Data and Memory Optimization Techniques for Embedded Systems

P.Panda[1] F.Catthoor[2,3] N.Dutt[4] K.Danckaert[2] E.Brockmeyer[2] C.Kulkarni[2]
A.Vandecappelle[2]
P.G.Kjeldsberg[5]

[1] Synopsys Inc., 700 E. Middlefield Rd, Mountain View, CA, 94043, U.S.A
[2] IMEC, Kapeldreef 75, Leuven, Belgium
[3] Katholieke Univ. Leuven, Belgium
[4] Center for Embedded Computer Systems, University of California, Irvine, CA 92697, U.S.A
[5] NTNU, Trondheim, Norway

Abstract

We present a survey of the state-of-the-art techniques used in performing data and memory-related optimizations in embedded systems. The optimizations are targeted directly or indirectly at the memory subsystem, and impact one or more out of three important cost metrics: area, performance, and power dissipation of the resulting implementation.

We first examine architecture-independent optimizations in the form of code transformations. We next cover a broad spectrum of optimization techniques that address memory architectures at varying levels of granularity, ranging from register files to on-chip memory, data caches, and dynamic memory (DRAM). We end with memory addressing related issues.

1 Introduction

In the design of embedded systems, memory issues play a very important role, and often impact significantly the embedded system's performance, power dissipation, and overall cost of the implementation. Indeed, as new processor families and processor cores begin to push the limits of high performance, the traditional processor-memory gap widens and often becomes the dominant bottleneck in achieving high performance. While embedded systems range from simple micro-controller based solutions to high-end mixed hardware/software solutions, embedded system designers need to pay particular attention to issues such as minimizing memory requirements, improving memory throughput, and limiting the power dissipated by the system's memories.

Traditionally, much attention has been paid to the role of memory system design in the compiler, architecture and CAD domains. Many of these techniques, while applicable to some extent, do not not exploit fully the optimization opportunities present in embedded system design. From an application viewpoint, embedded systems are special-purpose, and are thus amenable to aggressive optimization techniques that can fully utilize knowledge of the applications. Whereas many traditional memory-related hardware and software optimizations had to account for variances due to general purpose applications, memory optimizations for embedded systems can be tailored to suit the expected profile of code and data. Furthermore, from an architectural viewpoint, the embedded system designer pays great attention to the customization of the memory subsystem (both on-chip, as well as off-chip): this leads to many non-traditional memory organizations, with a standard cache hierarchy being only one of many memory architectural options. Finally, from a constraint viewpoint, the embedded system designer needs to meet not only system performance goals, but also has to do this within a power budget (especially for mobile applications), and meet real-time constraints. The system performance should account for not only the processor's speed but also the system bus load to the shared board-level storage units such as main memory and disk. Even the L2 cache is shared in a multi-processor context. As a result of all this, the memory and bus subsystem costs become a significant contributor to overall system costs, and thus the embedded system designer attempts to minimize memory requirements with the goal of lowering overall system costs.

In this survey, we present a variety of optimization techniques for data and memory used in embedded systems. We begin in Section 2 with a survey of several global optimizations that are independent of the target architectural platform, and which, more often than not, always result in improved performance, cost and power. These optimizations take the form of source-to-source code transformations that precede many traditional compiler and synthesis steps, and which move the design to a superior starting point in the design space exploration of alternative embedded system realizations.

Next, in Section 3, we discuss optimization opportunities in the context of specific memory modules, customized memory architectures, and their use in both a hardware (or behavioral) synthesis context, as well as in a software (or traditional compiler) context. This section progresses from optimization techniques applied to memory elements closest to the computational engines – registers and register files – and then discusses optimization techniques for increasingly distant memory structures: SRAM, cache, and DRAM. We survey approaches in the modeling of these disparate memory structures, their customization and their optimization.

Finally, in Section 4, we survey memory address generation technqiues. An important byproduct of applying both platform-independent, as well as memory architecture-specific optimizations, is that the memory accesses undergo a significant amount of transformation from the original source code. Thus attention must be paid to effective generation of the target memory addresses, implemented either as code running on a programmable processor, or as data consumed by a variety of hardware and software engines.

Since this survey covers primarily data-related optimizations, we do not address in detail techniques that are specific to instructions, instruction caches, etc. However, we point out analogous optimizations that apply to instructions in relevant sections.

2 Platform-Independent Code Transformations

The importance of performing loop control flow transformations prior to the memory organisation related tasks has been recognized quite early in compiler theory (for an overview see [11]) and the embedded system synthesis domain [150]; it follows that if such target architecture independent transformations are not applied, the resulting memory organisation will be heavily suboptimal. In this section, we will examine the role of source-to-source code transformations in the solution to the data transfer and storage bottleneck problem. This is especially important for embedded applications where performance is not the only goal; cost issues such as memory footprint and power consumption are also crucial. The fact that execution speed and energy for a given application form at least partly different objective functions that require different optimisation strategies in an embedded context has been conclusively shown since 1994 (for example, see early work in [158] and [101]). Even in general-purpose processors they form different axes of the exploration space for the memory organisation [18].

Many of these code transformations can be carefully performed in a platform-independent order [22, 30]. This very useful property allows us to apply them to a given application code without having any prior knowledge of the platform architecture parameters such as memory sizes, communication scheme, and datapath type. The resulting optimized code can then be passed through a platform-dependent stage to obtain further cost-performance improvements and trade-offs. We will see in subsequent sections that the optimisations are especially useful when the target architecture has a customizable memory organisation.

We first discuss global (data-flow and) loop transformations in Section 2.1. This will be followed by data reuse related transformations in Section 2.2. Finally, in Section 2.3 we study the link with and the impact on processor partitioning and parallelisation. A good overview of research on system-level transformations can be found in [21] and [17], with the latter focussing on low power techniques. In the following subsections we limit ourselves to discussion of the most directly related work.

2.1 Code rewriting techniques for access locality and regularity

Code rewriting techniques, consisting of loop (and sometimes also data flow) transformations, are an essential part of modern optimizing and parallelizing compilers. They are mainly used to enhance the temporal and spatial locality for cache performance, and to expose the inherent parallelism of the algorithm to the outer (for asynchronous parallelism) or inner (for synchronous parallelism) loop nests [4, 157, 11]. Other application areas are communication-free data allocation techniques [25] and optimizing communications in general [57].

Most work has focused on interactive systems, with very early work since end of the 70's [91]. Environments like Tiny [156], Omega at U.Maryland [70], SUIF at Stanford [4, 61], the Paradigm compiler at Univ. of Illinois [12] (and earlier work [122]) and the ParaScope Editor [98] at Univ. of Rice are representative of this large body of work.

In addition, research has been performed on (partly) automating the steering of these loop transformations. Many transformations and methods to steer them have been proposed which **increase the parallelism**, in several contexts. This has happened in the array synthesis community (e.g. at Saarbrucken [143] (mainly intended for interconnect reduction in practice), at Versailles [41] and E.N.S.Lyon [33] and at the Univ. of SW Louisiana [133]) in the parallelizing compiler community (e.g. at Cornell [87], at Illinois [110], at Stanford [155, 4], at Santa Clara [132]), and finally also in the high-level synthesis community (at Univ. of Minnesota [118] and Univ. of Notre-Dame [119]).

Efficient parallelism is however partly coupled to **locality of data access** and this has been incorporated in a number of approaches. Examples are the work on data and control-flow transformations for distributed shared-memory machines at the Univ. of Rochester [26], or heuristics to improve the cache hit ratio and execution time at the Univ. of Amherst [99]. Rice Univ. has recently also started looking the actual memory bandwidth issues and the relation to loop fusion [39]. At E.N.S.Lyon the effect of several loop transformation on memory access has been studied too [42]. A quite broad transformation framework including interprocedural analysis has been proposed in [100]. It is focused on parallelisation on a shared memory multiprocessor. The memory related optimisations are still performed on a loop nest basis (so still "local") but the loops in that loop nest may span different procedures and a fusing preprocessing step tries to combine all compatible loop nests that do not have dependencies blocking their fusing.

It is thus no surprise that these code rewriting techniques are also very important in the context of data transfer and storage (DTS) solutions, especially for embedded applications that permit customized memory organisations. As the first optimization step in the design methodology proposed in [44, 35, 96], they are able to significantly reduce the required amount of storage and transfers and improve the access behaviour thus enabling the ensuing steps of more platform-dependent optimisations. As such, the global loop transformations mainly increase the locality and regularity of the accesses in the code. In an embedded context this is clearly good for memory size (area) and memory accesses (power) [44, 35] but of course also for pure performance [96], even though the two objectives do not fully lead to the same loop transformation steering. The main distinction from the vast amount of earlier related work in the compiler literature, is however that they perform these transformations across all loop nests in the entire program [44]. Traditional loop optimizations performed in compilers, where the scope of loop transformations is limited to one procedure or usually even one loop nest, can enhance the locality (and parallelization possibilities) within that loop nest, but it may not solve the global data flow and associated buffer space needed between the loop nests or procedures. A recent transformation framework including interprocedural analysis proposed in [100] is a step in this direction: it is focused on parallelisation for a shared memory multiprocessor. The memory related optimisations are still performed on a loop nest basis (and are thus"local") but the loops in that loop nest may span different procedures and a fusing preprocessing step tries to combine all compatible loop nests that do not have dependencies blocking their fusing. The goal of the fusing is primarily to improve parallelism.

The global loop and control flow transformation step proposed in [35, 44, 96] can be viewed as a precompilation phase, applied prior to conventional compiler loop transformations. This preprocessing also enables later memory customization steps such as memory hierarchy assignment, memory organization, and in-place mapping (Section 2.2) to arrive at the desired reduction of storage and transfers. A global data flow transformation step [20] can be applied that modifies the algorithmic dataflow to remove any redundant data transfers typically present in many practical codes. A second class of global data flow transformations also serves as enabling transformations for other steps in an overall platform-independent code transformation methodology by breaking data-flow bottle-necks [20]. However, that topic will not be further elaborated in this survey.

In this section, we first discuss a simple example to show how loop transformations can significantly reduce the data storage and transfer requirements of an algorithm. Next, we illustrate how this step can be automated in a tool.

Example:

Consider the following code, where the first loop produces an array b[], and the second loop reads b[] and another array a[] to produce an update of the array b[]. Only the b[] values have to be kept afterwards.

```
for (i=0; i<N; ++i)
  for (j=0; j<=N-L; ++j)
```

```
    b[i][j] = 0;
for (i=0; i<N; ++i)
  for (j=0; j<=N-L; ++j)
    for (k=0; k<L; ++k)
      b[i][j] += a[i][j+k];
```

Should this algorithm be implemented directly, it would result in high storage and bandwidth requirements (assuming that N is large), since all b[] signals have to be written to an off-chip background memory in the first loop, and read back in the second loop. Rewriting the code using a loop merging transformation, gives the following:

```
for (i=0; i<N; ++i)
  for (j=0; j<=N-L; ++j)
    b[i][j] = 0;
    for (k=0; k<L; ++k)
      b[i][j] += a[i][j+k];
  }
```

In this transformed version, the b[] signals can be stored in registers up to the end of the accumulation, since they are immediately consumed after they have been produced. In the overall algorithm, this reduces memory bandwidth requirements significantly since L is typically small.

□

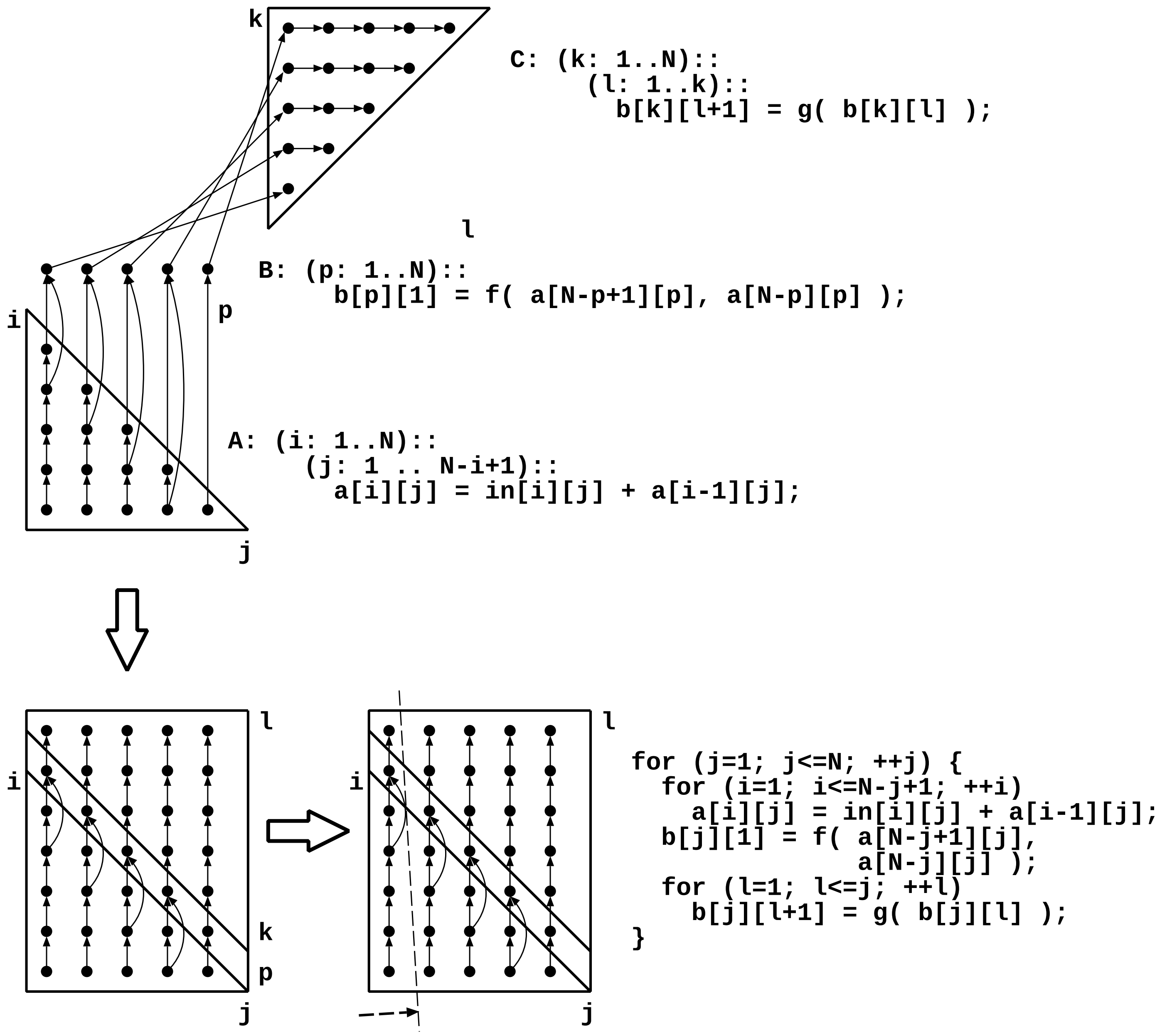
A few researchers have addressed automation of the loop transformations described above. Most of this work has focussed solely on increasing the opportunities for parallelisation (for early work see e.g. [110, 155]). Efficient parallelism is however partly coupled to **locality of data access** and this has been incorporated in a number of approaches. Partitioning or blocking strategies for loops to optimize the use of caches have been studied in several flavours and contexts (see e.g. [79, 94]). In an embedded context also the memory size and energy angles have been added to this, as illustrated by the early work in [44, 158, 35] to increase locality and regularity globally, and more recently also in e.g. [42] and [68]. In addition, memory access scheduling has a clear link also to certain loop transformations to reduce the embedded implementation cost. This is illustrated by the work on local loop transformations to reduce the memory access in procedural descriptions [77], the work on multi-dimensional loop scheduling for buffer reduction [120], and the Phideo project where "loop" transformations on periodic streams have been applied to reduce an abstract storage and transfer cost [153].

To automate the proposed loop transformations, the approach of [44, 31] makes use of a polytope model [43, 21]. In this model, each n-level loop nest is represented geometrically by an n-dimensional polytope. An example is given in Figure 1, where the loop nest at the top is two-dimensional and has a triangular polytope representation, because the inner loop bound is dependent on the value of the outer loop index. The arrows in the Figure represent the data dependencies; they are drawn in the direction of the data flow. The order in which the iterations are executed can be represented by an ordering vector which traverses the polytope. To perform global loop transformations, a two-phase approach is used. In the first phase, all polytopes are placed in one common iteration space. During this phase, the polytopes are merely considered as geometrical objects, without execution semantics. In the second phase, a global ordering vector is defined in this global iteration space. In Figure 1, an example of this methodology is given. At the top, the initial specification of a simple algorithm is shown; at the bottom left, the polytopes of this algorithm are placed in the common iteration space in an optimal way, and at the bottom right an optimal ordering vector is defined and the corresponding code is derived.

Most existing loop transformation strategies work directly on the code. Moreover, they typically work on single loop nests, thereby omitting the global transformations which are crucial for storage and transfers. Many of these techniques also consider the body of each loop nest as one union [33], whereas in [43] each statement is represented by a polytope, which allows more aggressive transformations. An exception to the "black box" view on the loop body is formed by the "affine-by-statement" [32] techniques which transform each statement separately. However, the two-phase approach still allows a more global view on the data transfer and storage issues.

2.2 Code rewriting techniques to improve data reuse

When the system's memory organization includes a memory hierarchy, it is particularly important to optimize data transfers and storage so as to efficiently utilize the memory hierarchy. This can be achieved by optimizing the



```
C: (k: 1..N)::
     (l: 1..k)::
       b[k][l+1] = g( b[k][l] );
```

```
B: (p: 1..N)::
     b[p][1] = f( a[N-p+1][p], a[N-p][p] );
```

```
A: (i: 1..N)::
     (j: 1 .. N-i+1)::
       a[i][j] = in[i][j] + a[i-1][j];
```

```
for (j=1; j<=N; ++j) {
  for (i=1; i<=N-j+1; ++i)
    a[i][j] = in[i][j] + a[i-1][j];
  b[j][1] = f( a[N-j+1][j],
               a[N-j][j] );
  for (l=1; l<=j; ++l)
    b[j][l+1] = g( b[j][l] );
}
```

Figure 1. Example of automatable loop transformation methodology.

data transfers and storage in the code to expose maximally the data reuse possibilities. The compiler literature up to now focused on improving the data reuse by performing loop transformations (see above). But in addition to that (important) step, we can also positively influence the behaviour of the program code on a processor with a memory hierarchy by explicitly adding copies of subsets of the data in the source code. As far as we know, no formal technique has been published on where to add such extra loop nests and data copies. So code rewriting techniques, consisting of loop and data flow transformations, are essential as a preprocessing step to achieve this, because they significantly improve the overall regularity and access locality of the code. This enables the next step of the platform-independent transformation flow, namely the data reuse decision step, to arrive at the desired reduction of storage and transfers. During this step, hierarchical data reuse copies are added to the code, exposing the different levels of reuse which are inherently present (but not directly visible) in the transformed code. This differs from a conventional approach where after the loop transformation preprocessing, the hardware cache control determines the size and "time" of these copies based on the available locality of access. In the approach of [38] a global exploration of the data reuse copies is performed to globally optimize the size and timing of these copies in the code. A custom memory hierarchy can then be designed on which these copies can be mapped in a very efficient way (see e.g. [159]). However, even for a predefined memory hierarchy as typically present in a programmable processor context, the newly derived code from this step implicitly steers the data reuse decisions and still results in a major benefit on system bus load, system power budget, and cache miss behaviour (see e.g. [80]). This compile-time exploration of data reuse and code modification appears to be a unique approach not investigated elsewhere.
Example:
Consider the following example, that has already undergone the loop transformations discussed in the previous subsection:

```
for (i=0; i<N; ++i)
  for (j=0; j<=N-L; ++j) {
    b[i][j] = 0;
    for (k=0; k<L; ++k)
      b[i][j] += a[i][j+k];
  }
```

When this code is executed on a processor with a small cache, it would perform much better than the initial code. To map it on a custom memory hierarchy however, the designer has to know the optimal size of the different levels of this hierarchy. To this end, signal copies (buffers) are added to the code in order to make the data reuse explicit. For the example, this results in the following code (the initialization of *a_buf*[] has been left out for simplicity):

```
int a_buf[L];
int b_buf;
for (i=0; i<N; ++i)
  {initialize a_buf}
  for (j=0; j<=N-L; ++j) {
    b_buf = 0;
    a_buf[(j+L-1)%L] = a[i][j+L-1];
    for (k=0; k<L; ++k)
      b_buf += a_buf[(j+k)%L];
    b[i][j] = b_buf;
  }
```

In this code, two data reuse buffers are present:

- *a_buf*[] (L words), for the a[][] signals
- *b_buf* (1 word), for the b[][] signals

□

In the general case, more than one level of data reuse buffers is possible for each signal. A formal methodology, where all possible buffers are arranged in a tree, is described in [159] . For each signal, such a tree is generated, and an optimal alternative is selected.

