# ADOPT: Efficient Hardware Address Generation in Distributed Memory Architectures

Miguel Miranda Francky Catthoor‡ Martin Janssen Hugo De Man‡
IMEC Lab. Kapeldreef 75, 3001 Leuven, Belgium. miranda@imec.be
‡Professor at the Katholieke Universiteit Leuven

## Abstract

*An address generation and optimization environment (ADOPT) for distributed memory architectures, is presented. ADOPT is oriented to minimize the area overhead introduced by the use of large numbers of customized address calculation units, needed to cope with the increasing bandwidth requirements of memory intensive real-time signal processing applications. Different high-level optimizing architectural alternatives are explored, such as algebraic optimizations and efficient data-path clustering and assignment, to minimize the space/time-multiplexed address unit cost. Furthermore, in order to significantly reduced the routing complexity, typically present in partitioned architectures, a methodology for the synthesis of a distributed architecture for hierarchical local controllers for address generation, is also proposed. The techniques presented are demonstrated on a realistic test-vehicle, showing significant savings on the overall addressing cost.*

## 1. Introduction

The problem of address generation in the majority of real-time multidimensional signal processing (RMSP) applications has to be dealt with in any design which involves indexed signals stored in background memories which are randomly accessed. