Cache-efficient Memory Layout of Aggregate Data Structures

Preeti Ranjan Panda
Synopsys Inc.
700 E. Middlefield Rd.
Mountain View, CA 94043, USA
panda@synopsys.com

Luc Semeria
Clearwater Networks Inc.
160 Knowles Dr.
Los Gatos, CA 95023, USA
luc@clrw.net

Giovanni de Micheli
Computer System Laboratory
Stanford University
Stanford, CA 94305, USA
nanni@stanford.edu

ABSTRACT

We describe an important memory optimization that arises in the presence of aggregate data structures such as arrays and `structs` in a C/C++ based system design methodology. We present an algorithm for determining an optimized memory layout of such data. Our implementation consists of a pointer analysis and resolution phase, followed by memory layout optimization. Experiments on typical applications from the DSP domain result in up to 44% improvement in memory performance.

Categories and Subject Descriptors

B.3 **[Memory Structures]**; B.5.1 **[Register-Transfer-Level Implementation]**: Design—*Memory Design*; B.5.2 **[Register-Transfer-Level Implementation]**: Design Aids—*Automatic Synthesis, Optimization*; B.7.1 **[Integrated Circuits]**: Types and Design Styles—*Memory Technologies*; D.3.4 **[Programming Languages]**: Processors—*Compilers,Optimizations*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Data Layout, Data Cache, Embedded Processors

1. INTRODUCTION

Since complex embedded applications are typically composed of both synthesized hardware blocks as well as application software executing on an embedded processor, the supporting design methodology has evolved to embrace several aspects of both hardware synthesis and software compilation of embedded code [6, 5, 4]. While Hardware Description Languages (HDLs) such as VHDL and Verilog were adequate to describe behavioral-level circuitry in the past, the push towards System-on-a-Chip level integration found them wanting in sufficiently high abstract modeling constructs. To alleviate the problem, the industry has been experimenting with replacing HDLs by C/C++ (e.g., [1]). In this design methodology, new optimization and exploration opportunities arise not only in the hardware synthesis side of the equation, but also on the compilation side. Many optimizations that are deemed computationally expensive for traditional compilers can now be performed in the embedded domain because a designer can afford to spend extra time to improve performance and power characteristics since there is only a single application.

Design of the memory subsystem needs careful attention because the same functionality can be implemented in several different ways, each with varying impact with respect to performance, area, and power consumption. In recent years, researchers have analyzed several different memory architectures and the impact of storage policies on these architectures in embedded systems. Among the earliest research efforts to study the memory allocation problem in the context of video data streams was the PHIDEO system [7]. Exploration environments for determining an optimized application specific on-chip memory architectures are studied in [9, 3]. More recently, in the context of C/C++ based synthesis, researchers have addressed the issue of finding reasonable hardware interpretations for dynamic memory allocation-related features [12, 10].

The work presented in this paper complements the *data layout* optimizations reported in works such as [8], where the focus is on the relative placement of user-declared data structures in memory so as to improve cache performance. We demonstrate that a significant amount of additional benefit can be obtained by either coalescing/merging different arrays or splitting an array of `structs` into separate arrays depending on the circumstances. The work also complements the memory synthesis techniques such as [11] by way of extending the analysis of aggregate data structures into the embedded processor-cache environment, thereby forming a more complete framework for hardware/software co-design.

2. OVERALL DESIGN FLOW

In typical programming languages such as C/C++, the relative placement of individual elements of an aggregate data structure is mandated by the language itself, and a compiler has no option but to obey the storage requirements, even if better alternatives are available from the performance point of view. This is because a C compiler typically sees only a part of the whole program at a time, and modifying the storage of data in non-trivial ways is unsafe in general. However, in embedded systems, where the entire application is usually visible to the compiler, an aggressive compiler can perform such structural transformations to the data after verifying

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

the safety of the operation. Works such as [13] have investigated a general pointer analysis framework, which has been incorporated into techniques for hardware synthesis of pointers [10]. The same analysis information can be used to determine whether it is safe to transform the memory representation of data structures.