2.3 Relation between task and data-parallelism

Parallelization is a standard technique used to improve the performance of a system by using multiple processing units operating simultaneously. However, for a given piece of code, the system's performance can vary widely and there does not appear to be a straightforward solution for effective parallelization. This is also true with respect to the impact of parallelization on data storage and transfers. Most of the research effort in the compiler/architecture domain addresses the problem of parallelization and processor partitioning [4, 108, 122]. Data communication between processors is usually taken into account in more recent methods [1] but they use an abstract model (i.e. a virtual processor grid, which has no relation with the final number of processors and memories). Furthermore, these techniques typically use execution speed as the only evaluation metric. However, in embedded systems, power and memory size are also important, and thus different strategies for efficient parallelization have to be developed. A first approach for more global memory optimization in a parallel processor context was described in [29, 95], where the authors describe an extensive precompiler loop reorganization phase prior to the parallelization steps.

Two important parallelization alternatives are task and data parallelization. In task parallelization, the different subsystems of an application are assigned to different processors. In data parallelization, each processor executes the whole algorithm, but only on a part of the data. Hybrid task-data parallel alternatives are also possible. When data transfer and storage optimization is an issue, even more attention has to be paid to the way in which the algorithm is parallelized. The authors of [29, 95] have explored this on several realistic demonstrator examples, among which is a Quadtree Structured Difference Pulse Code Modulation (QSDPCM) application. QSDPCM is an interframe compression technique for video images. It involves a motion estimation step, and a quadtree based encoding of the motion compensated frame-to-frame difference signal. Table 1 shows an overview of the achieved results when 13 processors are used as a target, using pure data as a baseline. The estimated area and power figures have been obtained using a proprietary model from Motorola. From this table, it is clear that the rankings for the different alternatives (initial and transformed) are clearly distinct. For the transformed description, the task level oriented hybrids are better. This is true because these kinds of partitionings keep the balance between double buffers (present in task level partitionings) and replicates of array signals with the same functionality in different processors (present in data level partitionings). However it is believed that the optimal partitioning depends highly on the number of the different submodules of the application and on the number of processors that will be used.

Version	**Partitioning**	**Area**	**Power**
Initial	Pure data	1	1
	Pure task	0.92	1.33
	Modified task	0.53	0.64
	Hybrid 1	0.45	0.51
	Hybrid 2	0.52	0.63
Transformed	Pure task	0.0041	0.0080
(by loop and)	Modified task	0.0022	0.0040
(data reuse)	Hybrid 1	0.0030	0.0050
(decisions)	Hybrid 2	0.0024	0.0045

Table 1. Overall results for data memory related cost exploration in QSDPCM.

With regard to the memory size required for the storage of the intermediate array signals is concerned, the results of the partitionings based on the initial description prove that this size is reduced when the partitioning becomes more data oriented. Initially this size is smaller for the first hybrid partitioning (245 K) which is more data oriented than the second hybrid partitioning (282 K) and the task-level partitioning (287 K). However, this can change after the transformations are applied. In terms of the number of memory accesses to the intermediate signals the situation is simpler. The number of accesses to these signals always decreases as the partitioning becomes more data oriented. The table also shows the huge impact which this platform-independent transformation stage can have on highly data-dominated applications like this video coder. Experiments on several processor platforms for different demonstrators [30] have shown the importance of applying these optimizations.

2.4 Dynamic Memory Allocation

Embedded system designers typically use a C/C++ based design environment in order to model embedded systems at a suitably high level of abstraction. At this level, designers may use complex programming constructs that are not well understood by hardware synthesis tools. Hardware Description Languages (HDLs) such as VHDL and Verilog offer the array data structure as a means for specifying logical memories. However, the modeling facilities offered by HDLs are increasingly inadequate for system-level designers, who need the full expressive power of high-level modeling languages. One such useful feature is dynamic memory allocation. A system description may dynamically allocate and free memory using the `new/delete` operators and `malloc/free` function calls. Although the tasks implied by these constructs were originally intended to be performed by an operating system, it is possible for a hardware synthesis tool to translate them into reasonable hardware interpretations.

In [160] the authors describe a system where dynamic data types are specified at a very high abstraction level (such as association tables); these abstract data types are then refined into synthesizable hardware through two main phases. In a first main phase they are refined to concrete data structures [160]. For instance an association table with two access keys can be refined into a three level data structure where the first level is a linked list, the second a binary tree and the third one a pointer array. Each of these 3 levels is accessed by subkeys that are repartitioned from the original keys. An automated technique for that exploration is proposed in [162]. In a second main phase, dynamic allocation and freeing duties are performed by a *virtual memory manager* which performs the typical tasks involved in maintaining the free list of blocks in memory: keeping track of free blocks, choosing free blocks, freeing deleted blocks, splitting and merging blocks [154]. An exploration technique is proposed in [34], in which different memory allocators are generated for different data types. Following this, a *basic group splitting* operation splits the memory segment into smaller *basic groups* to increase the allocation freedom by, for instance, splitting an array of structures into its constituent fields. These logical memory segments are then mapped into physical memory modules in the Storage Bandwidth Optimization (SBO) step as described in Section 3.2.

In [130], an approach at a lower abstraction level is proposed. It is specifically targeted to a hardware synthesis context and assumes that the virtual memory managers are already fixed. So the outcome of the above approach can be directly used as input for this step. Here, the actual number and size of the memory modules are specified by the designer, along with a hint of which `malloc` call is targeted at which memory module. A general purpose memory allocator module which performs the block allocation and freeing tasks is also instantiated for each memory module. However, the allocator can be optimized and simplified when the size arguments to all `malloc` calls for a single module are compile-time constants and when constant-size data is allocated and freed within the same basic block. In the latter case, the dynamic allocation is replaced by a static array declaration.

2.5 Memory Estimation

Estimation techniques that assess the memory requirements of an application are critical for helping the system designer select a suitable memory realization. At the system level, no detailed information is available about the size of the memories required for storing data in alternative realizations of an application. To guide the designer and help in choosing the best solution, estimation techniques for the storage requirements are therefore needed very early in the system design trajectory. For data dominant applications, the high-level description is typically characterized by large multi-dimensional loop nests and arrays. A straightforward memory size estimate can be computed by multiplying the dimensions of individual arrays and summing up the sizes of different arrays. However, this could result in a huge overestimate, since not all the arrays, and certainly not all parts of one array, are alive at the same time. In this context an array element, also denoted a signal, is alive from the moment it is written, or produced, and until it is read for the last time. This last read is said to consume the element [3]. Since elements with non-overlapping lifetimes can share the same physical memory location (the *in-place mapping problem* [150]), a more accurate estimate has to account for mapping arrays and parts of arrays to the same place in memory. To what degree it is possible to perform in-place mapping, depends heavily on the order in which the elements in the arrays are produced and consumed. This is mainly determined by the execution ordering of the loop nests surrounding the instructions accessing the arrays.

At the beginning of the design process, little information about the execution order is known. Some is given from the data dependencies between the instructions in the code and the designer may restrict the ordering for example due to I/O constraints. In general however, the execution order is not fixed, giving the designer considerable freedom in the implementation. As the process progresses, the designer takes decisions that gradually fix the ordering, until the full execution ordering is known. To steer this process, estimates of the upper and lower bounds on the storage requirement are needed at each step, given the partially fixed execution ordering.

The storage requirements for scalar variables can be determined by a clique partitioning formulation for performing register allocation (described in Section 3.1.1). However, such techniques break down when used for large multi dimensional arrays, due to the huge number of scalars present when each array element is treated as a scalar. To overcome this shortcoming, several research teams have tried to split the arrays into suitable units before or as a part of the estimation. Typically each instance of array element accessing in the code is treated separately. Due to the code's loop structure, large parts of an array can be produced or consumed by the same code instance. This reduces the number of elements the estimator must handle compared to the scalar approach.

In [151], a production time axis is used to find the maximum difference between the production and consumption time for any two depending instances, giving the storage requirement for one array. The total storage requirement is the sum of the requirements for each array. Only in-place mapping internally to an array is considered, not the possibility of mapping arrays in-place of each other. In [52] the data dependency relations between the array references in the code are used to find the number of array elements produced or consumed by each assignment. From this, a memory trace of upper and lower bounding rectangles as a function of time is found with the peak bounding rectangle indicating the total storage requirement. If the difference between the upper and lower bounds for this critical rectangle is too large, the corresponding loop is split into two and the estimation is rerun. In the worst-case situation a full loop unrolling is necessary to achieve a satisfactory estimate, which can become expensive. [163] describes a methodology based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. They show that it is only necessary to find the number of live variables for one instruction in each innermost loop nest to get the minimum memory size estimate. However, the live variable analysis is performed for each iteration of the loops, which makes it computationally hard for large multi dimensional loop nests. A major limitation for all of these techniques is their requirement of a fully fixed (imperative) execution ordering.

In contrast to the methods described in the previous paragraph, the storage requirement estimation technique presented in [9] does not assume an execution ordering. It starts with an extended data dependency analysis resulting in a number of non-overlapping basic sets of array elements and the dependencies between them. The size of the dependency is the number of elements consumed (read) from one basic set while producing the dependent basic set. The maximal combined size of simultaneously alive basic sets gives the storage requirement.

The high-level estimation methodology described in [74] goes a step further, and takes into account partially fixed execution ordering, achieved by an array data flow analysis preprocessing [40, 123].

Example:

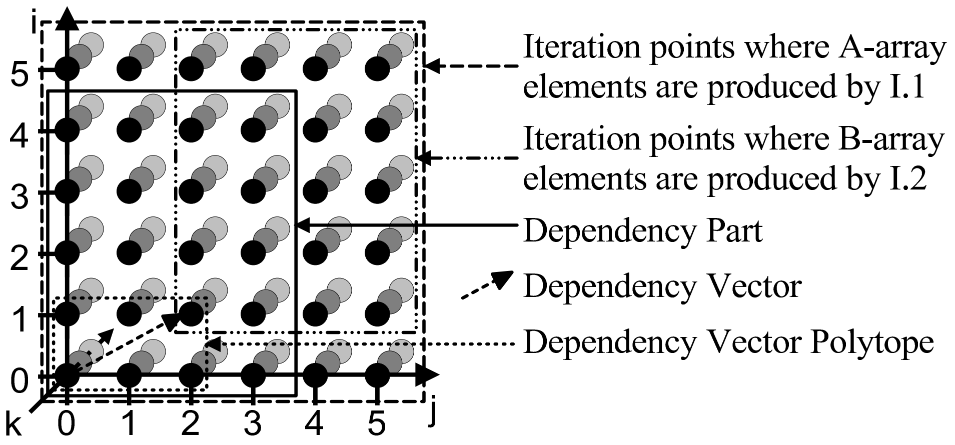
Consider the simple application code example shown in Fig. 2. Two instructions, I.1 and I.2, produce elements of two arrays, A and B. Elements from array A are consumed when elements of array B are produced. This gives rise to a flow type data dependency between the instructions [10].

```
for (i=0; i<=5; i++)
        for (j=0; j<=5; j++)
                for (k=0; k<=2; k++){
I.1             A[i][j][k] = f( in[i][j][k] );
I.2             if ((i > 0) & (j > 1)) B[i][j][k] = g( A[i-1][j-2][k] );
                }
```

Figure 2. Simple application code example in C

The loops around the operations define an iteration space [10], as shown in Fig. 3. Each point within this space represents one execution of the operations inside the loop nest. For our example, at each of these iteration points one A-array element and, when the if clause condition is true, one B-array element is produced. In general not all elements produced by one operation are read by a depending operation. A Dependency Part (DP) is therefore defined containing all the iteration points for which elements are produced that are read by the depending operation. Next, a Dependency Vector (DV) is drawn from any iteration point in the DP producing an array element to the iteration point producing the depending element. Usually this DV is drawn from the point in the DP that is nearest to the origin. Finally, the chosen DV spans a rectangular Dependency Vector Polytope (DVP) in the N-dimensional space with sides parallel to the iteration space axes. The N dimensions of this DVP are defined as Spanning Dimensions (SD). Since the SD normally only comprises a subset of the iterator space dimensions, the remaining dimensions are denoted Nonspanning Dimensions (ND), but this set can be empty. For the DVP in Fig. 3, i and j are SDs while k is ND.

□



Iteration points where A-array elements are produced by I.1
Iteration points where B-array elements are produced by I.2
Dependency Part
Dependency Vector
Dependency Vector Polytope
i
k
j
0
1
2
3
4
5

Figure 3. Iteration space with Dependency Part, Dependency Vector, and Dependency Vector Polytope

Using the concepts presented above, [75] goes into detail regarding the size estimation of individual dependencies. The main contribution is the use of the DP and DVP for calculation of the upper bound and lower bound on the dependency size respectively. As the execution ordering is gradually fixed during the design phases, dimensions and array elements are removed from the DP or added to the DVP to comprise tighter bounds until they converge for a fully fixed ordering. Whether dimensions and array elements are removed from the DP or added to the DVP is in general decided by the partial fixation of spanning and nonspanning dimensions. It has been shown that the size of a dependency is minimized if spanning dimensions are fixed innermost and nonspanning dimensions outermost. Table 1 summarizes estimation results for the dependency in Fig. 3 for a number of partially fixed execution orderings. The results are compared with those achieved with the methodology in [9] where the execution ordering is ignored, and with manually calculated exact results for best-case (BC) and worst-case (WC) ordering.

Fixed Dimension(s)		Lower	Upper	Balasa	Exact
Outermost	Innermost	bound	bound	et al'95	BC/WC
None		4	36	60	6/33
k		4	12	60	6/11
	k,i	31	31	60	31/31
	j	6	14	60	6/14
	i,j	6	6	60	6/6
	k,i,j	6	6	60	6/6

Table 2. Dependency size estimates of simple example (counted in number of scalar dependencies)

In order to achieve a global view of the storage requirement for an application, the combined size of simultaneously alive dependencies must be taken into account [74] but this falls outside the scope of this survey. Application of this approach to the MPEG-4 [106] Motion Estimation Kernel demonstrates how the designer can be guided in applying the critical early loop transformations to the source code. Fig. 4 shows estimates of upper and lower bounds on the total storage requirement for two major arrays. In Step a) no ordering is fixed, giving a large span between the upper and lower bounds. At b) one dimension is fixed outermost in the loop nest with large changes in both upper and lower bounds as results. For Step c) an alternative dimension is fixed outermost in the loop nest. Here the reduction of the upper bound is much larger than in b) while the increase of the lower bound is much smaller. Even with such limited information it is possible for the designer to conclude that the outer dimension used in c) is better than the one used in b). At d) there is an additional fixation of a second outermost dimension with a reduced uncertainty of the storage requirement as a result. Finally at Step e) the execution ordering is fully fixed. The estimation results guide the designer towards an optimized solution.

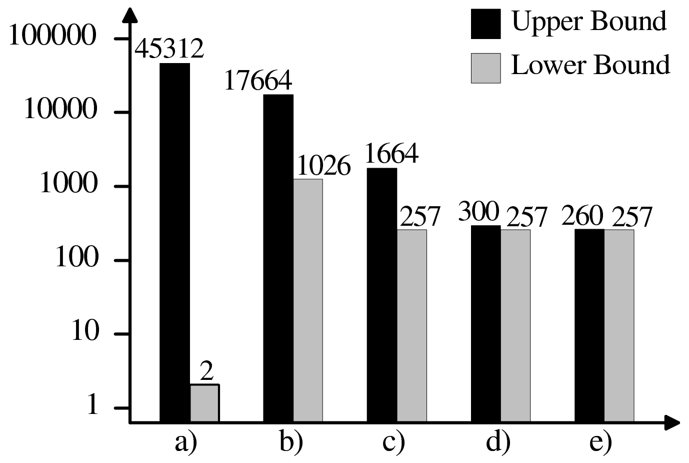


Figure 4. Storage requirement of ME kernel

3 Memory Modeling, Customization, and Optimization

In the previous section, we outlined various source-level transformations that guarantee improved memory characteristics of the resulting implementation irrespective of the target memory architecture. We now survey optimization strategies designed for target memory architectures at various levels of granularity, starting from registers and register files through SRAM, cache, and DRAM.

3.1 Memory Allocation in High Level Synthesis

In this section we discuss several techniques for performing memory allocation in High Level Synthesis (HLS) research. The early techniques generally assumed that the *Scheduling* phase of HLS, which assigns operations in a *Data Flow Graph (DFG)* had already been performed. Following scheduling, all variables that need to be preserved over more than one control step are stored in registers. The consequent optimization problem, called *Register Allocation* [45], is the minimization of the number of registers assigned to the variables because the register count impacts the area of the resulting design.

3.1.1 Register Allocation by Graph Colouring

Early research efforts on the register allocation topic can be ultimately traced back to literature on compiler technology. In [23], the authors present a graph colouring based heuristic for performing register allocation. The *life time* [3] of each variable is first computed, and a graph is constructed whose nodes represent variables and the existence of an edge indicates that the life times are *overlapping*, i.e., they cannot share the same register; a register can be shared only by variables with non-overlapping life times. Thus, the problem of minimizing the register count for a given set of variables and their life times is equivalent to the *graph colouring* problem [46]: assign colours to each node of the graph such that the total number of colours is minimum, and no two adjacent nodes share the same colour. This minimum number is called the *chromatic number* of the graph. In the register allocation problem, the minimum register count is equal to the chromatic number of the graph and each colour represents a different physical register.

Graph colouring is a well known NP-complete problem, so an appropriate approximation algorithm is generally employed. In [148] the register allocation problem is formulated by the *clique partitioning* problem: partition a graph into the minimum number of *cliques* or fully connected subgraphs. This problem is equivalent to graph colouring (if a graph G has a chromatic number χ, then its complement graph G' can be partitioned into a minimum number of χ cliques). Their greedy heuristic initially creates one clique for each node, then proceeds by merging individual cliques into a larger one, at each step selecting for the merge those cliques that have the maximum number of common neighbours.

