = x[i + j + 2] …;
        … = x[i + j + 3] …;
        … = x[i + j + 2] …;
        … = x[i + j + 3] …;
        … = x[i + j + 4] …;
    }
    … = x[i + 9] …;
    … = x[i + 10] …;
    … = x[i + 11] …;
}
for(j = 0; j < 10; j++)
{
    … = x[9+j] …;
}
```

(a) Unroll-and-jam

```
for (i = 0; i < 9; i += 3)
{
    x3 = x[i];
    x4 = x[i + 1];
    for (j = 0; j < 9; j += 3)
    {
        x0 = x3;
        x1 = x4;
        x2 = x[i + j + 2];
        x3 = x[i + j + 3];
        x4 = x[i + j + 4];
        … = x0 …;
        … = x1 …;
        … = x2 …;
        … = x1 …;
        … = x2 …;
        … = x3 …;
        … = x2 …;
        … = x3 …;
        … = x4 …;
    }
    … = x[i + 9] …;
    … = x[i + 10] …;
    … = x[i + 11] …;
}
for(j = 0; j < 10; j++)
{
    … = x[9+j] …;
}
```

(b)scalar replacement

Fig. 5. Scalar replacement in round robin fashion after unroll-and-jam

unrolling vector is defined as $\vec{v} = [v_1\ v_2\ \ldots\ v_m\ \ldots\ v_M]$. With the understandings of the multiple-index subscripted variables, we develop a parameter estimation model to determine the unrolling vector for loop nests containing multiple-index subscripted variables. This estimation model has taken into account the distance vectors of the reuses, the boundaries and the incremental step size of the loop nest, and the number of available registers. In our model the extended code size from unroll-and-jam is not considered, since this is not purpose of this paper. However, further consideration on extended code size should be incorporable into our method without much modification.

*1) Parameters To Estimate:* The optimal unrolling vector not only reveals a significant amount of reduction in memory access but also avoids register spilling. To find such an optimal unrolling vector for a given loop nest, measurements are needed. The measurements take both the effect of scalar replacement and the effect of unroll-and-jam into account. The details are listed below.

- The total number of reuses exposed by unroll-and-jam, $Ne$. Reuse is the 'read' after the first access.
- The number of scalar variables required by scalar replacement, $Nr$. The number of registers required cannot be greater than the number of available registers. This is a condition to prevent register spilling.
- The total number of memory accesses after the optimization, $Na$. This computes the total number of memory accesses after scalar replacement in round robin fashion.
- The ratio of the memory accesses after and before the optimization, $R$. This parameter reflects the effect of optimization. This is the objective to minimize.

*2) Unrolling Vector:* The algorithm for selecting an optimized unrolling vector based on the parameter estimations is presented in Fig. 4. In Fig. 4, the first step is to do some initialization work for the iterative searching of the optimal unrolling vector. It initializes both the temporary unrolling vector (for searching) and the default optimal unrolling vector to the original incremental step size vector $\vec{t}$. During the search, it performs the parameters estimation for every possible unrolling vector. The unrolling vector leading to minimal memory accesses under the register restriction will be chosen. If there is more than one such unrolling vector, the one with smaller code size will win. If no unrolling vector satisfies the register restriction, the original incremental step size is the optimal unrolling vector.

### *D. Scalar Replacement*

Scalar replacement replaces subscripted variables (array references) by temporary scalar variables to effect reuse [5]. Scalar replacement is the last transformation in the method, which is performed after unroll-and-jam. This transformation increases the register usage for most normal compiler with coloring-based register allocators and is the most effective way of reducing memory operands [14]. Register operand also has shorter running times due to elimination of potential stalls and cache misses.

