

# Fast and Power-Efficient Dynamic Data-Layout with DMA-Capable Memories

Mohammed Javed Absar\*, Francesco Polletti<sup>†</sup>, Pol Marchal\*, Francky Catthoor\* and Luca Benini<sup>†</sup>

\*IMEC vzw., Katholieke Universiteit Leuven, Leuven, Belgium, Email:javed@imec.be

<sup>†</sup>Universita' di Bologna, Bologna, Italy

**Abstract**—Cache takes advantage of the spatial and temporal locality commonly found in programs to bridge the gap between processor and memory speeds. Dynamic Data-Layout, to maintain a high level of spatial locality throughout the entire course of the application run, has attracted some attention recently. However, its usage has been highly conservative, due to the concerns over power and execution time increase resulting from the explicit-copying required by the processors to change layout at run-time. In this paper we show that such concerns can be allayed to a large extent through the use of DMA-Capable Memories (i.e. memories with specialized DMA which perform the layout change inexpensively). We validate our claims by performing dynamic data-layout on five applications. Our testing environment is an integrated SystemC-based simulator which tracks, on a cycle-by-cycle basis, the energy spent by each component of the system. By performing dynamic data-layout using the proposed scheme we get over 20% reduction in energy and execution time, even for applications where explicit copy actually results in worse performance compared to the static layout situation.

## I. INTRODUCTION

Cache provides fast and cheap (in terms of power) access to the data compared to the lower level memories (L2-cache and main memory). It is able to do so by virtue of being closer to the processor and much smaller in size compared to lower level memories. Cache therefore allows considerable reduction in overall execution time and power consumption of embedded systems. For the cache to perform well, however, the program must exhibit high temporal and spatial locality.

In general, array elements with nearby indexes tend to be accessed closer in time. This characteristic exhibited by ordinary programs is called spatial locality. Caches exploit this by loading a cache-line, i.e. a number of nearby memory locations whenever any one of those locations are accessed.

Loop transformations can be used to improve spatial locality. However, as noted in [7], there are three drawbacks to using loop transformations to influence spatial locality:

- loop transformations are constrained by data-dependencies
- complex imperfectly nested loops pose a challenge for loop transformations
- locality characteristics of all the arrays accessed in the nest are affected by them, some perhaps adversely

If the layout of every array remains fixed throughout the entire duration of the program we term it as *static data-layout*. The layout of the individual arrays could be different within the same program. Note that with an m-dimensional

array, m-factorial layouts are possible. If we include diagonal layouts, then many more combinations are possible. Whatever the layout for each of the arrays in the program, if they are all fixed for the entire duration of the program execution we still refer to it as static-layout. If the layout of an array is changed at run-time we term it as *dynamic data-layout*.

```
for (i=...) for (j=...) f1 (a [i] [j]) ;  
...  
for (i=...) for (j=...) f2 (a [j] [i]) ;
```

In the toy-example above, the array  $a$  is accessed in first line in row-major form. The same array further down in third line is accessed in column-major form. Assuming the array is quite large that only a small part of it fits in the cache, spatial locality would play a big role in the cache performance of the above code. For high spatial reuse, the array must initially be stored in row-major form and then must be laid out as column-major for the third line.

Dynamic layout, as in example above, has its advantages and drawbacks. While it can be effective in increasing spatial locality once the layout has been changed to the locally optimal one, the re-mapping itself may need large amount of data transfers. That is, there is an overhead involved which may actually increase the overall execution time and energy consumption. However, in this paper we show that with new hardware possibilities this overhead can be reduced significantly. The novel idea introduced in this paper is that by using DMA-Capable-Memories (i.e. memories coupled with an intelligent DMA) layout changes can be done rather inexpensively at run-time and so even for program regions where reuse is not high, layout changes can be used to improve power and execution time by increasing the spatial locality exploitation capacity of the data cache. In fact, for such cases we were able to obtain more than 20% reduction in energy and execution time.

## II. RELATED WORK

The state-of-the-art in techniques to improve temporal locality of programs is quite advanced today [2][12], even though it has been limited somehow to access to arrays inside nested loops by indexes that are linear functions of enclosing iterators. Loop transformations can sometimes be sufficient to improve spatial locality. But, as discussed before, relying always upon them may not be a good idea.