A polynomial time solution to the register allocation problem was presented in [82]. The authors apply the *left-edge* algorithm to minimize the register count by first sorting the life time intervals of the variables in order of their start times and then making a series of passes, each pass allocating a new register and assigning non-overlapping intervals from the sorted set to the register. This algorithm guarantees the minimum number of registers for straight-line code with no branches, and runs in polynomial time. The register allocation problem has also been formulated as a bipartite graph matching, where the edges are weighted with the expected interconnect cost [63].

Subsequent refinements to the register allocation problem in HLS were based on a higher level of design abstraction – the target architecture was a *Register File* with a fixed number of ports rather than scattered individual registers. A critical problem to solve in the presence of loops was also how to deal with data that exhibited dependencies accross the loop iterations. That was solved by cyclic approaches (see e.g. [48] for an early technique). A good survey of these scalar approaches is provided in [138].

3.1.2 Allocating Scalar Variables to Single and Multiport Memories

A new optimization problem arises when individual registers are replaced by a register file or a memory module. The registers (or memory locations) can no longer be all accessed simultaneously; the number of allowed simultaneous accesses is limited to the number of available ports in the memory. This results in a stronger interaction of the memory allocation decision with the scheduling phase of HLS.

In [7], the authors present a technique to allocate multiport memories in HLS. To exploit the increased efficiency of grouping several registers into a single multiport memory, the technique attempts to merge registers with disjoint access times. While clique partitioning is sufficient to handle the case of a single port memory, a more general framework is needed to handle multiport memories. The technique formulates a 0-1 Linear Programming problem by modeling the port types (read, write, and read/write), the number of ports, and the accesses scheduled to each register in each control step. Since the Linear Programming problem is NP-complete, a branch-and-bound heuristic is employed.

Example:

Consider a scheduled sequence with states S1 and S2 involving three registers R1, R2 and R3 to be mapped into a dual-port memory:

$$S1 : R1 \leftarrow R2 + R3$$
$$S2 : R2 \leftarrow R1 + R1$$

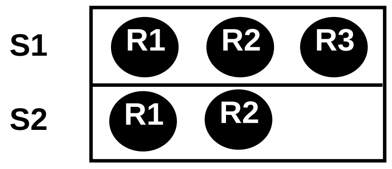


Figure 5. Usage of registers in each control step

The usage of the registers in each control step is shown in Figure 5. To determine which registers to group into the 2-port memory, we need to solve the following problem:

$$\text{maximize } (x_1 + x_2 + x_3)$$

under the constraints

$$x_1 + x_2 + x_3 \leq 2$$
$$x_1 + x_2 \leq 2$$

where x_i is 0 or 1 depending on whether register Ri is assigned to the multiport memory. In this example, the solution is: $x_1 = 1, x_2 = 1, x_3 = 0$. That is, the maximal set of registers that can be assigned to the memory while obeying the constraints is $\{R1, R2\}$. The procedure can then be repeated for assigning the remaining registers to other multiport memories.

□

However, the sequentialization of the memory assignment does not lead to the minimum number of memory modules. In order to minimize the total number of multiport memories, the memories themselves need to be incorporated into the problem formulation. [2] describes MAP, a generalization of the 0-1 Integer Linear Programming (ILP) problem that performs this minimization.

The relative cost benefits of storing data in discrete registers or SRAM modules is performed in [78] in the context of allocating storage for globals signals while synthesizing multiple VHDL processes. Area and performance trade-offs involved in parallel accesses due to storage in discrete registers and sequential accesses due to storage in RAM are performed while generating the clustering solution for registers.

The allocation of scalar variables to register files or multiport memories results in a significant advantage over discrete registers – the interconnect cost of the resulting circuit is reduced drastically. Further, there may be an additional interconnect optimization opportunity when the multiport register file allows us to optionally connect each register to only those ports that are necessary. This decision impacts the number of interconnections between the functional units and memory ports and the related optimization problem is to minimize this number in order to reduce chip area.

Both memory allocation strategies described above propose 0-1 ILP formulations for reducing the interconnect cost. In [72], the memory allocation and interconnect minimization steps are reversed, using the reasoning that the interconnect cost is dominant in determining chip area. In [83], the authors present a method to handle the memory allocation decision during HLS scheduling rather than as a post processing step by weighting the priority function used by the *List Scheduling* algorithm [45] to attempt equal distribution of memory data transfers in the control steps.

3.1.3 Modeling Memory Accesses in HLS

The memory allocation techniques discussed earlier used a simple model of memory accesses: data is read from memory; computations are performed; and data is written back to memory in the same clock cycle. This model works for registers and small registers files. However, when data is stored in a reasonably large on-chip SRAM, the access times are higher and significant compared to the computation time. Accessing memory data may actually require one or more clock cycles. Clearly, the scheduling model of memory accesses needs to be updated to handle this more complex protocol.

The *Behavioral Template* scheduling model presented in [92] offers a way of handling memory accesses in a manner that is consistent with the way other operations are viewed by the scheduler. Every operation is represented by a template; complex operations may take multiple cycles in the template, with different stages representing local scheduling constraints among the stages. Behavioral templates can be used to model memory accesses as shown in Figure 6. The Synopsys Behavioral Compiler tool [142] uses this concept of templates to perform scheduling. Extensions of this modeling methodology to handle more complex memory access protocols (e.g., DRAMs) are presented in Section 3.8.

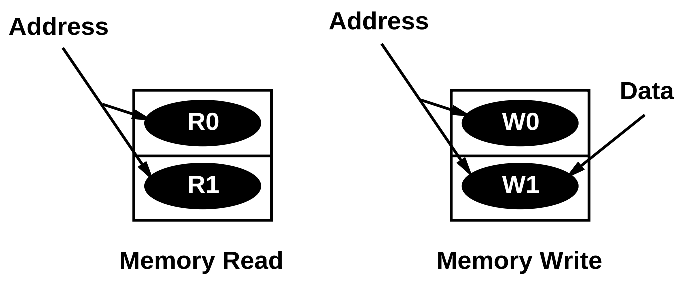


Figure 6. Scheduling template for RAM accesses: 2-cycle memory read and 2-cycle memory write. The address needs to be valid for both cycles. For memory write, the data needs to be ready by the second cycle.

In this section we have surveyed the important techniques used in incorporating memory elements in the behavioral synthesis steps of scheduling and allocation. The allocation methods typically handle only scalar variables. When the specification contains arrays they can be handled automatically only by treating the array as a large collection of scalars. Obviously, this method has its limitations since the computation times increase rapidly when the arrays are large. The next generation of memory optimization techniques discussed in the following subsections focus their attention on large arrays as well as more complex protocols.

3.2 Ordering and Bandwidth Reduction

In many cases, a fully customized memory architecture can give superior memory bandwidth and power characteristics over traditional hierarchical memory architecture that includes data caches. This is particularly true when the application is amenable to detailed compile-time analysis.

Although a custom memory organization has the potential to significantly reduce the system cost, achieving these cost reductions is not trivial, especially manually. Designing a custom memory architecture means deciding how many memories to use and of which type (single-port, dual-port, etc). In addition, the memory accesses have to be ordered in time, so that the real-time constraints (the cycle budgets) are met. Finally, each array must be assigned to a memory, so that arrays can be accessed in parallel as required to meet the real-time constraints [21]. These issues are relevant and have a large impact on the memory bandwidth and power, even when the basic memory hierarchy is fixed, e.g. based on two cache levels and DRAM memory: modern low-power memories [64] allow much customization, certainly in terms of bank assignment (e.g. SDRAMs), sizes (e.g. [93]) or ports (e.g. several modern SDRAMs).

One important factor that affects the cost of the memory architecture is the relative ordering of the memory accesses contained in the input specification. Techniques for optimizing the number of resources given the cycle budget are relevant in the scheduling domain, as are most of the early techniques that operate on the scalar-level (e.g. [121]). Many of these scalar techniques try to reduce the memory related cost by estimating the required number of registers for a given schedule but that does not scale for large array data. The few exceptions that have been published are the stream scheduler in [153, 152], the rotation scheduler in [120] and the percolation scheduler of [109]. They schedule both accesses and operations in a compiler-like context, but with more emphasis on the cost and performance impact of the array accesses and also include a more accurate model of their timing. Only few of them try to reduce the required memory bandwidth, which they do by minimizing the *number* of simultaneous data accesses [152]. They do not take into account *which* data is being accessed simultaneously. Also no real effort is spent to optimize the data access conflict graphs such that subsequent register/memory allocation tasks can do a better job.

Variables in real life applications present a wide variety of access patterns and locality types (for instance scalars, such as indexes, present usually high temporal and moderate spatial locality, while vectors with small stride present high spatial locality, and vectors with large stride present low spatial locality, and may or may not have temporal locality). Several researchers including Gonzalez et.al [56] have proposed splitting a cache into a spatial cache and a temporal cache that store data structures with high temporal and high spatial locality respectively. These approaches rely on a dynamic prediction mechanism to route the data to either the spatial or the temporal caches, based on a history buffer. In an embedded system context, the approach of Grun et al. [55] uses a similar split-cache architecture, but allocates the variables statically to the different local memory modules, avoiding the power and area overhead of the dynamic prediction mechanism. Thus by targeting the specific locality types of the different variables, better utilization of the main memory bandwidth can be achieved. The useless fetches due to locality mismatch are thus avoided. For instance, if a variable with low spatial locality is serviced by a cache with a large line size, a large number of the values read from the main memory will never be used. The approach in [55] shows that the memory bandwidth and memory power consumption can be reduced significantly.

The scheduling freedom among memory accesses can also be exploited to generate memory architectures with lower cost (number of memories) and lower bandwidth (number of ports). This issue is addressed by the Storage Bandwidth Optimization (SBO) technique and the associated Storage Cycle Budget Distribution (SCBD) step in [161].

Example:

Suppose the data flow graph shown in Figure 7(a) has to be scheduled with a time constraint of six cycles. Each access requires one cycle. A satisfying schedule that minimizes the number of simultaneous memory accesses is shown in Figure 7(b). Surprisingly, this schedule leads to a sub-optimal implementation. Figure 7(c) shows a *conflict graph* for this schedule, where each node represents an array and an edge between nodes indicates that the two are being accessed in parallel in some control step. The significance of the edge is that the nodes need to be assigned to different single port memories (analogous to the register allocation problem) or different ports of the same multiport memory, both expensive alternatives. A colouring of the graph reveals a chromatic number of 3, i.e., three single port memories are required to satisfy all the conflicts. However, consider the alternative schedule of Figure 7(e) and the corresponding graph of Figure 7(f). This graph has a chromatic number of 2, resulting in the simpler and lower cost memory assignment of Figure 7(g). This example demonstrates that the relative ordering of the memory accesses

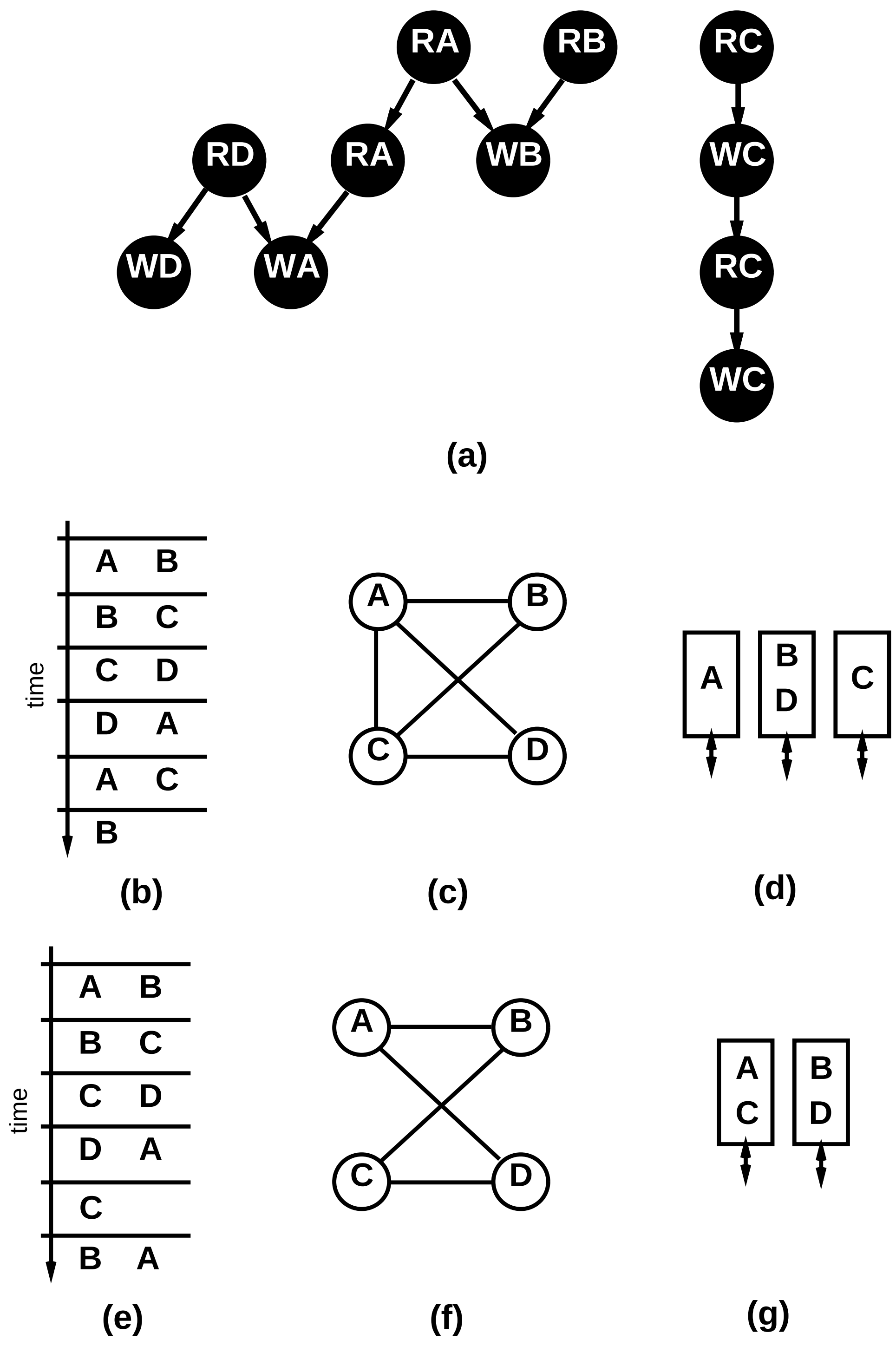


Figure 7. Storage Bandwidth Optimization (a) Data flow graph (b) candidate schedule (c) conflict graph (d) memory assignment (e) alternate schedule (f) new conflict graph (g) new assignment

has a significant impact on the memory cost.

□

The condition of the same array being accessed multiple times in the same control step is represented by self-loops in the conflict graph, and leads to multiport memory implementations. An iterative *conflict-directed ordering* step generates a partial ordering of the CDFG that minimizes the required memory bandwidth.

This SCBD step has to be followed by a memory allocation and signal-to-memory assignment step as described in the next section.

3.3 Memory Packing and Array-to-Memory Assignment

In a custom memory architecture, the designer can choose memory parameters such as the number of memories, and the size, and number of ports on each memory. This decision, which takes into account the constraints derived in the previous section, is the focus of the memory allocation and assignment (MAA) step. The problem can be subdivided into two subproblems. First, memories must be allocated: a number of memories is chosen from the available memory types and the different port configurations, possibly different types are mixed with each other, and some memories may be multi-ported. The dimensions of the memories are however only determined in the second stage. When arrays are assigned to memories, their sizes can be added up and the maximal bit-width can be taken to determine the required size and bit-width of the memory. With this decision, the memory organization is fully determined.

Allocating more or less memories has an effect on the chip area and the energy consumption of the memory architecture (see Fig. 8). Large memories consume more energy per access than small memories, due to the longer word- and bit-lines. Therefore, the energy consumed by a single large memory containing all the data is much larger than when the data is distributed over several smaller memories. Also the area of the one-memory solution is often higher when different arrays have different bit-widths. For example, when a 6-bit and an 8-bit array are stored in the same memory, two bits are unused for every 6-bit word. By storing the arrays in different memories, one 6 bits wide and the other 8 bits wide, this overhead can be avoided.

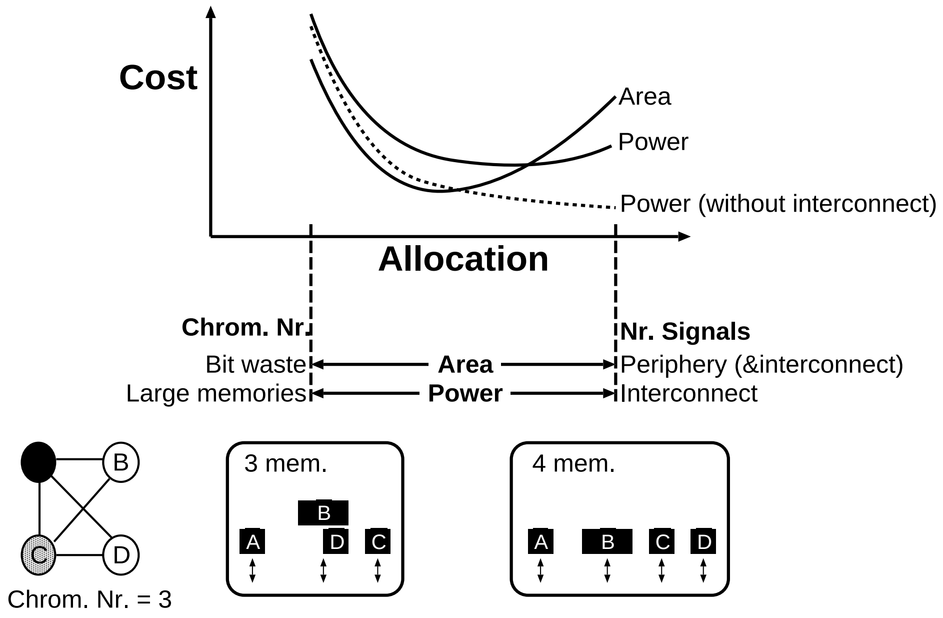


Figure 8. Tradeoff between number of memories and cost during allocation and assignment.

The other end of the spectrum is to store all the arrays in different memories. This also leads to relatively high energy consumption due to the increase in the external global interconnection lines connecting all these (small) memories with each other and with the data-paths. Likewise, the area occupied by the memory system goes up, due to the interconnections and due to the fixed address decoding and other overhead per memory.

Clearly, the interesting memory allocations lie somewhere between the two extremes. The area and the energy function reach a minimum in between, but at different points. The useful exploration region to trade off area with energy consumption lies in between the two minima [19].

The cost of the memory organization depends not only on the allocation of memories, but also on the assignment of arrays to the memories (the discussion above assumes an optimal assignment). When several memories are available, many ways exist to assign the arrays to them. In addition to the conflict cost mentioned earlier, the optimal assignment of arrays to memories depends on the specific memory types used. For example, the energy consumption of some memories is very sensitive to their size, while for others it is not. In the former case, it may be advantageous to accept some wasted bits in order to keep the heavily accessed memories very small, and vice-versa.

A lot of research in recent years has concentrated on the general problem of how to efficiently store data specified in an abstract specification into a given target memory architecture. Both the specification as well as the target architecture tend to be widely varying in different design contexts, leading to many different approaches to solve the problem. The usual assumption is that the embedded environment has no virtual memory system, and the compiler/ synthesis tool can statically assign data to actual memory locations.

The memory packing problem occurs when a set of memories from the designer's point of view (*logical memories*) have to be assigned to a given set of memory modules (*physical memories*) while respecting certain constraints on on the performance of the overall system or minimizing an optimization criterion such as the total delay, area, or power dissipation.

The packing problem was first considered in [69] in the context of an FPGA where the number and access times of the available physical memories is fixed and the logical memories, which are associated with a required data access rate, have to be assigned to these physical memories while satisfying the access time requirements. Mapping multiple logical memories to the same physical memory would, in this formulation, cause a multiplexing of the data access, thereby reducing the actual access rates. For example, if two logical memories are mapped into the same physical memory with access time 10 ns, then the effective access time for both memories would double to 20 ns. This scenario is relevant in stream-based systems.

Example:

Suppose we have an application with the logical memory requirements shown in Figure 9(a): two $2K \times 7$ memories and one $5K \times 8$ memory with the maximum access times of 30 ns and 10 ns respectively. A $2K \times 7$ memory refers to a memory with 2K words with bit width 7. There are two $8K \times 8$ physical memories available on the FPGA.

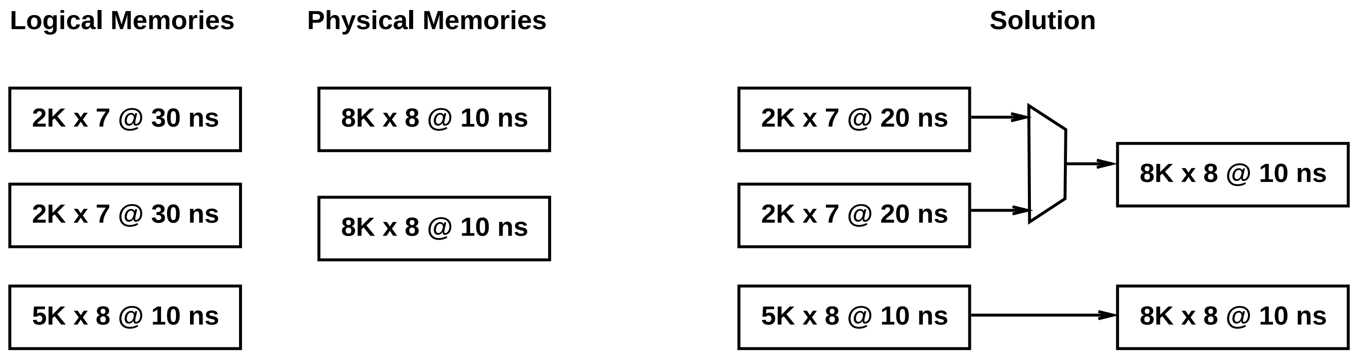


Figure 9. Mapping logical memories to physical while satisfying required access rates

A possible solution is shown in Figure 9(b). The two $2K \times 7$ logical memories are assigned to one $8K \times 8$ physical memory with the effective access time of 20 ns, which satisfies the access time requirements of the system.

□

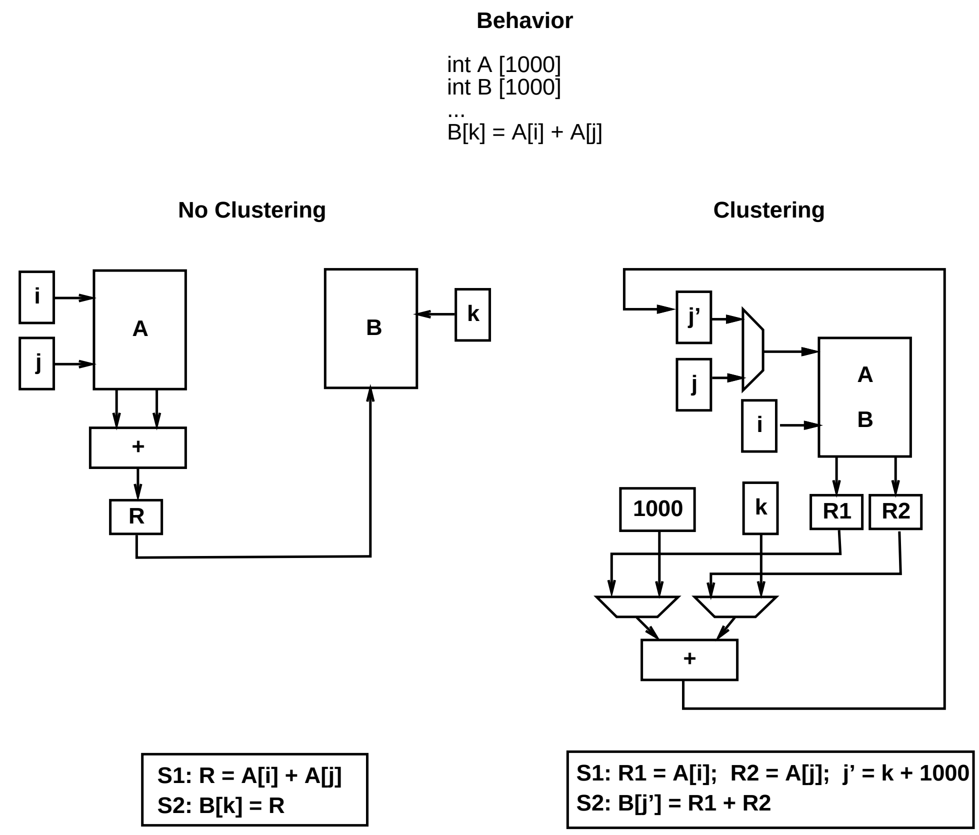
The logical memories would need to be split when either the bit-width or the word count exceeds that of the available physical memories. In the MemPacker utility presented in [69], the memory mapping problem is solved by a branch-and-bound algorithm that attempts to minimize the estimated area of the memory subsystem. The decision tree is pruned when an illegal packing is encountered, i.e., when an assignment violates the access time requirements.

The MemPacker algorithm assumes that the required size and data access times required for the logical memories is known *a priori*. However, this assumption is not always correct. When memory allocation forms part of an automated synthesis framework, the required access times, etc., have to be inferred from the user specification and constraints on the overall system. The MeSA algorithm [125] attempts to integrate the memory allocation step with *array clustering* (grouping of behavioral arrays into the same physical memory) into a behavioral synthesis framework.

Example:

An example behavioral statement and the impact of array clustering on the architecture and schedule is shown in Figure 10. There is an area and performance overhead arising from the additional multiplexers and registers. The MeSA algorithm uses a hierarchical clustering approach and a detailed model of the memory area to evaluate the impact of candidate architectures.

□



```
int A [1000]
int B [1000]
...
B[k] = A[i] + A[j]
```

S1: R = A[i] + A[j]
S2: B[k] = R

S1: R1 = A[i]; R2 = A[j]; j' = k + 1000
S2: B[j'] = R1 + R2

Figure 10. Array clustering in MeSA

A framework for clustering array variables into memories was presented in the ASSASYN tool [128], which recognizes that the packing can actually be done in two dimensions. Apart from mapping arrays into distinct memory addresses of the same memory module, (*vertical concatenation*), two arrays can also be packed so that corresponding elements occupy different bit ranges in the same physical word *horizontal concatenation* if the sum of the required bit-widths of the arrays is less than or equal to the bit-width of the physical memory. An example is shown in Figure 11. *a* and *b* are packed horizontally while *a* and *c* are packed vertically. ASSASYN has at its core a set of *move* transformations that generate candidate architectures in a simulated annealing-based optimization framework.

A general scheme for solving the memory packing problem that takes into account the bit-width, word count, and the number of ports was included in the HLLM (High Level Library Mapping) approach ([65]). The authors present exhaustive solutions as well as linear-time approximations to combine three separate tasks: bit-width mapping, word mapping, and port mapping. In [6], the authors present a technique for reducing memory cost in a *memory selection* algorithm that attempts to combine memory allocation and pipelining in an attempt to reduce memory cost. A hierarchical clustering technique is used here too in order to group DFG nodes into the same memory module.

A recursive strategy of memory splitting (resulting in a distributed assignment) starting from a single port solution for a sequential program has been proposed in [16].

The step of automated Memory Allocation and Assignment (MAA) including the organisation of arrays in the physical memories, before the scheduling or the procedural ordering are fully fixed, has been addressed in [8], and has been coupled to the (extended) conflict graph discussed above (see [136]). The resulting techniques lead to a

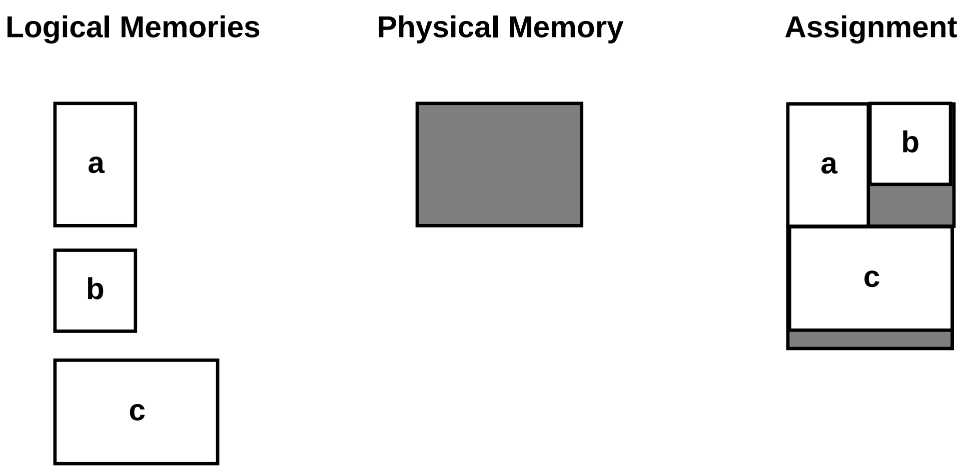


Figure 11. Horizontal and vertical concatenation

significantly extended search space for memory organisation solutions, which is effectively explored in the HIMALAIA tool [149]. An alternative MAA approach based on the conflict graphs is proposed in [135].

3.4 Memory Access time versus Cost Exploration using Pareto Curves

By combining the SCBD and MAA steps of the previous subsections, one can effectively explore and trade off different solutions in the performance, power and area space [18] . Indeed, every step of the SCBD-MAA combination generates a set of valid solutions for a different cycle budget. Therefore it becomes possible to make the right tradeoff within this solution space. Note that without automated tool support, the type of tradeoffs discussed here are not feasible on industrial-strength applications, and thus designers may miss the opportunity of exploring a larger design space.

When the input behavior has a single thread and the goal is to reduce power, the tradeoff can be based solely on the tool output. The given cycle budget defines a conflict graph which can be used for the MAA tool [149]. Obviously, the power and area cost increase when the cycle budget is lowered: more bandwidth is needed which requires multi-port memories (increasing power) or more memories (increases area). This is illustrated on a Binary Tree Predictive Coding (BTPC) application, a lossless or lossy image compression algorithm based on multi-resolution that involves a complex algorithm. The platform-independent code transformation steps [21] discussed in Section 2, are applied manually on this example and the platform-dependent steps (using tools) give accurate feedback about performance and cost [149]. Figure 12 shows the relation between speed and power for the original, optimized and intermediate transformed specifications. The off-chip signals are stored in separate memory components. Four memories are allocated for the on-chip signals. Every step leads to a significant performance improvement without increasing the system cost. For every description, the cycle budget can be traded for system cost, demonstrating the significant effect of the platform-independent code transformation [19].

This performance-power function, when generated per task, can be used to tradeoff cycles assigned to a task at the system level. Assigning too few cycles to a single task causes the entire application to perform poorly. The cycle and power estimates help the designer to assign tasks to processors and to distribute the cycles within the processors over the different tasks [18]. Minimizing the overall power within a processor is possible by applying function minimization on all the power-cycle functions together.

The interaction of the data-path's power/performance with the memory system creates another level of tradeoffs. The assignment of cycles to memory accesses and to the data-path is important for the overall power consumption. A certain percentage of the overall time can be spent in memory accesses and the remaining part to computational issues taking into account the system pipelining (see Figure 13). The data-path cost versus performance tradeoff is well known: when fewer cycles are available, more hardware is needed and more power is consumed (see dotted curve). The tools discussed in [19] can provide the other half of the graph, namely the memory related cost (solid curve). Combining the two leads to an optimized implementation (dashed curve).

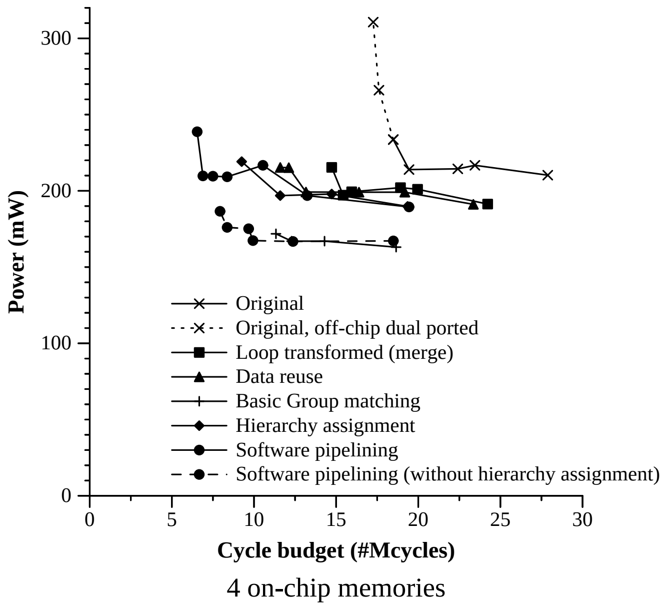


Figure 12. Power versus performance for BTPC example.

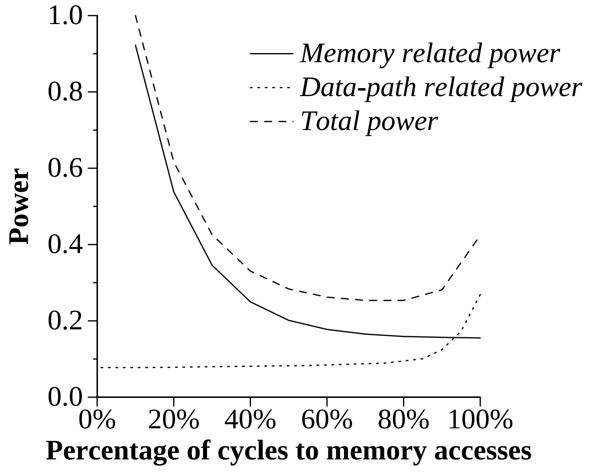


Figure 13. Tradeoff cycles assigned to memory accesses and data-path

3.5 Reducing Bus Transitions for Low Power

Most area and performance optimizations we have discussed so far also indirectly reduce power dissipation. For example, reducing the number of memory modules from two to one not only reduces area, but also reduces power dissipation because the data and address buses, which consist of high capacitance wires, are now fewer and shorter. Similarly, performance optimizations that reduce the number of memory accesses also reduce power as a side effect because of reduced switching on the memory circuitry and address/data buses. However, certain classes of optimizations are targeted explicitly at reducing power dissipation, even at the expense of additional area or performance cost. We discuss techniques that attempt to reduce switching activity on the memory address and data bus while keeping the number of memory accesses unchanged. The extra computation typically introduces a negligible additional switching activity compared to the power savings from reduced bus activity. As observed in [111], typical switching of off-chip buses causes three orders of magnitude more energy than on-chip wires.

Minimization of transition activity on memory buses can be effected by two different approaches: encoding and data organization.

3.5.1 Encoding

The Bus-Invert coding technique described in [137] attempts to decrease the peak power dissipation by encoding the data stream in order to reduce the switching activity on the buses. Although this is more generally applicable to any I/O bus, it is well suited for memory address and data buses. Before placing data on the bus, it is encoded by using an extra control bit that indicates whether the data bits have been inverted.
Example: Suppose the sequence of values on the data bus is:

```
00000000
11111111
00000000
```

This represents the peak power dissipation on the bus because all eight bits transition at once. The Bus-Invert coding introduces a control bit to the bus, which is 1 when the Hamming Distance [76] between successive values is greater than half the bus width. The above sequence is encoded as

```
000000000
000000001
000000000
```

□

The coding scheme incurs an area overhead due to the extra control bit and the encoding and decoding circuitry as well as a possibly small performance overhead due to the computation of the encoded data, but lowers peak power dissipation to the case where only half the number of data bits are toggling, i.e., by 50%. The authors have extended this coding scheme to *limited-weight codes*, which are useful in protocols where data values are represented by the presence of a bit-transition rather than by 1 or 0.

Often, the correlations expected in the memory address and data streams can be exploited to generate intelligent encodings that result in low-power implementations. The most common scenario occurs in the Processor-Memory interactions. High degrees of correlation are observed in the instruction address stream because of the principle of *locality of reference. Spatial locality* of reference occurs when consecutive references to memory result in accesses to nearby data. This is readily observed in the instruction addresses generated by a processor: there is a high probability that the next instruction executed after the current one lies in the next instruction memory location. This correlation can be exploited in various ways to encode the instruction addresses for low power.

To encode the streams of data that are known at design time (e.g. addresses for memories), [158] first proposed a *Gray Coding* technique [76], relying on the well-known observation that the Gray Code scheme results in exactly one bit transition for any two consecutive numbers. Later [140] applied that idea to the instruction stream. [107] proposed a *working-zone encoding*, observing that programs tend to spend a lot of time in small regions of code (e.g., loops). They partition the address bus into two parts: the most significant part identifies the working zone and the least significant part carries an offset in the working zone. This ensures that, as long as the processor executes instructions from the working zone, the most significant bits will never change. An additional control bit is used to identify the case when the address referenced does not belong to the working zone any more. The T0 encoding [15] relies on a similar principle. Here, an additional control bit on the address bus indicates whether the next address is consecutive or not. If it is consecutive, the control line is asserted, and remains so as long as successive instructions executed are consecutive in memory. This is superior to the Gray code because in the steady state, the address bus does not switch at all. In cases where the processor uses the same address bus to address both instruction and data memory, a judicious combination of T0 and Bus-Invert encodings looks promising [14].

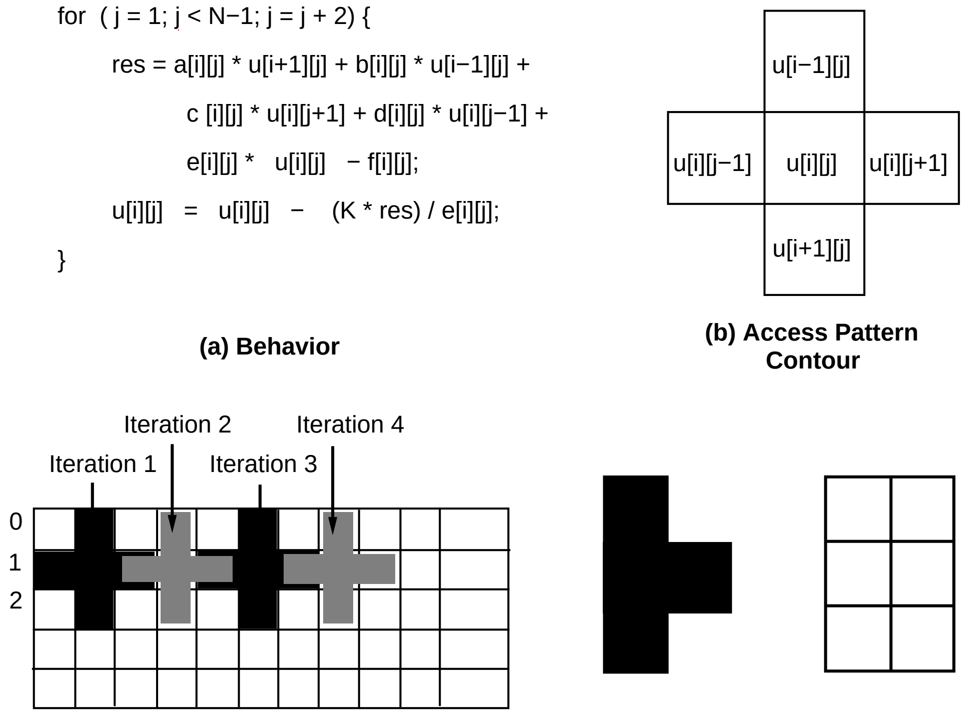
3.5.2 Data Organization

Power reduction through reduced switching activity on the memory address and data bus can also be brought about by the appropriate reorganization of memory data so that consecutive memory references exhibit spatial locality. This locality, if correctly exploited, results in a power-efficient implementation because, in general, the Hamming distance between nearby addresses is less than that between those far apart. This optimization is orthogonal to the encoding optimization discussed earlier. An advantage of the data organization is that, since the analysis is done statically, it obviates the need for an expensive encoding scheme for detecting run-time correlations dynamically. However, data organization can also be combined with an encoding scheme to further reduce switching activity.

A considerable amount of flexibility exists in the actual scheme used to store arrays in memory. For example, two-dimensional arrays can be stored in *row-major* or *column-major* style. In [111], the authors evaluate the impact of three storage schemes: row-major, column-major, and *tile-based*, on the memory address bus switching activity.

```
for  (i = 1;  i < N−1; i++)
    for  ( j = 1; j < N−1; j = j + 2) {
        res = a[i][j] * u[i+1][j] + b[i][j] * u[i−1][j] +
                  c [i][j] * u[i][j+1] + d[i][j] * u[i][j−1] +
                  e[i][j] *   u[i][j]   − f[i][j];
        u[i][j]   =   u[i][j]   −    (K * res) / e[i][j];
    }
```

(a) Behavior



(b) Access Pattern Contour

(c) Inner Loop Execution Trace

(d) New Elements in each Iteration

(e) Tile

Figure 14. Inferring the tile size

Example:

Tile-based storage of array data is illustrated in Figure 14. For the example in Figure 14(a), the memory access trace for array u is shown in Figure 14(b). New elements accessed in each iteration are shown graphically in Figure 14(c). Note that one element is re-used from the previous iteration and can be registered instead of being accessed from memory again. The tile shown in Figure 14(e) is the smallest rectangle enclosing the access pattern of Figure 14(d). Array u can now be stored tile-wise in order to exploit spatial locality.

□

Tile-based storage involves extra complexity in the address generation hardware, but the overall impact is shown to be minimal in [111], in comparison to the large reduction reduction in off-chip bus transition count (experiments in [111] show an average reduction of 63%). An additional optimization opportunity occurs when an ASIC implementation is not limited to a single memory module, but can use several memory modules from a library. In that case, the tile generated above is used to split the arrays into logical partitions, which are then assigned to the physical memories using a bin-packing heuristic.

The idea of ordering data to reduce switching activity on the address and data buses is analogously applicable to instructions. When scheduling flexibility of instructions exists within a basic block, it is possible for the compiler to order the instructions in such a way that the instruction bus switching between access of successive instructions is minimized [145].

3.6 Reducing Memory Size Requirements

One important memory-related optimization in system design is the reduction in the data memory size requirements for an application. This optimization can sometimes be effected by reducing the actual allocated space for temporary arrays by performing an *in-place* mapping of different sections of the logical array into the same physical memory space when the lifetimes of these sections are non-overlapping [21]. We illustrate in-place mapping using an example routine that performs autocorrelation in a linear prediction coding vocoder.

Example:

The initial algorithm is shown below. Two signals `ac-inter[]` and `Hamwind[]`, with respective sizes of 26400 and 2400 integer elements, are responsible for most of the memory accesses and are dominant in size. The loop nest has non-rectangular access patterns dominated by accesses to the temporary variable `ac-inter[]`. Thus to reduce power, we need to reduce the number of memory accesses to `ac-inter[]`. This is possible only by first reducing the size of `ac-inter[]` and then placing this signal in a local memory.

```
for(i6=0;i6<11;i6++) {
    ac-inter[i6][i6] = Hamwind[0] * Hamwind[i6];
    ac-inter[i6][i6+1] = Hamwind[1] * Hamwind[i6+1];
    for(i7=(i6+2);i7<2400;i7++)
        ac-inter[i6][i7] = ac-inter[i6][i7-1] +
            ac-inter[i6][i7-2] + (Hamwind[i7-i6] * Hamwind[i7]);
    AutoCorr[i6] = ac-inter[i6][23991];
}

for(i8=0;i8<10;i8++) {
    v[0][i8] = AutoCorr[i8+1];
    u[0][i8] = AutoCorr[i8];
}
```

`ac inter[]` has a dependency only on two of its earlier values, thus only three (earlier) integer values need to be stored for computing each of the autocorrelated values. Thus by performing intra-signal in-place data mapping, as shown below, we can drastically reduce the size of this signal from 26400 to 33 integer elements.

```
for(i6=0;i6<11;i6++) {
    ac-inter[i6][i6%3] = Hamwind[0] * Hamwind[i6];
    ac-inter[i6][(i6+1)%3] = Hamwind[1] * Hamwind[i6];
    for(i7=(i6+2);i7<2400;i7++)
        ac-inter[i6][i7%3] = ac-inter[i6][(i7-1)%3] +
            ac-inter[i6][(i7-2)%31 + (Hamwind[i7-i6] * Hamwind[i7]);
}

for(i8=0;i8<10;i8++) {
    v[0][i8] = ac-inter[i8+][2]; /* 2399 % 3 = 2 */
    u[0][i8] = ac-inter[i8][2];
}
```

The signal `AutoCorr[]` is a temporary signal. By reusing the memory space of signal `ac inter[]` for storing `AutoCorr[]` we can further reduce the total memory space required. This is achieved by inter-signal inplace mapping of array `AutoCorr[]` on `ac-inter[]`. Thus initially `ac-inter[]` could not have been accommodated in the on-chip local memory due to the large size of this signal but now we have removed this problem. This results both in reduced memory size and a reduction in the associated power consumption.

□

CAD techniques are needed to explore the many in-place mapping opportunities, the theory behind which is not presented in this survey. Effective techniques for the intra-signal mapping are described in [36, 84, 124], and for the inter-signal mapping in [37].

3.7 Data Cache-related Optimizations

The processor-cache interface is a familar architecture where system designers can benefit from decades of advances in compiler technology. Researchers in the recent past have addressed several problems related to the processor core-cache interface in embedded systems. However, the application specific design scenario presents opportunities for several new optimizations arising from three counts:

1. **Flexibility of the architecture:** many design parameters can be customized to fit the requirements of the application, e.g., cache size.
2. **Longer available compilation times:** many aggressive optimizations can be performed because more compilation times are now available.
3. **Full knowledge of the application:** The assumption that the compiler has access to the entire application at once allows us to perform many global optimizations that are skipped by traditional compilers, e.g., changing data layouts.

In this section, we survey new optimizations related to the memory hierarchy, viz., data caches that occur in the context of embedded systems.

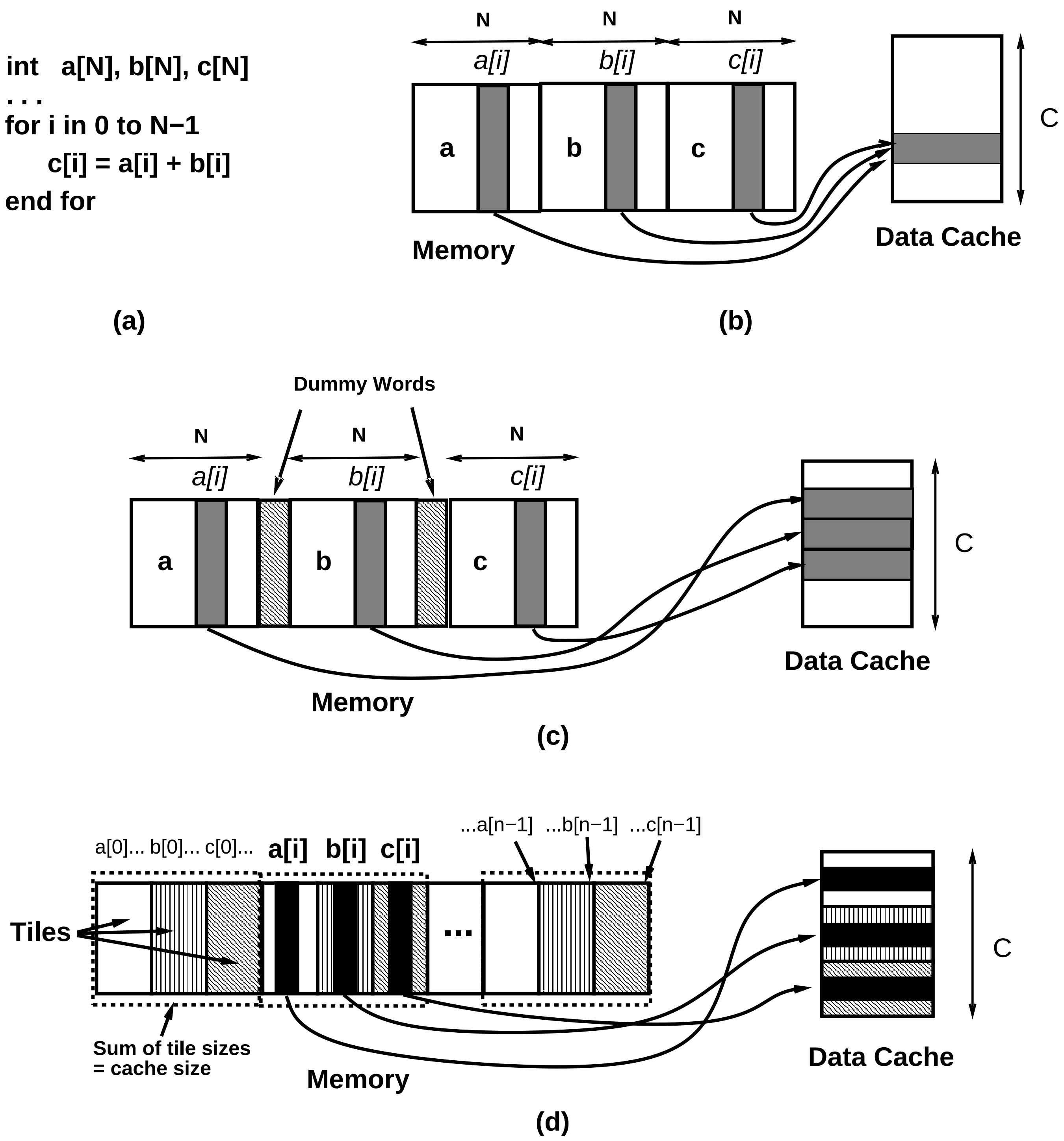
The data cache is an important architectural memory component that takes advantage of spatial and temporal locality by storing recently used memory data on-chip. Compiler optimizations to take advantage of data caches in the processor architecture have been widely studied [62]. We present here some data cache optimizations that have been proposed in order to take advantage of the flexibilities available with embedded systems. In the recent past, new architectural features such as block buffering and sub-banking [67, 140] have been proposed for low power cache design. In this survey, we do not cover these features, which have general applicability, in detail. Instead, we have covered only those arhitectural enhancements and associated compiler optimizations that are application-specific. Also, since the survey is about data-related optimization, we do not cover instruction caches.

3.7.1 Data Layout

The advance knowledge of the actual application to be executed on the system allows us to perform aggressive data layout optimizations [112, 81]. This refers to the observation that since the entire application is statically known, we can make a more intelligent placement of data structures in memory so as to improve the memory performance. Such optimizations are typically not performed by compilers because, for instance, they cannot assume that the translation unit under compilation represents the entire program – decisions on the best placement of data cannot be made because routines in a separate translation unit (a different source file that is not yet compiled) might access the same data in a completely different pattern, thereby invalidating all the previous analysis. However, if we assume that the entire application is available to us – not an unreasonable assumption in application specific design – then we can make intelligent decisions on data placement by analyzing the data access patterns.

Example:

Consider a direct-mapped cache of size C ($C = 2^m$) words, with a cache line size M words (i.e., M consecutive words are fetched from memory on a cache read miss), and a *write-through* cache with a *fetch-on-miss* policy [62].



```
int   a[N], b[N], c[N]
. . .
for i in 0 to N−1
     c[i] = a[i] + b[i]
end for
```

Figure 15. (a) sample code (b) cache conflicts: $a[i], b[i]$ and $c[i]$ map into the same cache line (c) data layout avoids cache conflicts – insertion of dummy words between arrays (d) alternate data layout – tiles interleaved in memory

Suppose the code fragment in Figure 15(a) is executed on a processor with the above cache configuration, where N is an exact power of 2, and $N \geq C$. Assuming that a single array element occupies one memory word, let array a begin at memory location 0, b at N, and c at $2N$. In a direct-mapped cache, the cache line that would contain a word located at memory address A, is given by: $(A \bmod C)/M$. In the above example, array element $a[i]$ would be located at memory address: i. Similarly, we have $b[i] : N + i$ and $c[i] : 2N + i$. Since N is a multiple of C, all of $a[i]$, $b[i]$, and $c[i]$ will map into the same cache line, as shown in Figure 15(b). Consequently, every data access results in a cache miss. Such memory access patterns are known to result in extremely inefficient cache utilization, especially because many applications deal with arrays whose dimensions are an exact power of two. The cache misses lead to inferior designs both in terms of performance and energy consumption. In such situations, increasing the cache size is not an efficient solution, because the cache misses are not caused due to lack of capacity. Note that there is only one active cache line during one loop iteration. The conflict-misses can be avoided if the cache size C is made greater than N, but there is an associated area and access time penalty incurred when cache size is increased. Reorganizing the data in memory results in a more elegant solution, while keeping the cache size relatively small.

[112] introduced a data layout technique where the thrashing caused by excessive cache conflicts is prevented by introducing M dummy memory words between two consecutive arrays that are accessed in an identical pattern in the loops. Array a begins at 0, array b begins at location: $N + M$ (instead of N) and array c begins at: $2N + 2M$ (instead of $2N$), as shown in Figure 15(c). This ensures that $a[i], b[i]$, and $c[i]$ are always mapped into different cache lines, and their accesses do not interfere with each other in the data cache. The cache size can still be small while maintaining efficient access.

A different data layout approach to avoiding the cache conflict problem is to split the initial arrays into sub-arrays (or *tiles*) of equal size (Figure 15(d)) [81]. The tile size is chosen such that the sum of the tile sizes of all arrays involved in a loop does not exceed the data cache size. Now, the tiles are merged in an interleaved fashion to form one large array. The fact that the tiles of different arrays accessed in the same loop iteration are adjacent in the data layout ensures that data in the tiles never conflict in the cache.

□

In a more complex application with several loop nests, both approaches outlined above would need to be generalized to handle different access patterns in different loops. The generalizations are discussed in [112, 81]. The optimal results are obtained by a combination of both approaches [81].

In addition to the relative placement and tiling of arrays, an additional degree of freedom is the row-major vs. column-major storage of multi-dimensional arrays discussed earlier in the context of reducing address bus transitions Section 3.5.2. Array access patterns in a loop can be used to infer the storage strategy that results in the best locality. However, different loop nests in an application may require conflicting storage strategies for the same array. [26] presents a general approach to solving this problem by combining both data and control transformation. Their algebraic framework evaluates the combined effects of storage strategy (row-major vs. column-major) and loop interchange transformations and selects the best candidate.

It should be noted that data layout for improving data cache performance has an analogue in instruction caches. Groups of instruction at different levels of granularity such as functions [97] and basic blocks [147, 146] can be laid out in memory so as to improve instruction cache performance.

3.7.2 Cache Access Scheduling

Traditionally, caches are managed by the hardware and thus cache accesses are transparent to schedulers, both in the compiler and in the custom hardware synthesis (HLS) domains. From the viewpoint of a traditional scheduler, all memory accesses are treated uniformly, assuming they take the same amount of time. For instance, cache accesses are scheduled optimistically, assuming they are cache hits; the scheduler relies on the memory controller to account for the longer delays of misses. However, the memory controller gets only a local view of the program, and is unable to perform the kinds of global optimizations afforded by a compiler. Recent approaches to cache access scheduling [54] have proposed a more accurate timing model for the memory accesses. By attaching accurate timing information to cache hits and misses, the compiler's scheduler is able to better hide the latency of the lengthy cache miss operations.

Prefetching was proposed as another solution to increase the hit ratio of caches, and has been studied extensively in the compiler and architecture communities. Hardware prefetching techniques [66] use structures such as stream buffers to recognize patterns in the stream of accesses in the hardware (through some recognition/prediction mechanism), and allocate streams to stream buffers, allowing prefetching of data from the main memory. Software prefetching [105] inserts prefetch instructions in the code, that bring data from the main memory into the cache well before it is needed in a computation.

3.7.3 Scratch Pad Memory

In the previous section, we have studied techniques for laying out data in memory for the familiar target architecture consisting of a processor core, a data cache, and external memory. However, an embedded system designer is not restricted to using only this memory architecture. Since the design needs to execute only a single application, we can use unconventional architectural variations that suit the specific application under consideration. One such design alternative is *Scratch Pad memory* [117, 115].

Scratch-Pad memory refers to data memory residing on-chip, that is mapped into an address space disjoint from the off-chip memory, but connected to the same address and data buses. Both the cache and Scratch-Pad memory (usually SRAM) allow fast access to their residing data, whereas an access to the off-chip memory (usually DRAM) requires relatively longer access times. The main difference between the Scratch-Pad SRAM and data cache is that the SRAM guarantees a single-cycle access time, whereas an access to the cache is subject to cache misses. The concept of Scratch Pad memory is an important architectural consideration in modern embedded systems, where advances in embedded DRAM technology have made it possible to combine DRAM and logic on the same chip. Since data stored in embedded DRAM can be accessed much faster and in a more power-efficient manner than that in off-chip DRAM, a related optimization problem that arises in this context is how to identify critical data in an application, for storage in on-chip memory.