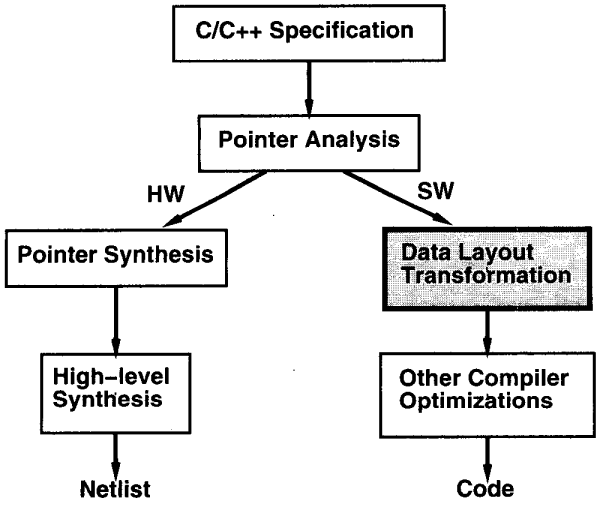

Figure 1: Where data layout transformation fits into the overall design flow

The overall framework is shown in Figure 1. The C/C++ specification is first analyzed for pointer arithmetic and the pointers are resolved. The pointer synthesis phase uses the result of the analysis to convert the original code into code with no pointers, which is then taken through a traditional hardware synthesis flow. In the software path, we perform the data layout transformation followed by the regular compiler optimizations and code generation phases. The hardware path may also have its own data layout transformations.

3. LAYOUT OF STRUCTS: AN EXAMPLE

The data restructuring optimization and its implications can be illustrated with the example shown in Figure 2(a). p is an array of `structs` with two fields (a and b) in each element, and q is an array of integers. The obvious way to store the data in memory is shown in Figure 3(a). Consider an example data cache with 4 lines; 2 words per line. When $p[0].a$ is accessed in the first iteration of Loop L1, the cache fills the entire line, thus fetching $p[0].b$ also. However, $p[0].b$ is never used in the loop, thus the cache subsystem is under-utilized. Similarly, in Loop 2, $p[0].a$ and $p[1].a$ are also unnecessarily fetched into the data cache (Figure 3(c)), thereby degrading performance.

The under-utilization problem can be solved by reorganizing the data based on an analysis of the memory accesses in the application. The transformed code is shown in Figure 2(b). The original $x.a$ has now become a, and the original $x.b$ and q have been merged into a new `struct` y. This ensures that spatial locality is maximized by storing consecutively accessed data in nearby locations. The storage strategy for the transformed code is shown in Figure 4(a). As shown in Figure 4(b) and (c), no useless data is fetched into the cache in any iteration of the two loops.

The data layout transformation results in the cache miss ratio decreasing from 62.5% to 37.5%. For a typical processor-memory

```
struct x {
   int a;
   int b;
} p[1000];
int q[1000];
...
avg = 0;
L1:  for (i = 0; i < 1000; i++)
   avg = avg + p[i].a;
avg = avg / 1000;
...
L2:  for (i = 0; i < 1000; i++)
{
   p[i].b = p[i].b + avg;
   q[i] = p[i].b + 1;
}
```

(a)

```
int a[1000]; // originally x.a
struct y {
   int q; // originally q
   int b; // originally x.b
} r[1000];
...
avg = 0;
L1:  for (i = 0; i < 1000; i++)
   avg = avg + a[i];
avg = avg / 1000;
...
L2:  for (i = 0; i < 1000; i++)
{
   r[i].b = r[i].b + avg;
   r[i].q = r[i].b + 1;
}
```

(b)

Figure 2: (a) Example code fragment (b) Transformed code

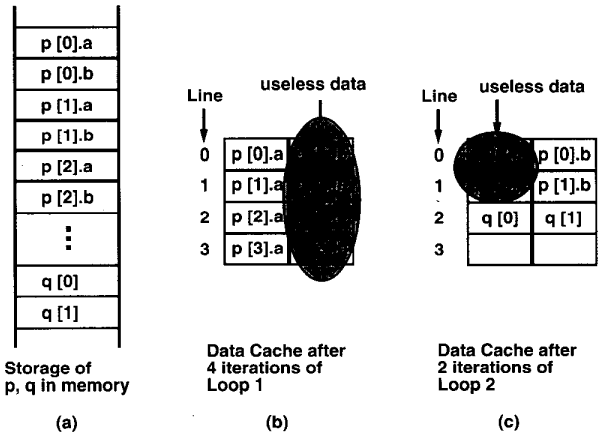

Figure 3: Cache behavior of original code

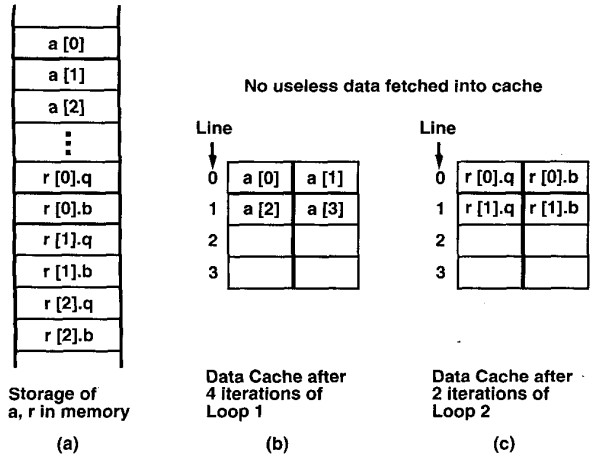

Figure 4: Cache behavior of transformed code.

system interface where a cache miss incurs a penalty of, say 15 processor cycles, and a cache hit results in a single cycle data access, this amounts to a cycle-count reduction of 34%.

The data restructuring transformation is not subsumed by other known optimizations in compiler/architecture and CAD literature. Data fetched into cache during Loop L1 is, in general, not present when L2 begins because of capacity misses. In the example of Figure 2(a), the *loop fusion* transformation does not apply because of data dependencies - *avg* needs to be computed before L2 begins. Finally, just splitting the arrays into individual components (without regrouping) is generally not sufficient because of the possibility of cache conflicts. Although the illustrative transformation of the code from Figure 2(a) to Figure 2(b) was simple, the general problem of deriving the most efficient data layout is non-trivial. In Section 4, we provide an algorithm for this transformation.

4. DATA LAYOUT TRANSFORMATION

The data layout problem is formulated as follows: given a set of arrays of either simple data types such as integer, or aggregate data types such as `structs`; and a set of innermost loops in a program accessing different arrays with different array index expressions, determine an efficient layout of the arrays in memory that maximizes the data cache utilization.

As a first step, we dismantle the user-specified grouping of data. For example, an array of `structs` may be split into separate arrays of its constituent fields.

4.1 Splitting into individual arrays

In the example code of Section 3, the splitting of the array of `structs` into two different arrays was trivial. However, in general, the task of splitting the address space while preserving the integrity of the code is rendered more complex by the presence of constructs such as the dereferencing, arithmetic, and type casting of pointers. The underlying memory representation in C (as in many other programming languages) is a continuous address space. Pointers represent addresses in this address space. However, to perform our optimizations, we need to map this address space into distinct sets of locations. For our memory representation, we use the notion of *location sets* introduced in [13]. Location sets support all C data structures including arrays, `structs`, arrays of `structs` and `structs` containing arrays. In order to handle large C programs efficiently, some memory data, such as the different elements of an array, are combined into a single location set. The representation of the memory into a set of distinct location sets is created by accurately analyzing how the different locations are being accessed in the code.

```
struct {
   int a;
   int b;
} ts[10];
int ti[20];
int *p;
...
if(...)
   p = (int *)ts;
else
   p = (int *)ti;

for(i=0; i<10; i++) {
   ...  = *p;
   p += 2;
}
```