A formal framework for improving spatial locality of arrays, that are accessed in loop-nest by affine subscript functions, by changing layout of arrays in memory was presented by Kandemir et al [6]. The authors consider loop and data layout transformations together in a unified framework and attempt to come up with transformations for both (loop and data) to improve temporal and spatial locality. For each array they try to establish a *fastest changing dimension*, i.e. a single array-dimension which changes when the innermost loop executes. The array is laid out with the fastest changing direction being the innermost dimension in the traversal of the array in memory. Anderson et al [1] had a similar approach for improving spatial locality but were restricted to inheriting loop transformation decisions made in a previous step by the SUIF compiler [11].

Dynamic data-layout has attracted attention only recently. Ding and Kennedy [3] look at the problem from the perspective of irregular programs, such as molecular dynamics simulations, where access pattern are unknown until run-time and keep changing throughout the execution. This situation can occur in multi-media systems as well and we are looking at that but in this paper we limit ourselves to regular access inside nested-loops.

In [5] Kandemir extends his earlier work on static layout [6] to handle layout changes at run-time. The layout of arrays for different regions of the code (program segments) are determined at compile-time and the book-keeping code that is necessary to dynamically transform the layouts of arrays between different program segments is inserted in the code (again at compile time). However the layouts are changed during program execution.

Recently, algorithms to improve cache behavior of signal transforms, e.g. Fast Fourier Transform (FFT) and Walsh-Hadamard Transform (WHT), using dynamic data-layout have been formulated [9]. The optimization however is applicable only to the class of signal transforms that can be factorized.

### III. DYNAMIC DATA LAYOUT WITH DCM

#### A. Motivating Example

Consider the code below which is an adaptation of the program to compute inverse of a matrix [10]. Here, an  $N \times N$  given matrix  $A$  is decomposed into a lower triangular matrix  $L$  and an upper triangular matrix  $U$  such that  $A = LU$ . The new matrices  $L$  and  $U$  can be used to solve the equation  $Ax = b$ , where  $x$  is a  $N \times 1$  vector of  $N$  unknowns and  $b$  is  $N \times 1$  vector of given constants. Since  $A = LU$ ,  $Ax = b$  can be re-written as  $L(Ux) = b$ . Firstly we solve  $Ly = b$ , treating  $y$  as a  $N \times 1$  vector of  $N$  unknowns. This is trivial to solve. Once we have the values for  $y$ , we solve  $Ux = y$  to find the vector  $x$ . The Eq.  $Ax = b$  is solved repeatedly, setting  $x$  to (1000...), (0100..), (0010...). The solutions form the columns of matrix of  $A^{-1}$ .

```
/*decompostion : A = LU */
for(i = 0 ; i < n ; i++)
  for(j = 0 ; j < n ; j++){
```

```
    sum = 0 ;
    if( j > i ){
      for( k = 0 ; k < i; k++ )
        sum += L[j][k] * U[k][i];
      L[j][i] = (A[j][i] - sum)/U[i][i];
    }else{
      for(k = 0 ; k < j; k++)
        sum += L[j][k] * U[k][i];
      U[j][i] = A[j][i] - sum ;
    }
  }
}
/*solve Ax=b repeatedly (b unit vec)*/
...
for( j = 0 ; j < n ; j++){
  ...
  for( i = n-1 ; i > 0 ; i--){
    sum = 0 ;
    for( k = i+1; k < n ; k++)
      sum += U[i][k]*x[k]
    x[i] = (x[i] - sum)/U[i][i];
  }
}
```

Suppose the matrices involved above are much large than the cache size. In such cases, for the second part (solving  $Ax = b$ ), having matrices  $L$  and  $U$  stored as row major would be much better for cache performance. Now for the first part (decomposition), we can indeed perform loop transformations to change access of  $U$  from column-major (as in the code above) to row-major. But then we would have changed access to  $L$  from row-major to column-major. This is a classical problem of using loop transformation to influence spatial locality. What, therefore, is required in the above example is a dynamic layout change of  $U$  from column-major to row-major, at the point where the LU-Decomposition ends.