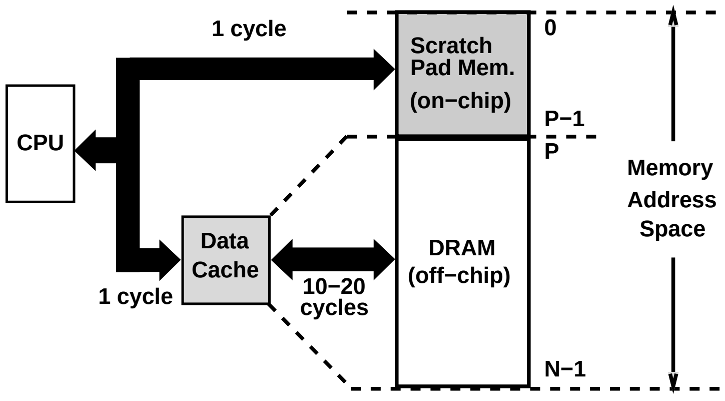


Figure 16. Division of Data Address Space between Scratch Pad memory and off-chip memory

The data address space mapping is shown in Figure 16, for a sample addressable memory of size N data words. Memory addresses $0 \ldots (P-1)$ map into the on-chip scratch pad memory, and have a single processor cycle access time. Memory addresses $P \ldots (N-1)$ map into the off-chip DRAM, and are accessed by the CPU through the data cache. A cache hit for an address in the range $P \ldots N-1$ results in a single-cycle delay, whereas a cache miss, which leads to a block transfer between off-chip and cache memory, may result in a delay of say 10-20 processor cycles.
Example:

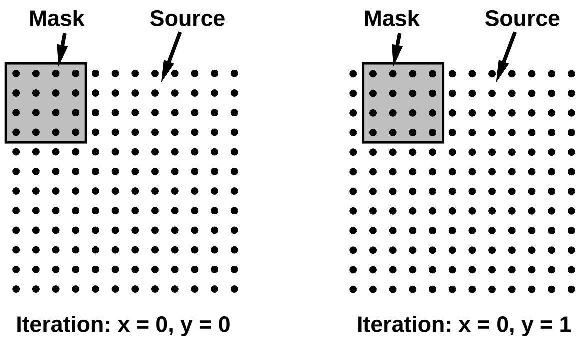
A small (4×4) matrix of coefficients, *mask*, slides over the input image, *source*, covering a different 4×4 region in each iteration of y, as shown in Figure 17. In each iteration, the coefficients of the mask are combined with the region of the image currently covered, to obtain a weighted average, and the result, *acc*, is assigned to the pixel of the output array, *dest*, in the center of the covered region. If the two arrays *source* and *mask* were to be accessed through the data cache, the performance would be affected by cache conflicts. This problem can be solved by storing the small *mask* array in the Scratch-pad memory. This assignment eliminates all conflicts in the data cache – the data cache is now used for memory accesses to *source*, which are very regular. Storing *mask* on-chip, ensures that frequently accessed data is never ejected off-chip, thereby significantly improving the memory performance and energy dissipation.
□

The memory assignment of [117] first determines a *Total Conflict Factor* (TCF) for each array based on the access frequency and possibility of conflict with other arrays and then considers the arrays for assignment to scratch pad memory in the order of TCF/(array size), giving priority to high-conflict/small-size arrays.

A scratch pad memory storing a small amount of frequently accessed data on-chip, has an equivalent in the instruction cache. The idea of using a small buffer to store blocks of frequently used instructions was first introduced

```
# define N 128
# define M 4
# define NORM 16
int source[N][N], dest [N][N];
int mask [M][M];
int acc, i, j, x, y;
⋮
for (x = 0; x < N − M; x++)
    for (y = 0; y < N − M; y++) {
        acc = 0;
        for (i = 0; i < M; i++)
            for (j = 0; j < M; j++)
                acc = acc + source[x+i][y+j] ∗ mask[i][j];
        dest[x+M/2][y+M/2] = acc/NORM;
    }
```

(a)



(b)

Figure 17. (a)Procedure CONV (b) Memory access pattern in CONV

in [66]. Recent extensions of this strategy are the Decoded Instruction Buffer [5] and the L-cache [13].

3.7.4 Cache Architecture Exploration

We now survey some recent research efforts that address the exploration space involving cache memories. A number of distinct memory architectures could be devised to exploit different application specific memory access patterns efficiently. Even if we restrict the scope of the architecture to those involving on-chip memory only, the exploration space of different possible configurations is too large, making it infeasible to exhaustively simulate the performance and energy characteristics of the application for each configuration. Thus, exploration tools are necessary for rapidly evaluating the impact of several candidate architectures. Such tools can be of great utility to a system designer by giving fast initial feedback on a wide range of memory architectures [115].

SRAM vs Data Cache

[114] presents MemExplore, an exploration framework for optimizing the on-chip data memory organization that addresses the following problem: given a certain amount of on-chip memory space, partition this into data cache and scratch pad memory so that the total access time and energy dissipation is minimized, i.e., the number of accesses to off-chip memory is minimized. In this formulation, an on-chip memory architecture is defined as a combination of the total size of on-chip memory used for data storage; the partitioning of this on-chip memory into: scratch memory, characterized by its size; and data cache, characterized by the cache size; and the cache line size. For each candidate on-chip memory size T, the technique considers different divisions of T into cache (size C) and scratch pad memory (size $S = T - C$), selecting only powers of 2 for C. The procedure described in Section 3.7.3 is used to identify the right data for storage in scratch pad memory. Among the data assigned to be stored in off-chip memory (and hence accessed through the cache), an estimation of the memory access performance is performed by combining and analysis of the array access patterns in the application and an approximate model of the cache behavior. The result of the estimation is the expected number of processor cycles required for all the memory accesses in the application. For each T, the (C, L) pair that is estimated to maximize performance is selected.

Example:

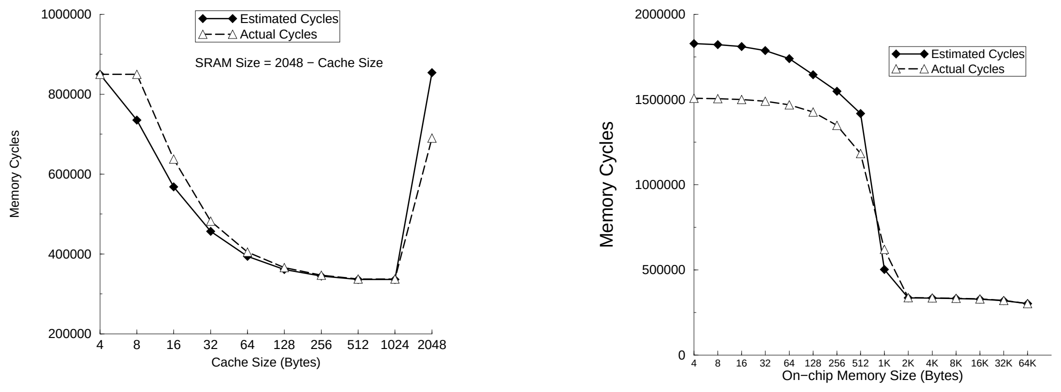


Figure 18. ***Histogram*** **Example (a) Variation of memory performance with different mixes of cache and Scratch-pad memory, for total on-chip memory of 2 KB (b) Variation of memory performance with total on-chip memory space**

Typical exploration curves of the MemExplore algorithm are shown in Figure 18. Figure 18(a) shows that the ideal division of a 2K on-chip space is 1K scratch pad memory and 1K data cache. Figure 18(b) shows that very little performance improvement is observed beyond a total on-chip memory size of 2KB.

□

The exploration curves of Figure 18 are generated from fast analytical estimates, which are three orders of magnitude faster than actual simulations, and are independent of data size. This estimation capability is important in the initial stages of system design, where the number of possible architectures is large, and a simulation of each architecture is prohibitively expensive.

Performance vs Power

Exploration of the effects of cache size on performance and cache power consumption was first studied in [80]. For a voice coder application, it was shown that the performance was maximal for cache sizes of 512 words or higher. However, the power of the memory related parts was minimal for 128 words. That minimum was also clearly influenced by other preceding code transformations such as the in-place mapping approach. Without the in-place mapping, the minimum moved to 256 words. [134] investigated this issue along other axes, as they also examine the impact of the data layout technique of Section 3.7.1, the tiling/blocking optimization [62], and set associativity on the data cache energy dissipation. In their study of the MPEG decoder algorithm, they reported that the minimum energy configuration of the implementation resulted from a cache size of 64 bytes, whereas the minimum delay configuration occurred at a size of 512 bytes. Both these studies provide important data for energy-delay trade-offs.

Datapath width and Memory Size

The CPU's bit-width is an additional parameter that can be tuned during architectural exploration of customizable processors. [131] studied the relationship between the width of the processor data path and the memory subsystem. This relationship is important when different data types with different sizes are used in the application. The key observation made is that as datapath width is decreased, the data memory size decreases because of less wasted space. For example, storing 3-bit data in a 4-bit word instead of 8-bit word), but the instruction memory might increase (e.g., storing 7-bit data in an 8-bit word requires only one instruction to access it, but requires two instructions if a 4-bit datapath is used. The authors use a RAM and ROM cost model to evaluate the cost of candidate bit-widths in a combined CPU-memory exploration.

In addition to the data cache research reviewed here, exploration studies involving the instruction cache have also been reported [73, 86, 88]

3.8 DRAM Optimizations

Applications involving large amounts of data typically need to store them in off-chip DRAMs when the on-chip area does not afford sufficient storage. The on-chip memory optimizations of the previous subsections are not adequate to handle the complex protocols associated with DRAM memory efficiently. New abstractions are needed to model the various available memory access modes in order to effectively integrate the DRAM into an automated synthesis system. This is especially important with the advent of embedded DRAM technology, where it is possible to integrate DRAM and logic into the same system on a chip [115].

3.8.1 DRAM Modeling for HLS and Optimization

The DRAM memory address is internally split into a *row address* consisting of the most significant bits and a *column address* consisting of the least significant bits. The row address selects a page from the core storage and the column address selects an offset within the page to arrive at the desired word. When an address is presented to the memory during a READ operation, the entire page addressed by the row address is read into the page buffer, in anticipation of spatial locality. If future accesses are to the same page, then there is no need to access the main storage area since it can just be read off the page buffer, which acts like a cache. Thus, subsequent accesses to the same page are very fast.

[113] describes a scheme for modeling the various memory access modes and uses them to perform useful optimizations in the context of a HLS environment.

Example:

Figure 19(a) shows a simplified timing diagram of the *read cycle* of a typical DRAM. The Memory Read cycle is initiated by the falling edge of the RAS (Row Address Strobe) signal, at which time the row address is latched from the address bus. The column address is latched at the falling edge of CAS (Column Address Strobe) signal, which should occur at least $T_{ras} = 45$ ns later. Following this, the data is available on the data bus after $T_{cas} = 15$ ns. Finally, the RAS signal is held high for at least $T_p = 45$ ns to allow for *bit-line precharge*, which is necessary before the next memory cycle can be initiated.

In order to use the above information in an automated scheduling tool, we need to abstract out a set of control data flow graph (CDFG) nodes from the timing diagram [113]. The CDFG node cluster for the memory read operation consists of 3 stages (Figure 19(b)): (1) row decode; (2) column decode; and (3) precharge. The row and column addresses are available at the first and second stages respectively, and the output data is available at the beginning of the third stage. Assuming a clock cycle of 15 ns, and a 1-cycle delay for the addition and shift operations, we can derive the schedule shown in Figure 19(d) for the code in Figure 19(c), using the memory read model in Figure 19(b).

Since the four accesses to array b are treated as four independent memory reads, each of these incurs the entire read cycle delay of $T_{rc} = 105$ ns, i.e., 7 cycles, requiring a total of $7 \times 4 = 28$ cycles.

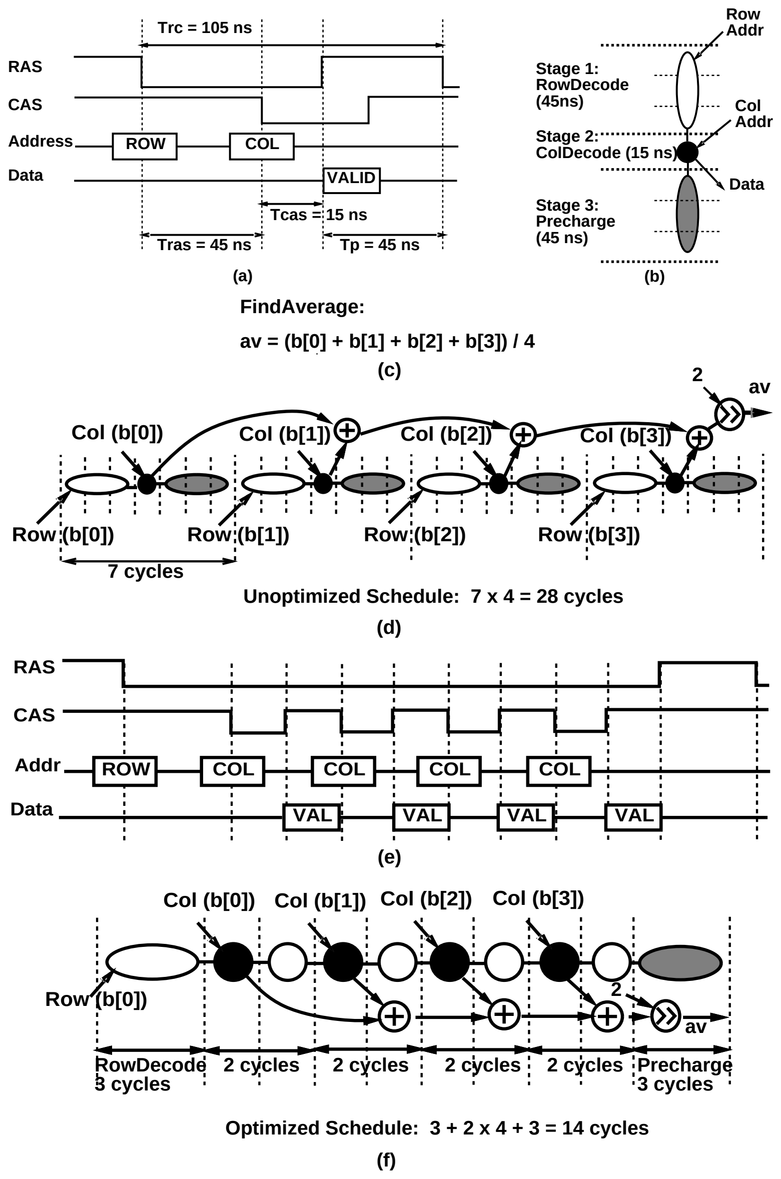


Figure 19. (a) Timing diagram for Memory Read cycle (b) Model for Memory Read operation (c) Code for *FindAverage* (d) Treating the memory accesses as independent Reads (e) Timing diagram of *page mode read* cycle (f) Treating the memory accesses as one page mode read cycle

However, DRAM features such as page mode read can be efficiently exploited to generate a much tighter schedule for behaviors such as the *FindAverage* example, which access data in the same page, in succession. Figure 19(e) shows the timing diagram for the *page mode read* cycle, and Figure 19(f) shows the schedule for the *FindAverage* routine using the page mode read feature. Note that the page mode does not incur the long row decode and precharge times between successive accesses, thereby eliminating a significant amount of delay from the schedule. In this case, the column decode time is followed by a *minimum pulse width* duration for the CAS signal, which is also 15 ns in our example. Thus, the effective cycle time between successive memory accesses has been greatly reduced, resulting in an overall reduction of 50% in the total schedule length. □

The key feature in the reduction of the schedule length in the example above is the recognition that the input behavior is characterized by memory access patterns that are amenable to the page mode feature, and the incorporation of this observation in the scheduling phase. Some additional DRAM-specific optimizations discussed in [113] are:

Read-Modify-Write (R-M-W) optimization that takes advantage of the R-M-W mode in modern DRAMs which provides support for a more efficient realization of the common case where a specific address is read, the data is involved in some computation, and then the output is written back to the same location.

Hoisting where the row-decode node is scheduled ahead of a conditional node if the first memory access in both branches are on the same page.

Unrolling optimization in the context of supporting the page mode accesses indicated in Figure 19(f).

The models that are described here can be introduced as I/O profiles in the memory access ordering (SCBD) approach of section 3.2. A good overview of the performance implications of the architectural features of modern DRAMs is found in [28].

3.8.2 Synchronous DRAM/Banking Optimization

As DRAM architectures evolve, new challenges are presented to the automatic synthesis of embedded systems based on these memories. *Synchronous DRAM* represents an architectural advance that presents another optimization opportunity: multiple memory banks. The core memory storage is divided into multiple banks, each with its own independent page buffer, so that two separate memory pages can be simultaneously active in the multiple page buffers.

[71] has addressed the problem of modeling the access modes of synchronous DRAMs such as:

Burst mode read/write – fast successive accesses to data in the same page.

Interleaved row read/write modes – alternating burst accesses between banks.

Interleaved Column Access – alternating burst accesses between two chosen rows in different banks.

Memory bank assignment is performed by creating an interference graph between arrays and partitioning it into subgraphs so that data in each part is assigned to a different memory bank. The bank assignment algorithm is related to techniques such as [141] that address memory assignment for DSP processors such as the Motorola 56000 which has a dual-bank internal memory/register file [127, 27]. The bank assignment problem in [141] is targeted at scalar variables, and is solved in conjunction with register allocation by building a constraint graph that models the data transfer possibilities between registers and memories followed by a simulated annealing step.

[24] approached the SDRAM bank assignment problem by first constructing an *array distance table*. This table stores the *distance* in the DFG between each pair of arrays in the specification. A short distance indicates a strong correlation, possibly indicating that they might be, for instance, two inputs of the same operation, and hence, would benefit from being assigned to separate banks. The bank assignment is finally performed by considering array pairs in increasing order of their array distance information.

The presence of embedded DRAMs adds several new dimensions to traditional architecture exploration. One interesting aspect of DRAM architecture that can be customized for an application is the banking structure. Figure 20(a) illustrates a common problem with the single-bank DRAM architecture. If we have a loop that accesses in succession data from three large arrays A, B, and C, each of which is much larger than a page, then each memory access leads to a fresh page being read from the storage, effectively cancelling the benefits of the page buffer. This page buffer interference problem cannot be avoided if a fixed architecture DRAM is used. However, an elegant solution to the problem is available if the banking configuration of the DRAM can be customized for the application [116]. Thus, in the example of Figure 20, the arrays can be assigned to separate banks as shown in Figure 20(b). Since each bank has its own private page buffer, there is no interference between the arrays, and the memory accesses do not represent a bottleneck.