(a) Original Code

```
int ts_a[10];
int ts_b[10];
int ti[10];
struct spc_pointer {
   short tag; short index;
} p;
...
if(...)
   p.tag = 0; p_index=0;
else
   p.tag = 1; p_index=0;

for(i=0; i<10; i++) {
   if(p.tag==0) // p→{ts[i].a}
      ...  = ts_a[p.index/2];
   else // p→{ti[i]}
      ...  = ti[p.index];
   p.index += 2;
}
```

(b) Transformed Code

Figure 5: Resolution of pointers

Pointer analysis [13] is a compiler pass to identify at compile-time the potential values of the pointers in the program. This information is used to determine the set of locations each pointer may point to at any point in a program. All memory data for a program can be arranged into a set of distinct location sets which can be mapped to separate arrays (or scalar variables).

After partitioning the memory into a set of arrays, we still need to resolve pointers. Since the underlying memory layout is changed, the behavior of pointers and more specifically of type casting and pointer arithmetic may be altered. The resolution of pointers consists of two passes. The first pass, where pointers are dereferenced, consists of replacing loads and stores by case statements in which the locations the pointer may access are explicitly read or written.

The value of the pointers are then encoded as they do not represent addresses in the original address space anymore.

Consider the code segment in Figure 5(a). After analyzing the code, the two fields of the `struct` are represented by two distinct location sets which can be mapped to two separate arrays `ts_a` and `ts_b`. In this example, the mapping of the array of `structs` `ts` onto `ts_a` and `ts_b` is not straightforward. Resolving pointers makes this mapping easier. The resulting code after removing pointers and mapping the two fields of the structure to separate arrays is the shown in Figure 5(b). We omit the details for brevity.

4.2 The clustering algorithm

Our objective is to regroup the split arrays based on their memory accesses. We represent the input specification as a bipartite graph with a set of *a-nodes* representing arrays in the specification and a set of *l-nodes* representing loops (Figure 6). Since most of the memory accesses and computations occur in the innermost loops, we consider only innermost loops out of nested loop structures. An a-node and an l-node are connected if the corresponding array is accessed in the loop. l-nodes have a weight equal to the number of times the loop is executed. If the loop bound cannot be established at compile-time, we assume that a profiler can supply that information. At this level of abstraction, we do not differentiate between memory read and write accesses. However, we do assume that register allocation has already been performed. For example, in Loop 2 of Figure 2, we assume that the statement "`q[i] = p[i].b + 1`" does not involve a memory read because `p[i].b`, which was written in the previous statement, could be stored in a register.

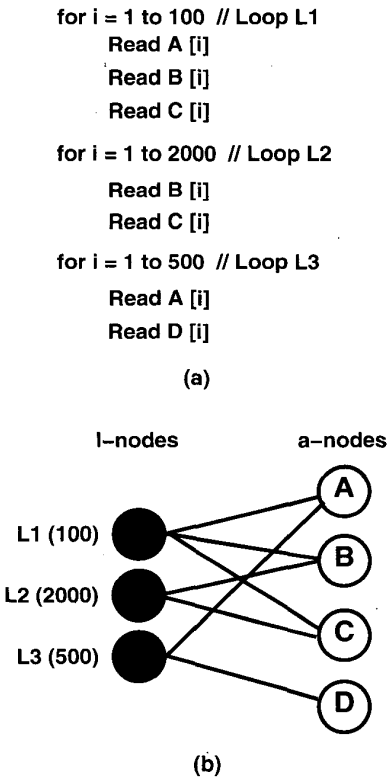

Figure 6: Sample memory access pattern in loops and the associated graph

Algorithm *Cluster*
Input: Specification represented as a series of loops and array accesses
Output: Group of clusters CS