When applying scalar replacement, there are few things to consider. Firstly, legality should always be taken into account. Scalar replacement is performed on variables having true dependence (read-after-write) or input dependence (read-after-read). When the reuse is true dependence, it should be ensure that there is no data update between the 'write' and the 'read' of the datum. When the reuse is input dependence, scalar replacement will not cause any legality problem. For multiple-index subscripted variables, the reuse generated by multiple loop indices is self-temporal reuse, which is always input dependence. Hence, there is no legality problem during scalar replacement. Secondly, when performing scalar replacement after unroll-and-jam, the temporary scalar variables are used in round robin fashion. For example, the code in Fig. 2(b) is optimized and shown in Fig. 5. In Fig. 5(a) unroll-and-jam is performed to expose the temporal reuses generated by the multiple-index subscripted array reference $x[i+j]$. Both loop $i$ and loop $j$ are unrolled by an unrolling factor of 3. The temporal reuses are the references having the same subscripts, which are then reduced by scalar replacement as shown in Fig. 5(b). Scalar replacement is applied in round robin fashion by introducing temporary variables $x0$, $x1$, $x3$ and $x4$. The assignments of $x3$ to $x0$ and $x4$ to $x1$ clear the data in $x3$ and $x4$, thus $x3$ and $x4$ can be loaded with new values. To enable the round robin fashion, $x3$ and $x4$ are initialized in the outer loop. In the outer loop, there is no data passing between $x3$ and $x4$, since the distance of two array references carried by loop $i$ is 1, which is less than the incremental step size 3. Hence, the round robin fashion is not applied in the outer loop. This example shows that scalar replacement in the round robin fashion greatly reduces the number of array references. This is because the data with lifetime across iterations are passed through scalar variables.

TABLE I
BENCHMARK FROM HLSYNTH95/MEMORY

| Benchmark | Description |
|---|---|
| Compression | An image compression scheme |
| Laplace | A Laplace algorithm to perform edge enhancement |
| LowPass | A low-pass filter to an image |
| SOR | A Successive Over-Relaxation (SOR) algorithm |
| Wavelet | The Daubechies 4-Coefficient Wavelet filter |

### E. Complexity

The algorithm has to first detect the distance vectors in a loop nest. Specifically, the process of searching multiple-index subscripted variable is found in $O(n)$ time, $n$ is the number of subscripted array references. After the detection process, the algorithm performs the test for loop interchange. This test is usually fast and requires $O(M)$ time, $M$ is the depth of the loop nest. Following loop interchange, the algorithm searches the loop index space for optimal unrolling vector based on the parameter estimations. This process requires time $O(MK)$, $M$ is the depth of a loop nest, and $K$ is the number of iterations in a loop. In practice, the algorithm runs surprisingly fast. This might be due to the acceptable complexity of practical applications.

## IV. PERFORMANCE EVALUATION

Our proposed method is applied to five benchmark programs taken from the High-level Synthesis Design Repository ((HLSynth95/memory) [1] listed in Table I. All the programs involve array accesses inside nested loops.

Performance of our method is measured in terms of

1) the accuracy of the measurements. This ensures that the optimal unrolling vector is found correctly.
2) the percentage of temporal reuses exploited.

### A. Accuracy

Accuracy refers to the reliability of the measurements for an unrolling vector. Measurements provide an estimation of the practical situation. Accurate measurements should be close enough to the practical situation during unroll-and-jam. To see how close our measurements are to the practical situation, we compare the estimated access ratio $R$ with the practical access ratio $R$ from the execution of the transformed program in a simulator [4].

The estimation is restricted by the number of available registers. For the sake of evaluation, the number of registers is arbitrarily set to 10. However, the principle can be applied to other values depending on the number of registers in a particular processor. For each program, we follow the procedure in section III to perform loop interchange (if any), unroll-and-jam and scalar replacement. For the unroll-and-jam, the program is transformed for each possible unrolling vector. By executing the transformed program in the simulator, we have the practical access ratio. Experiments show that estimated access ratio matches the practical access ratio with a high accuracy. For example, Table II shows the experimental results of the benchmark 'Compression'. This proves that the measurements in section III-C accurately estimate the effect of the transforms.