In order to customize the banking structure for an application, we need to solve the memory bank assignment problem – determine an optimal banking structure (number of banks) and determine the assignment of each array variable into the banks such that the number of page misses is minimized. This objective optimizes both the performance as well as the energy dissipation of the memory subsystem. The memory bank customization problem

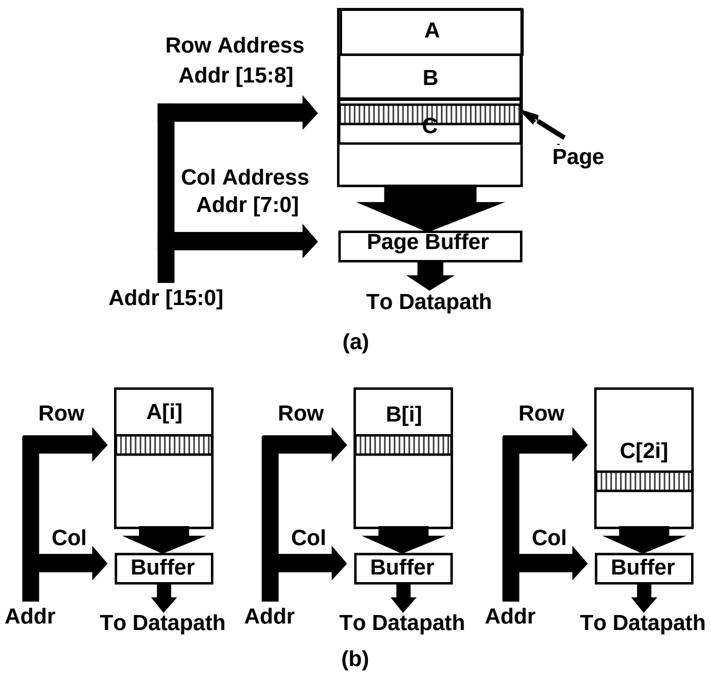


Figure 20. (a) Arrays mapped to a single-bank memory (b) 3-bank memory architecture

is solved in [116] by modeling the assignment as a partitioning problem – partition a given set of nodes into a given number of groups such that a given criterion (bank misses in this case) is optimized. The partitioning proceeds by associating a cost of assigning two arrays into the same bank, determined by the number of accesses to the arrays and the loop count. If the arrays are accessed in the same loop, then the cost is high, thereby discouraging the partitioning algorithm from assigning them to the same bank. On the other hand, if two arrays are never accessed in the same loop, then they are candidates for assignment into the same bank. This pairing is associated with a low cost, guiding the partitioner to assign the arrays together.

Bank assignment can also be seen as array-to-memory assignment of subsection 3.3 when the appropriate cost function and I/O profile constraints are introduced (see [18]).

3.8.3 Memory-aware compilation

Traditionally, the compiler is shielded from the detailed organization and timing of the memory subsystem; interactions with the memory subsystem are typically through read and write operations with the timing granularity at the level cache hit and miss delays. However, a memory-aware compiler approach [53] can aggressively exploit the detailed timing information of the memory subsystem to obtain improved scheduling results. In [53], the authors present an algorithm TIMGEN for including DRAM characteristics in a compiler framework. Detailed timing information of DRAM nodes is made available to the compiler, which can then make intelligent scheduling decisions based on the timing knowledge. For each instruction, TIMGEN traces a detailed timing path through the processor pipeline including different possible memory access modes. This information is then used during scheduling to generate aggressive schedules that are on the average 24% smaller than one which assumes no knowledge of memory timing.

3.8.4 Architectural Description Language (ADL) driven Processor-Memory Coexploration

Processor Architecture Description Languages (ADLs) have been developed to allow for a language-driven exploration and software toolkit generation approach [144, 60]. Currently most ADLs assume an implicit/default memory organization, or are limited to specifying the characteristics of a traditional memory hierarchy. Since embedded systems may contain non-traditional memory organizations, there is a great need to model explicitly the memory subsystem for an ADL-driven exploration approach. A recent approach [104] describes the use of the EXPRESSION ADL [59] to drive Memory Architecture Exploration. The EXPRESSION ADL description of the processor-memory architecture is used to explicitly capture the memory architecture, including the characteristics of the memory modules (such as caches, DRAMs, SRAMs DMAs), the parallelism and pipelining present in the memory architecture (e.g., resources used, timings, access modes). Each such explicit memory architecture description is then used to automatically generate the information needed by the compiler [53, 54] to efficiently utilize the features in the memory architecture, and to generate a memory simulator, allowing feedback to the designer on the match between the application, the compiler and the memory architecture.

4 Address Generation

One important consequence of all the above memory organisation related steps, is that the address sequences typically become much more complicated than in the original non-optimized application code. This is first of all due to the source code transformations like in-place mapping that introduce modulo arithmetic and loop transformations, which in turn generate more index arithmetic and manifest local conditions. Additional complexity is added due to the very distributed memory organisation used in embedded processors, both custom and programmable. As a result, *address generation*, which involves generating efficient assembly code or hardware to implement the translation of array references to actual memory addresses, is a critical stage in the entire data management flow.

Initial work on address generation was focussed only on regular DSP applications mapped on hardware. Central to this early research is the observation that if the generated addresses are known to be periodic (true for many DSP applications accessing large data arrays), then there is no need to use a full-blown arithmetic circuit to generate the sequence; a simple counter-based circuit can achieve the same effect. The first research on synthesizing hardware address generation focussed on generating efficient designs from a specified trace of memory address [50, 49] by recognizing the periodicity of the patterns and automatically building a counter-based circuit for generating the sequence of addresses. Since the problem of extracting the periodic behavior from an arbitrary sequence of addresses can be extremely difficult, works such as [49] rely on designer hints such as the number of memory accesses in the basic repeating pattern. The ZIPPO system [51] solves a generalization of the above problem by considering several address streams that are incident on different memory modules on-chip, and synthesizing an address generator that is optimized by sharing of hardware. Multiplexing of such sequences allowed even more exploration freedom and even better results [102].

Another simplification of address generation hardware can be achieved by employing certain interesting properties of the exclusive OR (XOR) function. [129] presents an *Address Bit Inversion* technique for generating simplified address generator at the expense of a small area overhead. The authors point out that if two arrays a and b have sizes $0...A-1$ and $0...B-1$ respectively, such that $A\&B=0$, i.e., the bit-wise AND of the arrays sizes is zero, then two disjoint address spaces are created by performing a bit-wise XOR on the index of one array with the size of the other.
Example:

Suppose we have to store two arrays a and b with sizes 3 and 4 words respectively in the same memory module. In order to access random elements $a[i]$ and $b[j]$ from memory, the arrays would normally be located contiguously in memory, and the addressing circuit would be implemented as follows:

Address for a[i]:	i
Address for b[j]:	3 + j

However, since the bitwise AND of 3 and 4 is zero, we can use the following implementation to generate distinct memory address spaces for a and b.

Address for a[i]:	i XOR 4
Address for b[j]:	j XOR 3

The latter addressing scheme is asymptotically faster because it incurs a maximum of one inverter delay whereas the former incurs the delay due to an adder circuit. The disadvantage of the Address Bit Inversion technique is that the array sizes have to be increased until the condition $A\&B=0$ is satisfied. This involves wastage of memory space, which is a maximum of 17.4% in the author's experiments. □

An alternative approach to the above work was proposed by the authors of [103] where custom architectures based on arithmetic building blocks (application specific units – ASUs) were explored. In this context, time multiplexing and merging of index expressions were crucial in obtaining a cost-efficient result. Easily computable measures could be estimated from the application code in deciding whether the counter-based or ASU-based approach worked best, on the basis of individual address expressions.

The combined Adopt methodology for address generation described in [103] is a general framework for optimizing address generators. First, an address expression extraction step obtains the address expressions from the internal Control/Data Flow Graph representation. Next, address expression splitting and clustering leads to several optimization opportunities while selecting the target architecture, which can be one of two types: *incremental address generation unit* and *custom address calculation unit*. Algebraic transformations are finally applied to globally optimize the generated addressing logic.

In addition to these custom address generation approaches, much effort has also been spent on the mapping of application code on programmable address generators. See [85, 89] for early work on exploiting auto-increment modes and to deal with limitations on (index) register storage. Additional optimisation steps have been introduced later to support algebraic factoring and polynomial induction variable analysis [58] and modulo arithmetic [47].

5 Conclusions

In this survey we presented a survey of contemporary and emerging data and memory optimization techniques for embedded systems.

First we discussed platform-independent memory optimizations that operate on a source-to-source level, and which typically guarantee improved performance, power and cost metrics, irrespective of the implementation's target architecture. Next we surveyed a number of data and memory optimization techniques applicable to memory structures at different levels of architectural granularity: from registers and register files, all the way up to off-chip memory structures. Finally, we discussed the attendant address generation optimizations that remove the address and local control overhead which appears as a byproduct of both platform-independent, as well as platform-dependent data and memory optimizations.

Given the constraints on the length of this manuscript, we have attempted to survey a wide range of both traditional approaches, as well as emerging techniques designed to handle memory issues in embedded systems, from the viewpoint of performance, power and area (cost). We have not addressed the context of much more parallel platforms including more data- and (dynamic) task-level parallelism. Many open issues remain in the context of memory-intensive embedded systems, and these have also not been addressed in this survey, including the issues of testing, validation and (formal) verification, embedded system reliability, and optimization opportunities in the context of networked embedded systems.

As complex embedded Systems-on-a-Chip (SOC) begin to proliferate, and as the software content of these embedded SOCs dominate the design process, memory issues will continue to be a critical optimization dimension in the design and development of future embedded systems.

Acknowledgements: We gratefully acknowledge the input from our colleagues at IMEC and the ACES laboratory at U.C.Irvine and their many research contributions that are partly summarized in this survey.

References

[1] A.Agarwal, D.Krantz, V.Nataranjan, "Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors", *IEEE Trans. on Parallel and Distributed Systems*, Vol.6, No.9, pp.943-962, Sep. 1995.

[2] I. Ahmad and C. Y. R. Chen. Post-processor for data path synthesis using multiport memories. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 276–279, Santa Clara, CA, November 1991.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1993.

[4] S.Amarasinghe, J.Anderson, M.Lam, and C.Tseng, "The SUIF compiler for scalable parallel machines", in *Proc. of the 7th SIAM Conf. on Parallel Proc. for Scientific Computing*, 1995.

[5] R. S. Bajwa, M. Hiraki, H. Kojima, D. J.. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on VLSI Systems*, 5(4):417–424, December 1997.

[6] S. Bakshi and D. Gajski. A memory selection algorithm for high-performance pipelines. In *Proceedings of the European Design Automation Conference*, pages 124–129, Brighton, U.K., 1995.

[7] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, J. G. Linders, and J. C. Majithia. Allocation of multiport memories in data path synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(4):536–540, April 1988.

[8] F.Balasa, F.Catthoor, H.De Man, "Dataflow-driven memory allocation for multi-dimensional processing systems", *Proc. IEEE Intnl. Conf. on Computer Aided Design*, San Jose CA, Nov. 1994.

[9] F.Balasa, F.Catthoor, H.De Man, "Background Memory Area Estimation for Multi-dimensional Signal Processing Systems", *IEEE Trans. on VLSI Systems*, Vol.3, No.2, pp.157-172, June 1995.

[10] U.Banerjee, "Dependence Analysis for Supercomputing", Kluwer Acad. Publ., Boston, 1988.

[11] U.Banerjee, R.Eigenmann, A.Nicolau, D.Padua, "Automatic program parallelisation", *Proc. of the IEEE*, invited paper, Vol.81, No.2, Feb. 1993.

[12] P.Banerjee, J.Chandy, M.Gupta, E.Hodges, J.Holm, A.Lain, D.Palermo, S.Ramaswamy, E.Su, "The Paradigm compiler for distributed-memory multicomputers", *IEEE Computer Magazine*, Vol.28, No.10, pp.37-47, Oct. 1995.

[13] N. Bellas, I. N. Hajj, C. D. Polychronopoulos, and G. Stamoulis. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Transactions on VLSI Systems*, 8(3):317–326, June 2000.

[14] L. Benini, G. de Micheli, E. Macii, M. Poncino, and S. Quer. Power optimization of core-based systems by address bus encoding. *IEEE Transactions on VLSI Systems*, 6(4):554–562, December 1998.

[15] L. Benini, G. de Micheli, E. Macii, D. Sciuto, and C. Silvano. Address bus encoding techniques for system-level power optimization. In *Design, Automation and Test in Europe*, Paris, France, February 1998.

[16] L.Benini, A.Macii, M.Poncino, "A recursive algorithm for low-power memory partitioning", *Proc. IEEE Intnl. Symp. on Low Power Design*, Rapallo, Italy, pp.78-83, Aug. 2000.

[17] L.Benini, G.De Micheli, "System-level power optimization techniques and tools", *ACM Trans. on Design Automation for Embedded Systems (TODAES)*, Vol.5, No.2, April 2000.

[18] E.Brockmeyer, A.Vandecappelle, F.Catthoor, "Systematic Cycle budget versus System Power Trade-off: a New Perspective on System Exploration of Real-time Data-dominated Applications", *Proc. IEEE Intnl. Symp. on Low Power Design*, Rapallo, Italy, pp.137-142, Aug. 2000.

[19] E.Brockmeyer, S.Wuytack, A.Vandecappelle, F.Catthoor, "Low power storage cycle budget tool support for hierarchical graphs", *Proc. 13th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, Madrid, Spain, pp.20-22, Sep. 2000.

[20] F.Catthoor, M.Janssen, L.Nachtergaele, H.De Man, "System-level data-flow transformations for power reduction in image and video processing", *Proc. Intnl. Conf. on Electronic Circuits and Systems*, Greece, pp.1025-1028, Oct. 1996.

[21] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, A.Vandecappelle, "Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design", ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.

[22] F.Catthoor, K.Danckaert, C.Kulkarni, T.Omnes, "Data transfer and storage architecture issues and exploration in multimedia processors", book chapter in "Programmable Digital Signal Processors: Architecture, Programming, and Applications" (ed. Y.H.Yu), Marcel Dekker, Inc., New York, 2000.

[23] G. Chaitin, M. Auslander, A. Chandra, J. Coocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6, January 1981.

[24] H.-K. Chang and Y.-L. Lin. Array allocation taking into account sdram characteristics. In *Asia and South Pacific Design Automation Conference*, pages 497–502, Yokohama, January 2000.

[25] T-S.Chen, J-P.Sheu, "Communication-free data allocation techniques for parallelizing compilers on multicomputers", *IEEE Trans. on Parallel and Distributed Systems*, Vol.5, No.9, pp.924-938, Sep. 1994.

[26] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, CA, June 1995.

[27] J.-L. Cruz, A. Gonzalez, M. Valero, and N. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture*, pages 315–325, Vancouver, Canada, June 2000.

[28] Vinodh Cuppu, Bruce L. Jacob, Brian Davis, and Trevor N. Mudge. A performance comparison of contemporary dram architectures. In *International Symposium on Computer Architecture*, pages 222–233, Atlanta, GA, May 1999.

[29] K.Danckaert, F.Catthoor, H.De Man, "System-level memory management for weakly parallel image processing", *Proc. EuroPar Conference*, Lyon, France, August 1996. "Lecture notes in computer science" series, Vol.1124, Springer Verlag, pp.217-225, 1996.

[30] K.Danckaert, F.Catthoor, H.De Man, "Platform independent data transfer and storage exploration illustrated on a parallel cavity detection algorithm", *Proc. ACM Conf. on Par. and Dist. Proc. Techniques and Applications*, PDPTA'99, Vol.III, pp.1669-1675, Las Vegas NV, June 1999.

[31] K.Danckaert, F.Catthoor, H.De Man, "A preprocessing step for global loop transformations for data transfer and storage optimization", *Proc. Intnl. Conf. on Compilers, Arch. and Synth. for Emb. Sys.*, San Jose CA, Nov. 2000.

[32] A.Darte, Y.Robert, "Affine-by-statement scheduling of uniform loop nests over parametric domains", Technical report 92-16, LIP, Ecole Normale Sup. de Lyon, April 1992.

[33] A.Darte, T.Risset, Y.Robert, "Loop nest scheduling and transformations", in *Environments and Tools for Parallel Scientific Computing*, J.J.Dongarra et al. (eds.), Advances in Parallel Computing 6, North Holland, Amsterdam, pp.309-332, 1993.

[34] J.L.da Silva Jr, F.Catthoor, D.Verkest, H.De Man, "Power Exploration for Dynamic Data Types through Virtual Memory Management Refinement", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey CA, pp.311-316, Aug. 1998.

[35] E.De Greef, F.Catthoor, H.De Man, "Memory organization for video algorithms on programmable signal processors", *Proc. IEEE Int. Conf. on Computer Design*, Austin TX, pp.552-557, Oct. 1995.

[36] E.De Greef, F.Catthoor, H.De Man, "Reducing storage size for static control programs mapped onto parallel architectures", *Dagstuhl Seminar on Loop Parallelisation*, Schloss Dagstuhl, Germany, April 1996.

[37] E.De Greef, F.Catthoor, H.De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems", *Proc. Intnl. Conf. on Applic.-Spec. Array Processors*, Zurich, Switzerland, pp.66-75, July 1997.

[38] J.P.Diguet, S.Wuytack, F.Catthoor, H.De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey, pp.30-35, Aug. 1997.

[39] C.Ding, K.Kennedy, "The memory bandwidth bottleneck and its amelioration by a compiler", *Proc. Intnl. Parallel and Distr. Proc. Symp.(IPDPS)* in Cancun, Mexico, pp.181-189, May 2000.

[40] P.Feautrier, "Dataflow analysis of array and scalar references", *Int. J. of Parallel Programming*, Vol.20, No.1, pp.23-53, 1991.

[41] P.Feautrier, "Compiling for massively parallel architectures: a perspective", *Intnl. Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994. Also in "Algorithms and Parallel VLSI Architectures III" (eds. M.Moonen, F.Catthoor), Elsevier, pp.259-270, 1995.

[42] A.Fraboulet, G.Huard, A.Mignotte, "Loop alignment for memory access optimisation", *Proc. 12th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, San Jose CA, pp.71-70, Dec. 1999.

[43] F.Franssen, F.Balasa, M.van Swaaij, F.Catthoor, H.De Man, "Modeling Multi-Dimensional Data and Control flow", *IEEE Trans. on VLSI systems*, Vol.1, No.3, pp.319-327, Sep. 1993.

[44] F.Franssen, L.Nachtergaele, H.Samsom, F.Catthoor, H.De Man, "Control flow optimization for fast system simulation and storage minimization", *Proc. 5th ACM/IEEE Europ. Design and Test Conf.*, Paris, France, pp.20-24, Feb. 1994.

[45] D. Gajski, N. Dutt, S. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[46] M. R. Garey and D. S. Johnson. *Computers and Intractibility – A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[47] C.Ghez, M.Miranda, A.Vandecappelle, F.Catthoor, D.Verkest, "Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm", *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Lafayette LA, IEEE Press, pp.623-632, Oct. 2000.

[48] G.Goossens, J.Vandewalle, H.De Man, "Loop optimisation in register-transfer scheduling for DSP systems", *Proc. 26th ACM/IEEE Design Automation Conf.*, Las Vegas NV, pp.826-831, June 1989.

[49] D. Grant and P. B. Denyer. Address generation for array access based on modulus m counters. In *Proceedings of the European Conference on Design Automation*, pages 118–123, Amsterdam, February 1991.

[50] D. Grant, P. B. Denyer, and I. Finlay. Synthesis of address generators. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 116–119, Santa Clara, CA, November 1989.

[51] D. M. Grant, J. Van Meerbergen, and P. E. R. Lippens. Optimization of address generator hardware. In *European Design and Test Conference*, pages 325–329, Paris, March 1994.

[52] P.Grun, F.Balasa, and N.Dutt, "Memory Size Estimation for Multimedia Applications", *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes)*, Seattle WA, pp.145-149, March 1998.

[53] P. Grun, N. Dutt, and A. Nicolau. Memory aware compilation through accurate timing extraction. In *Design Automation Conference*, pages 316–321, Los Angeles, CA, June 2000.

[54] P. Grun, N. Dutt, and A. Nicolau. Mist: An algorithm for memory miss traffic management. In *ICCAD*, San Jose, 2000.

[55] P. Grun, N. Dutt, and A. Nicolau. Access pattern based local memory customization for low power embedded systems. In *DATE*, Munich, 2001.