for all arrays j
    ASSIGNED [j] = FALSE
Set of clusters CS = ϕ
Sort all loops in decreasing order of total number of array accesses
for all loops L
    for all arrays i in loop L with ASSIGNED [i] = FALSE
        min_cost = ∞
        for all existing clusters C (including ϕ)
            cost = find_cost (i, C);
            if (cost < min_cost)
                min_cost = cost
                best_cluster = C
        if best_cluster = ϕ
            Create new cluster $C' = \{i\}$
            CS = CS $\cup C'$ // add C' to the set of clusters
        else
            best_cluster = best_cluster $\cup\{i\}$
        ASSIGNED [i] = TRUE
end Algorithm

Figure 7: Algorithm *Cluster*

Figure 6 shows a sample memory access pattern and the associated bipartite graph. For example, the l-node corresponding to Loop L1 is connected to a-nodes A, B, and C. Algorithm *Cluster* in Figure 7 shows the overall strategy involved in clustering the individual arrays of the graph into `structs`.

The vector ASSIGNED keeps track of whether each array is assigned to a cluster yet or not. CS is the set of generated clusters, each consisting of one or more arrays. CS is initialized to the empty set. Since the number of possible clusters is exponentially large, we devise a heuristic strategy where we first select the arrays in the most frequently executed loops for assignment to clusters. Further, the assignment decision, once made, is never altered in the interest of reducing computational complexity.

We begin with the loop with the maximum access count, and consider for assignment to clusters all the arrays accessed in the loop. For every array i that is unassigned, we compute the *cost* of assigning the array to each of the clusters in CS, including the null cluster ϕ (which corresponds to the creation of a new cluster). We discuss the actual cost computation in Section 4.3. In the overall strategy, we select for assignment the cluster C that minimizes the cost. If C is the null cluster ϕ, then we create a new cluster C' with i as the sole element. Otherwise, we add i to the cluster C. Finally, we mark array i assigned and continue our cluster assignment process.

4.3 Cost computation

The most important step in Algorithm *Cluster* is *find_cost* (C, i), the computation of the cost associated with assignment of array i to cluster C. In our model, the cost comprises two components:

- the penalty associated with assigning two correlated arrays into separate clusters
- the penalty associated with assigning two uncorrelated arrays into the same cluster

Algorithm *find_cost*
Input: cluster C, array i
Output: integer representing cost of assignment

```
cost = 0
// Part I: cost due to separating i from correlated arrays
for all edges e connected to array i (i.e., for all loops accessing i)
    penalty = TRUE
    for all arrays a accessed in loop e
        if a ≠ i
        and ASSIGNED [a] = TRUE
        and CLUSTER [a] = C
        and are_correlated (i, a, e)
            penalty = FALSE
    if penalty = TRUE
        cost = cost + EDGE_WEIGHT (e)

// Part II: cost due to clustering i with uncorrelated arrays
for all edges e such that ∃ j, j ∈ C and j ≠ i
    if are_correlated (i, j, e) = FALSE
        cost = cost + EDGE_WEIGHT (e)
end Algorithm
```

Figure 8: Algorithm *find_cost*

The components above are distinct; one does not subsume the other. Whether two arrays x and y are correlated or not is determined by a function *are_correlated (x,y, l)* which returns TRUE if x and y are different arrays with the same number of elements, and the index expressions of their accesses in loop l are *affine* (linear) and differ by a constant. This condition is essential to maintain spatial locality in the data cache, and is inherent to the definition of correlation for data layout. For example, if the respective index expressions are `a[2i]` and `b[2i+1]`, then upon clustering `a` and `b`, a fetch to `b[2i+1]` possibly brings `a[2i+1]` into the cache (assuming cache line size ≥ 2 and `a[2i+1]` lies on the line boundary), which is used in the next iteration. However, if the expressions are `a[i]` and `b[2i]`, then there is no spatial correlation as the distance between the two accesses increases with increasing iteration count. Function *are_correlated* detects a correlation when the expressions are affine.