### B. Exploitation Rate

The effectiveness and flexibility of the procedure are now examined. This is done by computing the percentage of temporal reuse exploited in a program using (2).

TABLE II
COMPARISON OF ACCESS RATIOS FOR COMPRESSION UNDER 10-REGISTER RESTRICTION

| [v1 v2] | Estimated R | Practical R |
|---|---|---|
| [2 2] | 0.6370 | 0.6520 |
| [2 3] | 0.6040 | 0.6201 |
| [3 2] | 0.6040 | 0.6201 |

TABLE III
EXPLOITATION RATE UNDER 10-REGISTER RESTRICTION

| Prog. | Opt.$\vec{v}$ | Reg. Used | $R_{ideal}$ | $R_{Prac}$ | Expl. Rate (%) |
|---|---|---|---|---|---|
| Comp. | [2 3] | 8 | 0.4041 | 0.6201 | 66 |
| Lapl. | [1 3] | 9 | 0.2041 | 0.6081 | 49 |
| LowP. | [1 3] | 9 | 0.2041 | 0.6081 | 49 |
| SOR | [2 3] | 10 | 0.7534 | 0.8653 | 55 |
| Wave. | 6 | 10 | 0.5208 | 0.5833 | 87 |

$$ExploitationRate = \frac{1 - R_{prac}}{1 - R_{ideal}} \times 100\% \quad (2)$$

where $R_{prac}$ is the practical memory access ratio after and before the optimization. $R_{ideal}$ is the ideal memory access ratio. The ideal case happens when there are sufficient registers to store all the data having temporal reuses. The ideal access ratio is inherent to computations and does not depend on the way a program written.

Evaluation is performed on the benchmarks under the 10-register restriction. The results are shown in Table III. The second column in the table is the optimal unrolling vector selected by the procedure. The third column is the number of registers actually used during the optimization. The results in the last column show that the exploitation rate varies from 49% to 87% for different benchmarks. This is because the exploitation rate is determined by the data dependences in a particular program and the number of available registers. Low exploitation rate can be due to data dependences preventing transformations and/or limited number of registers. The last two columns in Table III show that significant improvements to data intensive applications have been achieved by our method.

## V. CONCLUSION

An automated method for reducing the number of memory access by transforming the source code of a data intensive program is proposed in this paper. This method has the advantage that the exploration space is broadened compared with previous works by considering temporal reuse generated by multi-index subscripts. Second, this approach performs a sequence of high-level compiler techniques to exploit maximum amount of temporal reuses under register restriction. Third, the measurements designed in our approach give an accurate estimation of the practical effect. Experimental results on a number of benchmarks confirm that significant improvements can be achieved for data intensive programs using our method.

## REFERENCES


[1] 1995 high-level synthesis design repository. *http://ftp.ics.uci.edu/pub/hlsynth/HLSynth95/*, 1995.

[2] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 233–246. ACM, June 1984.

[3] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. June 1995.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 53–65. ACM, June 1990.

[6] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–357, December 1997.

[7] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, November 1994.

[8] F. Catthoor, K. Danckaert, S. Wuytack, and N. Dutt. Code transformations for data transfer and storage exploration preprocessing in multimedia processors. *IEEE Journal on Design and Test of Computers*, 18(3):70–82, May-June 2001.

[9] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1998.

[10] R. Gonzales and M. Horowitz. Energy dissipation in general-purpose microprocessors. *IEEE Journal of Solid-state Circuit*, SC-31(9):1277–1283, September 1996.

[11] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Design Automation Conference*, pages 304–307. ACM, June 2000.

[12] A. Koseki, H. Komastu, and Y. Fukazawa. A method for estimating optimal unrolling times for nested loops. *Information Processing Society of Japan Journal*, 37(06):376–382, 1997.

[13] V. Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*, pages 153–166. ACM, June 1997.

[14] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *Proceedings of the IEEE Conference on Computer Aided Design*, pages 384–390. IEEE, November 1994.

[15] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.