The main drawback of changing the layout at run-time to improve spatial locality is that moving the data in memory to instrument layout change takes extra energy and time. We attempt to resolve this problem in the next section.

#### B. Efficient Layout Change with DMA-Capable Memories

In a traditional setting, the processor is used to change the layout. Basically, the processor copies an array element to its register and then writes it back to its new location. We term this process as *explicit copy*. Since processors are inefficient in doing data-transfers, performance and energy is lost. making layout change an expensive process. Such a solution was assumed in [5]. Naturally, the author provides an elaborate scheme on how to keep layout changes to a minimum.

In our system, we equip the memories with a *customized* memory access controller (DMA). Originally, DMAs were developed as a co-processor to free the processor from performing slow IO-operations. They copy data in bursts. Burst transfers require less cycles to complete than transferring each data at a time as is done by the processor. The DMA in our system is closely coupled with the memory and is

able to transfer a set of array elements from one location in memory to another. We provide a high-level API to program the DMA. Once programmed, the DMA is able to transform an entire two-dimensional array from row-major to column major (and vice-versa). It generates an interrupt once the transfer is complete. The processor can perform other tasks during the transfer.

Therefore, concerns over performance and power overhead resulting from aggressive usage of dynamic data-layout can be allayed to a large extent using our proposed technique. This opens up new opportunities in active localized layout transformation for not only improving spatial locality but also reducing conflict misses [8].

## IV. EXPERIMENTS

### A. Testing Environment

We performed our experiments on a SystemC-based cycle-accurate model of ARM multi-processor environment. The ARM processor has a local instruction cache (2KB Direct Mapped) and a data cache (2KB Direct Mapped). They are connected via the system bus (STBus) to the main memory (SDRAM). This memory has a customized DMA which can transfer a set of data from one location to another. Not only that, it can change the layout of the data (for example, from row-major to column major), during the copying. The system uses power models to compute energy consumption [4].

### B. Benchmarks

In total we performed experiments with five applications. For some applications it was very clear from the high reuse-factor that changing layout would be beneficial. For others it depends on how much the layout change itself would cost. For these cases we attempt to establish that our approach is superior to the existing art (explicit-copy).

1) *Matrix Addition*: This is a simple program where two  $N \times N$  matrices  $A$  and  $B$  are combine to generate a third matrix  $C$ , such that  $C = A + B^T$ .  $A$  and  $B$  are assumed to be stored originally in row-major format. If  $N \times N$  is small enough so that  $A$ ,  $B$  and  $C$  can all fit conveniently together in the cache, then no layout change is necessary. If fact, it would be an over-kill. We therefore set  $N \times N$  to large enough ( $128 \times 128$ ). Matrix addition is a simple process with no reuse, i.e. each element is accessed only once, and so the question is whether it is still a good idea to do layout transformation.

2) *Matrix Multiplication*: Two matrices  $A$  and  $B$ , each ( $50 \times 50$ ), are multiplied to generate a third matrix  $C = A \cdot B$ .

3) *Gaming Sound*: In a typical PC or handheld war-game the user(hero) receives sounds from many directions to which he must react to protect himself. The sound reaching the hero is delayed and attenuated depending on the distance and obstruction between the beasts (source) and the hero (receiver). The algorithm used here performs a mix of the different sounds reaching the hero with various attenuation and delays.

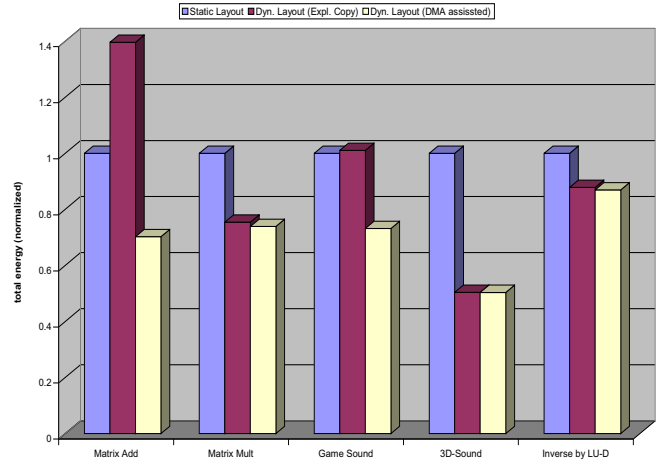


Fig. 1. Total energy spent in each version (static-layout; dynamic with explicit copy; and dynamic with DMA-Capable Memories) of each application.