The cost computation for assigning array i to cluster C is described in algorithm *find_cost (C, i)* in Figure 8. The first part computes the cost due to separating array i from other compatible arrays. This is done by checking all the loops accessing i to detect the existence of at least one other array a belonging to the same cluster C which is correlated to i. The *penalty* variable keeps track of this existence. If *penalty* is still TRUE at the end of the search, then we penalize the assignment, else, the incremental cost is zero. The amount of assigned cost is simply the loop count, since that is a measure of the extent to which the assignment proves costly.

In the second part, we compute the cost due to clustering i with uncorrelated arrays. For all other arrays j already assigned to cluster C, we check to see if i and j are uncorrelated in any loop; in such instances, we add the loop count as a penalty - the cost for clustering uncorrelated arrays together. Note that two arrays, while being correlated in one loop, may be uncorrelated in a different loop. This is how we handle conflicting array accesses in different loops. This completes the cost computation. The overall running time of the clustering algorithm, with n arrays and k array references in the program is $O(n^3k^2)$. We omit the details for brevity.

5. EXPERIMENTS

We implemented the clustering algorithm in the SUIF compiler framework [2]. After the pointer analysis phase, we split the aggregate data structures (e.g., arrays of `structs`) into individual arrays. We then executed the clustering algorithm and generated the transformed program with the revised data structures. Finally, we ran cache simulations using the SHADE simulator of the SPARC processor, with varying data cache parameters. We report the results in this section.

Figure 9 shows the applications on which we performed our experiments, on the x-axis. FFT (Fast Fourier Transformation), FIR (Finite Impulse Response), Kai-win (Kaiser Window-based coefficient computation), and RTPSE (Real-time power spectral estimation) are important signal processing applications. PIC-1-D (one dimensional particle-in-cell) and Disc-Ord (Discrete Ordinate Transport) are common in scientific computation.

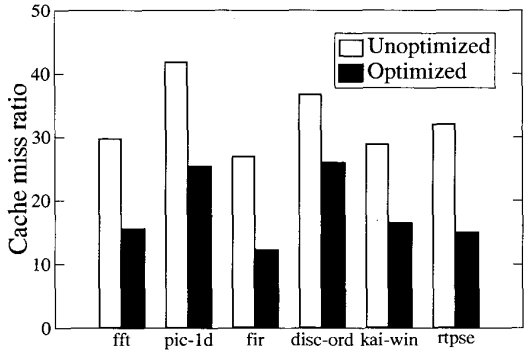

Figure 9: Data cache miss ratios

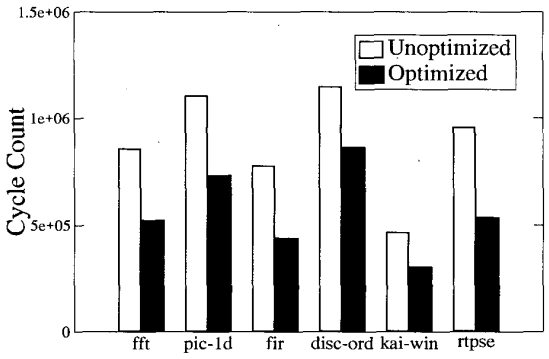

Figure 10: Comparison of cycle counts

Figure 9 shows the data cache miss ratio and after our data layout optimization. We observe a significant improvement in the cache performance. Figure 10 shows a comparison of the number of memory cycles computed assuming a cache miss penalty of 16 processor cycles. We observe an improvement of up to 44% on the original code. The measurements are taken on an architecture with a data cache size of 2 KB with a cache line size of 16 bytes. The run times of our algorithm are negligible (less than a second). In particular, since we rely on a static analysis of the application, the run time for data layout is not dependent on the array sizes, loop counts, and other dynamic run-time parameters.