[56] A. Gonzales, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *International Conference on Supercomputing (ICS)*, 1995.

[57] M.Gupta, E.Schonberg, H.Srinivasan, "A Unified Framework for Optimizing Communication in Data-Parallel Programs", *IEEE Trans. on Parallel and Distributed Systems*, Vol.7, No.7, pp.689-704, July 1996.

[58] S.Gupta, M.Miranda, F.Catthoor, R.Gupta, "Analysis of high-level address code transformations for programmable processors", *Proc. 3rd ACM/IEEE Design and Test in Europe Conf.*, Paris, France, pp.9-13, April 2000.

[59] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings DATE'99*, Munich, Germany, March 1999.

[60] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic software toolkit generation for embedded systems-on-chip. In *Proceedings ICVC'99*, Korea, 1999.

[61] M.Hall, J.Anderson, S.Amarasinghe, B.Murphy, S.Liao, E.Bugnion, M.Lam, "Maximizing multiprocessor performance with the SUIF compiler", *IEEE Computer Magazine*, Vol.30, No.12, pp.84-89, Dec. 1996.

[62] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach.* Morgan Kaufman, San Francisco, CA, 1994.

[63] C.Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu. Data path allocation based on bipartite weighted matching. In *Design Automation Conference*, pages 499 – 504, Orlando, June 1990.

[64] K.Itoh, K.Sasaki, Y.Nakagome, "Trends in low-power RAM circuit technologies", special issue on "Low power electronics" of the *Proceedings of the IEEE*, Vol.83, No.4, pp.524-543, April 1995.

[65] P. K. Jha and N. Dutt. Library mapping for memories. In *European Design and Test Conference*, pages 288–292, Paris, France, March 1997.

[66] N.Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Proc. ACM Int. Symp. on Computer Arch.*, pp.364-373, May 1990.

[67] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *International Symposium on Low Power Electronics and Design*, pages 143–148, Monterey, CA, August 1997.

[68] M.Kandemir, N.Vijaykrishnan, M.J.Irwin, W.Ye, "Influence of compiler optimisations on system power", *Proc. 37th ACM/IEEE Design Automation Conf.*, Los Angeles CA, pp.304-307, June 2000.

[69] D. Karchmer and J. Rose. Definition and solution of the memory packing problem for field-programmable systems. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 20–26, San Jose, CA, November 1994.

[70] W.Kelly, W.Pugh, "Generating schedules and code within a unified reordering transformation framework", Technical Report UMIACS-TR-92-126, CS-TR-2995, Institute for Advanced Computer Studies Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742, 1992.

[71] A. Khare, P. R. Panda, N. D. Dutt, and A. Nicolau. High-level synthesis with sdrams and rambus drams. *IEICE Transactions*, VLSI99(17), 1999.

[72] T. Kim and C. L. Liu. Utilization of multiport memories in data path synthesis. In *Design Automation Conference*, pages 298–302, Dallas, TX, June 1993.

[73] D. Kirovski, C. Lee, M. Potkonjak, and W. Mangione-Smith. Application-driven synthesis of memory-intensive systems-on-chip. *IEEE Transactions on Computer Aided Design*, 18(9):1316–1326, September 1999.

[74] P-G.Kjeldsberg, F.Catthoor, E.J.Aas, "Storage requirement estimation for data-intensive applications with partially fixed execution ordering", *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes)*, San Diego CA, pp.56-60, May 2000.

[75] P-G.Kjeldsberg, F.Catthoor, E.J.Aas, "Automated data dependency size estimation with a partially fixed execution ordering", *Proc. IEEE Int. Conf. on Comp. Aided Design*, Santa Clara CA, pp.44-50, Nov. 2000.

[76] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[77] D.Kolson, A.Nicolau, N.Dutt, "Minimization of memory traffic in high-level synthesis", *Proc. 31st ACM/IEEE Design Automation Conf.*, San Diego, CA, pp.149-154, June 1994.

[78] H. Kramer and J. Muller. Assignment of global memory elements for multi-process vhdl specifications. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 496–501, Santa Clara, CA, November 1992.

[79] D.Kulkarni, M.Stumm, "Linear loop transformations in optimizing compilers for parallel machines", *The Australian Computer Journal*, pp.41-50, May 1995.

[80] C.Kulkarni, F.Catthoor, H.De Man, "Cache transformations for low power caching in embedded multimedia processors", *Proc. Intnl. Parallel Proc. Symp.(IPPS)*, Orlando FL, pp.292-297, April 1998.

[81] C. Kulkarni, F. Catthoor, and H. De Man. Advanced data layout organization for multi-media applications. In *Proceedings Workshop on Parallel and Distributed Computingin Image Processing, Video Processing, and Multimedia (PDIVM'2000)*, Cancun, Mexico, May 2000.

[82] F.J.Kurdahi, A.C.Parker, "REAL: a program for register allocation", *Proc. 24th ACM/IEEE Design Automation Conf.*, Miami FL, pp.210-215, June 1987.

[83] H.-D. Lee and S.-Y. Hwang. A scheduling algorithm for multiport memory minimization in datapath synthesis. In *Proceedings of the Conference on Asia Pacific Design Automation Conference*, pages 93–100, Makuhari, Japan, August 1995.

[84] V.Lefebvre, P.Feautrier, "Optimizing storage size for static control programs in automatic parallelizers", *Proc. EuroPar Conference*, Passau, Germany, August 1997. "Lecture notes in computer science" series, Springer Verlag, Vol.1300, 1997.

[85] R.Leupers, P.Marwedel, "Algorithms for address assignment in DSP code generation", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose CA, pp.109-112, Nov. 1996.

[86] Y. Li and J. Henkel. A framework for estimating and minimizing energy dissipation of embedded hw/sw systems. In *Design Automation Conference*, pages 188–193, San Francisco, CA, June 1998.

[87] W.Li, K.Pingali. "A singular loop transformation framework based on non-singular matrices", *Proc. 5th Annual Workshop on Languages and Compilers for Parallelism*, New Haven CN, August 1992.

[88] Y. Li and W. H. Wolf. Hardware/software co-synthesis with memory hierarchies. *IEEE Transactions on Computer Aided Design*, 18(10):1405–1417, October 1999.

[89] C.Liem, P.Paulin, A.Jerraya, "Address calculation for retargetable code generation and exploration of instruction-set architectures", *Proc. 33rd ACM/IEEE Design Automation Conf.*, Las Vegas NV, pp.597-600, June 1996.

[90] P.Lippens, J.van Meerbergen, W.Verhaegh, A.van der Werf, "Allocation of multiport memories for hierarchical data streams", *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, Nov. 1993.

[91] D.B. Loveman, "Program improvement by source-to-source transformation", *Journal of the ACM*, Vol.24, No.1, pp.121-145, 1977.

[92] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using behavioral templates. In *ACM/IEEE Design Automation Conference*, pages 599–604, June 1995.

[93] A.Malik, B.Moyer, D.Cermak, "A low power unified cache architecture providing power and performance flexibility", *Proc. IEEE Intnl. Symp. on Low Power Design*, Rapallo, Italy, pp.241-243, Aug. 2000.

[94] N.Manjiakian, T.Abdelrahman, "Fusion of loops for parallelism and locality", Technical report CSRI-315, Comp. Systems Res. Inst. Univ. of Toronto, Canada, Feb. 1995.

[95] K.Masselos, K.Danckaert, F.Catthoor, C.E.Goutis, H.DeMan, "A methodology for power efficient partitioning of data-dominated algorithm specifications within performance constraints", *Proc. IEEE Intnl. Symp. on Low Power Design*, San Diego CA, pp.270-272, Aug. 1999.

[96] K.Masselos, F.Catthoor, C.E.Goutis, H.De Man, "A performance oriented use methodology of power optimizing code transformations for multimedia applications realized on programmable multimedia processors", *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Taipeh, Taiwan, IEEE Press, pp.261-270, Oct. 1999.

[97] S. McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Boston, MA, April 1989.

[98] K.McKinley, M.Hall, T.Harvey, K.Kennedy, N.McIntosh, J.Oldham, M.Paleczny, and G.Roth, "Experiences using the ParaScope editor: an interactive parallel programming tool", in *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, USA, May 1993.

[99] K.McKinley, S.Carr, C-W.Tseng, "Improving data locality with loop transformations", *ACM Trans. on Programming Languages and Systems*, Vol.18, No.4, pp.424-453, July 1996.

[100] K. McKinley. A compiler optimization algorithm for shared-memory multi-processors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, August 1998.

[101] T.H.Meng, B.Gordon, E.Tsern, A.Hung, "Portable video-on-demand in wireless communication", special issue on "Low power electronics" of the *Proceedings of the IEEE*, Vol.83, No.4, pp.659-680, April 1995.

[102] M.Miranda, F.Catthoor, H.De Man, "Address equation optimization and hardware sharing for real-time signal processing applications", *IEEE workshop on VLSI signal processing*, La Jolla CA, Oct. 1994. Also in *VLSI Signal Processing VII*, J.Rabaey, P.Chau, J.Eldon (eds.), IEEE Press, New York, pp.208-217, 1994.

[103] M. A. Miranda, F. V. M. Catthoor, M. Janssen, and H. J. De Man. High-level address optimization and synthesis techniques for data-transfer-intensive applications. *IEEE Transactions on VLSI Systems*, 6(4):677–686, December 1998.

[104] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by a memory-aware architecture description language. In *VLSIDesign*, Bangalore, 2001.

[105] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–207, Monterey, CA, October 1992.

[106] —, The ISO/IEC Moving Picture Experts Group Home Page, http://www.cselt.it/mpeg/

[107] E. Musoll, T. Lang, and J. Cortadella. Working-zone encoding for reducing the energy in microprocessor address buses. *IEEE Transactions on VLSI Systems*, 6(4):568–572, December 1998.

[108] M.Neeracher, R.Rühl, "Automatic parallelization of LINPACK routines on distributed memory parallel processors", *Proc. IEEE Int. Parallel Proc. Symp.*, Newport Beach CA, April 1993.

[109] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *ICPP*, St. Charles, IL, 1993.

[110] D.A.Padua, M.J.Wolfe. "Advanced compiler optimizations for supercomputers", *Communications of the ACM*, Vol.29, No.12, pp.1184-1201, 1986.

[111] P. R. Panda and N. D. Dutt. Low-power memory mapping through reducing address bus activity. *IEEE Transactions on VLSI Systems*, 7(3):309–320, September 1999.

[112] 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, October 1997.

[113] P. R. Panda, N. D. Dutt, and A. Nicolau. Incorporating DRAM access modes into high-level synthesis. *IEEE Transactions on Computer Aided Design*, 17(2):96–109, February 1998.

[114] P. R. Panda, N. D. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on Computer Aided Design*, 18(1):3–13, January 1999.

[115] 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.

[116] P.R.Panda, "Memory bank customization and assignment in behavioral synthesis", *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pp.477-481, Nov. 1999.

[117] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3), July 2000.

[118] K.Parhi, "Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs", *Proc. IEEE Int. Symp. on Circuits and Systems*, Portland OR, pp.1923-1928, May 1989.

[119] N.Passos, E.Sha, "Full parallelism of uniform nested loops by multi-dimensional retiming", *Proc. Int. Conf. on Parallel Processing*, Vol.2, pp.130-133, Aug. 1994.

[120] N.Passos, E.Sha, L-F.Chao, "Multi-dimensional interleaving for time-and-memory design optimization", *Proc. IEEE Int. Conf. on Computer Design*, Austin TX, pp.440-445, Oct. 1995.

[121] M.Pauwels, F.Catthoor, D.Lanneer, H.De Man, "Type-handling in bit-true silicon compilation for DSP", *Proc. Europ. Conf. on Circ. Theory and Design*, ECCTD, Brighton, U.K., pp.166-170, Sep. 1989.

[122] C.Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on the architecture design", *IEEE Trans. on Computers*, Vol.37, No.8, pp.991-1004, Aug. 1988.

[123] W.Pugh, D.Wonnacott, "An evaluation of exact methods for analysis of value-based array data dependences", *6th Wsh. on Programming Languages and Compilers for Parallel Computing*, Portland OR, pp.546-566, August 1993.

[124] F.Quillere, S.Rajopadhye, "Optimizing memory usage in the polyhedral model", *Proc. Massively Parallel Computer Systems Conf.*, April 1998.

[125] L. Ramachandran, D. Gajski, and V. Chaiyakul. An algorithm for array variable clustering. In *European Design and Test Conference*, February 1994.

[126] J. Robinson. Efficient general-purpose image compression with binary tree predictive coding. *IEEE Transactions on Image Processing*, 6(4):601–608, Apr. 1997.

[127] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting dual data-memory banks in digital signal processors. In *International conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, October 1996.

[128] H. Schmit and D. E. Thomas. Synthesis of application-specific memory designs. *IEEE Transactions on VLSI Systems*, 5(1), March 1997.

[129] H. Schmit and D. E. Thomas. Address generation for memories containing multiple arrays. *IEEE Transactions on Computer Aided Design*, 17(5), May 1998.

[130] L. Semeria, K. Sato, and G. De 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, March 2000.

[131] B. Shackleford, M. Yasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura. Memory-cpu size optimization for embedded system designs. In *Design Automation Conference*, June 1997.

[132] W.Shang, E.Hodzic, Z.Chen, "On uniformization of affine dependence algorithms", *IEEE Trans. on Computers*, Vol.45, No.7, pp.827-839, July 1996.

[133] W.Shang, M.O'Keefe, J.Fortes, "Generalized cycle shrinking", presented at workshop on "Algorithms and Parallel VLSI Architectures II", Bonas, France, June 1991. Also in *Algorithms and parallel VLSI architectures II*, P.Quinton and Y.Robert (eds.), Elsevier, Amsterdam, pp.131-144, 1992.

[134] W.-T. Shiue and C. Chakrabarti. Memory exploration for low power embedded systems. In *Design Automation Conference*, pages 140–145, June 1999.

[135] W-T.Shiue, S.Tadas, C.Chakrabarti, "Low power multi-module, multi-port memory design for embedded systems", *Proc. IEEE Wsh. on Signal Processing Systems (SIPS)*, Lafayette LA, IEEE Press, pp.529-538, Oct. 2000.

[136] P.Slock, S.Wuytack, F.Catthoor, G.de Jong, "Fast and extensive system-level memory exploration for ATM applications", *Proc. 10th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, Antwerp, Belgium, pp.74-81, Sep. 1997.

[137] M. R. Stan and W. P. Burleson. Bus-invert coding for low power I/O. *IEEE Transactions on VLSI Systems*, 3(1):49–58, March 1995.

[138] L.Stok, J.Jess, "Foreground memory management in data path synthesis" *Int. Journal on Circuit Theory and Appl.*, Vol.20, pp.235-255, 1992.

[139] L.Stok, "Data path synthesis", *INTEGRATION, the VLSI journal*, Vol.18, pp.1-71, June 1994.

[140] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: a case study. In M. Pedram, R. Brodersen, and K. Keutzer, editors, *Proceedings of the 1995 International Symposium on Low Power Design*, pages 63–68, New York, NY, 1995. ACM Press.

[141] A. Sudarsanam and S. Malik. Simultaneous reference allocation in code generation for dual data memory bank asips. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):242–264, April 2000.

[142] Synopsys Inc., Mountain View, CA. *Behavioral Compiler User Guide*, 1997.

[143] L.Thiele, "On the design of piecewise regular processor arrays", *Proc. IEEE Int. Symp. on Circuits and Systems*, Portland OR, pp.2239-2242, May 1989.

[144] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for systems-on-chip design. In *Proceedings APCHDL'99*, Japan, 1999.

[145] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling for power reduction in processor-based system design. In *Design, Automation, and Test in Europe*, Paris, France, February 1998.

[146] H. Tomiyama and H. Yasuura. Size-constrained code placement for cache miss rate reduction. In *International Symposium on System Synthesis*, pages 96–101, La Jolla, CA, November 1996.

[147] H. Tomiyama and H. Yasuura. Code placement techniques for cache miss rate reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[148] C-J.Tseng, D.Siewiorek, "Automated synthesis of data paths in digital systems", *IEEE Trans. on Comp.-aided Design*, Vol.CAD-5, No.3, pp.379-395, July 1986.

[149] A.Vandecappelle, M.Miranda, E.Brockmeyer F.Catthoor, D.Verkest, "Global Multimedia System Design Exploration using Accurate Memory Organization Feedback" *Proc. 36th ACM/IEEE Design Automation Conf.*, New Orleans LA, pp.327-332, June 1999.

[150] I.Verbauwhede, F.Catthoor, J.Vandewalle, H.De Man, "Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips", *Proc. VLSI'89, Int. Conf. on VLSI*, Munich, Germany, pp.209-218, Aug. 1989.

[151] I.Verbauwhede, C.Scheers, J.Rabaey, "Memory estimation for high-level synthesis", *Proc. 31st ACM/IEEE Design Automation Conf.*, San Diego, CA, pp.143-148, June 1994.

[152] W.Verhaegh, P.Lippens, E.Aarts, J.Korst, J.van Meerbergen, A.van der Werf, "Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing", *IEEE Trans. on Computer-aided design*, Vol.14, No.8, Aug. 1995.

[153] W.Verhaegh, P.Lippens, E.Aarts, J.van Meerbergen, A.van der Werf, "Multi-dimensional periodic scheduling: model and complexity", *Proc. EuroPar Conference*, Lyon, France, August 1996. "Lecture notes in computer science" series, Springer Verlag, pp.226-235, 1996.

[154] P.R.Wilson, M.Johnstone, M.Neely, D.Boles, "Dynamic Storage Allocation: A Survey and Critical Review", *Proc. Intnl. Wsh. on Memory Management*, Kinross, Scotland, UK, Sep. 1995.

[155] M.Wolf, M.Lam, "A loop transformation theory and an algorithm to maximize parallelism", *IEEE Trans. on Parallel and Distributed Systems*, Vol.2, No.4, pp.452-471, Oct. 1991.

[156] M.Wolfe, "The Tiny loop restructuring tool", *Proc. of Intnl. Conf. on Parallel Processing*, pp.II.46-II.53, 1991.

[157] M.Wolfe, "High-performance compilers for parallel computing", Addison-Wesley, 1996.

[158] S.Wuytack, F.Catthoor, F.Franssen, L.Nachtergaele, H.De Man, "Global communication and memory optimizing transformations for low power systems", *IEEE Intnl. Workshop on Low Power Design*, Napa CA, pp.203-208, April 1994.

[159] S.Wuytack, J.P.Diguet, F.Catthoor, H.De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings", *IEEE Trans. on VLSI Systems*, Vol.6, No.4, pp.529-537, Dec. 1998.

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

[161] S.Wuytack, F.Catthoor, G.De Jong, and H.De Man. Minimizing the required memory bandwidth in vlsi system realizations. *IEEE Transactions on VLSI Systems*, 7(4):433–441, December 1999.

[162] C.Ykman-Couvreur, J.Lambrecht, D.Verkest, F.Catthoor, H.De Man, "Exploration and Synthesis of Dynamic Data Sets in Telecom Network Applications", *Proc. 12th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, San Jose CA, pp.125-130, Dec. 1999.

[163] Y.Zhao and S.Malik, "Exact Memory Size Estimation for Array Computation without Loop Unrolling", *36th ACM/IEEE Design Automation Conf.*, New Orleans LA, pp.811-816, June 1999.