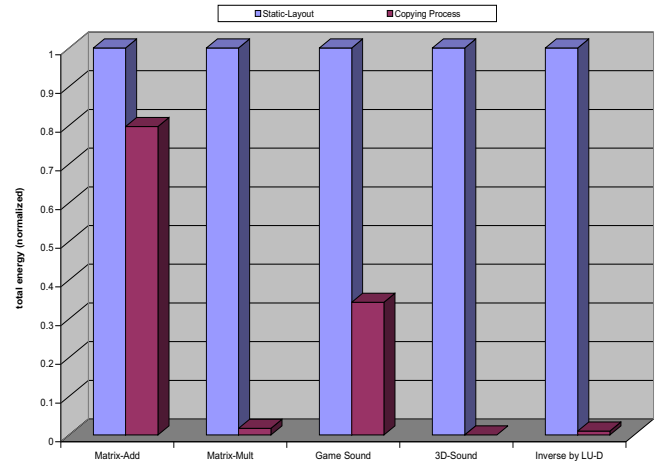


Fig. 2. Energy required to change the layout is compared with energy spent in executing the entire application.

4) *Sound-Spatialization*: This application is from the domain of audio signal processing. In a typical movie-hall or the modern home-theater system, there are usually six to eight independent sources of sound (speakers) placed in various directions. The listener therefore gets to enjoy a 3-D audio field. When users are constrained to use headphones (as in an aircraft), the same impression of 3-D sound can (almost) be re-created by mixing the sounds from the six channels in a way that takes into account the human auditory system. The algorithm that we use has a large set of coefficients which filter each of the sound inputs. There is high data reuse in this application.

5) *Matrix Inversion by LU-Decomposition*: This application was explained in detail before.

## V. RESULTS

For each application we have performed simulation with three versions of the code (a) static layout, (b) dynamic layout with explicit-copy and (c) dynamic layout using DMA-Capable-Memories. Fig. 1 shows the total energy spent by the system for each version and each application. For Matrix-Addition we tried to improve spatial locality firstly by per-

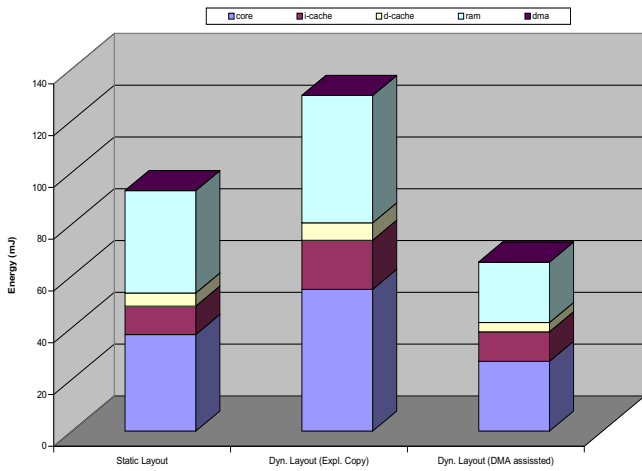


Fig. 3. Energy break-up for the matrix addition application. Note the significant increase in processor and RAM energy for the explicit copy case

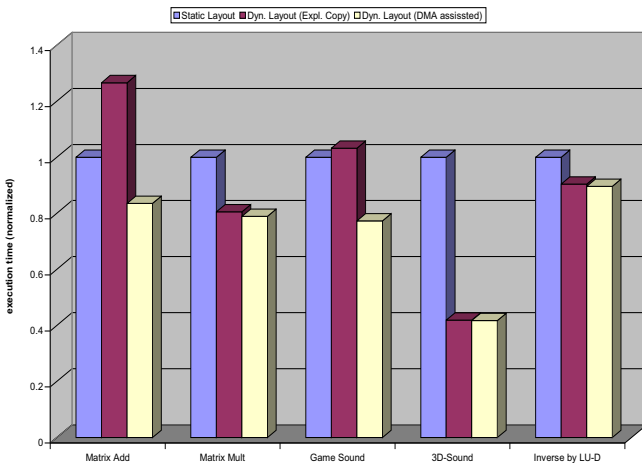


Fig. 4. Total execution time for each application. For Matrix-Addition and Game-Sound applications explicit-copy worsens the situation over static layout.

forming explicit copy (of array  $B$  from row-major to column-major). Even though during the addition phase spatial locality is good, the process of copying spends too much energy and so the overall performance is worse than the static-layout. Implementing the same layout change using DMA-Capable-Memories gives a much better overall performance.