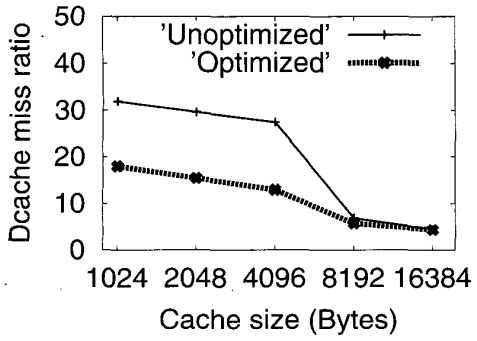

Figure 11: Variation of data cache miss ratio with cache size for FFT example

In Figure 11, we present the variation of the performance with increasing cache size keeping the line size constant for the FFT application. Note that beyond a cache size of 8 KB, there is no significant improvement in performance. This is expected, since, as mentioned in Section 3, if the cache is large enough to accommodate *all* the application data, then there is no useless data fetched into the cache; there are no capacity misses and the cache can be replaced by a simple on-chip RAM that stores the data. Our data restructuring optimization shows significant performance improvement when the cache sizes are relatively small compared to the total data size, which is the more realistic case, and especially useful in embedded applications, where chip area can be saved by using a smaller cache combined with aggressive compiler optimizations such as the one presented here.

6. CONCLUSION AND FUTURE WORK

We presented an algorithm for transforming the memory layout of aggregate data structures so as to improve the cache performance of embedded applications. The aggressive transformations are especially suited for embedded system design because we can assume that the entire application is visible to the compiler, thus making it possible to verify the safety of the data transformation operation. We presented an algorithm for automatically inferring the best clustering of individual fields of aggregate data structures such as arrays of `structs` based on the memory access patterns of the given application. Our experiments verify the soundness of the approach.

One area of refinement/improvement of the algorithm is the situation where an array is accessed with different index expressions in the same loop (e.g, `a[i]` and `a[2i]`). Currently, we consider array accesses to be correlated if there is at least one correlated access. We consider it beneficial to merge arrays `a` and `b` if accesses `a[i]`, `a[2i]`, `b[i]`, and `b[3i]` exist in a loop. However, this ignores the possible conflicts among the remaining accesses. The cost function could be refined to handle such cases. In addition, the greedy clustering algorithm could be improved by using better partitioning algorithms reported in CAD literature in various contexts.

7. REFERENCES

[1] The Open SystemC Initiative, http://www.systemc.org.

[2] The SUIF Compiler System, http://suif.stanford.edu.

[3] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, Dordrecht, June 1998.

[4] R. Ernst, J. Henkel, and T. Benner. Hardware-software co-synthesis of microcontrollers. *IEEE Design and Test of Computers*, 11(4):64-75, Dec. 1994.

[5] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[6] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, Boston, 1995.

[7] P. E. R. Lippens, J. L. van Meerbergen, W. F. J. Verhaegh, and A. van der Werf. Allocation of multiport memories for hierarchical data streams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 728-735, Santa Clara, CA, Nov. 1993.

[8] P. R. Panda, N. D. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):384-409, Oct. 1997.

[9] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, 1999.

[10] L. Séméria, K. Sato, and G. D. Micheli. Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C. In *Proceedings Design Automation and Test in Europe (DATE'00)*, pages 312-319, Paris, France, Mar. 2000.

[11] L. Séméria and G. D. Micheli. Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C. *IEEE Transactions on Computer Aided Design*, 20(2):213-233, Feb. 2001.

[12] S.Wuytack, J. L. da Silva, F.Catthoor, G. Jong, and C. Ykman-Couvreur. Memory management for embedded network applications. *IEEE Transactions on Computer Aided Design*, 18(5):533-544, May 1999.

[13] R. Wilson and M. Lam. Efficient context-sensitive analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1-12, La Jolla, CA, June 1995.