Fig. 2 shows compares the price paid in energy for doing the explicit-copy itself. For each application we show in the second column the energy spent in just changing the layout. For a fair comparison its value is normalized with respect to the energy of running the original application (with static-layout). Comparing Fig. 1 and Fig. 2 it is clear that Matrix-Add and Game Sound do not fare well with explicit-copy because it is far too expensive compared to the energy requirements of the whole application itself.

Fig. 3 shows the energy spent by different components of the system for each version of the Matrix-Add example. Because we use ARM7 core, the processor energy is high compared to the rest of the system. This undermines to some extent the significant gains on data cache and RAM. The increase in energy of explicit-copy comes from two sources,

RAM and the core and to some extent the data and instruction cache. The DMA-Capable-Memories approach conserve the processor energy by using the DMA. The DMA itself, being a dedicated engine, uses negligible energy as seen in Fig. 3.

Fig. 4 shows the overall execution time for each application. Note that in terms of both energy and execution time for the applications Matrix-Mult, 3D-Sound and Inverse by LU-D, explicit-copy is much better than static layout and only slightly worse than layout change using DMA-Capable Memories. This is so because of high reuse. In such cases, the benefits from layout improvement is so large that the cost of making the change is almost masked.

## VI. CONCLUSION

In this paper we have shown that changing the layout of multi-dimensional arrays in memories at run-time, to improve spatial locality in cache, can be done in a much more cost effective manner using DMA-Capable Memories. Results reveal that even in cases where an array with the new layout is used only once, such as in Matrix-Add, a significant 20% over gain in energy and execution time can be obtained. Therefore, our study attempts to create a new-mindset wherein data-layout optimization are performed at run-time in a much more aggressive manner compared to what exists today. We are currently working to extend our technique to localized layout transformations which would impart to a good extent the same degree of freedom to layout transformations as is currently enjoyed by loop transformations.

## REFERENCES

- [1] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *ACM SIGPLAN Conference on Programming Language, Design and Implementation (PLDI)*, 1995.
- [2] U. Banerjee. *Data Dependencies*. Kluwer Academic Publishers, 1988.
- [3] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *ACM SIGPLAN Conference on Programming Language, Design and Implementation (PLDI)*, 1999.
- [4] P. Francesco, P. Marchal, and D. Atienza. An integrated hardware/software approach for run-time scratchpad management. *ACM Design Automation Conference*, 2004.
- [5] M. T. Kandemir and I. Kadayif. Compiler-directed selection of dynamic memory layouts. *International Symposium on Hardware/Software Codesign (CODES)*, 2001.
- [6] M. T. Kandemir, J. Ramanujan, and A. Chowdhury. Improving cache locality by a combination of loop and data transformation. *IEEE Transaction on Computers*, 48(2), 1999.
- [7] M. T. Kandemir, J. Ramanujan, and A. Chowdhury. A compiler technique for improving whole-program locality. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.
- [8] C. Kulkarni, C. Ghez, and M. Miranda. Cache conscious data layout organization for embedded multimedia applications. *Design Automation and Test in Europe (DATE)*, 2001.
- [9] N. Park and V. Prasanna. Dynamic data layouts for cache-conscious implementation of signal transforms. *IEEE Transactions on Signal Processing*, 2004.
- [10] W. Press and B. Flannery. *Numerical Recipes in C Example Book : The Art of Scientific Computin*. Cambridge University Pres, 1992.
- [11] R. Wilson, M. Lam, J. Hennessy, R. French, and C. Wilson. Suif: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 1994.
- [12] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Conference on Programming Language, Design and Implementation (PLDI)*, 1988.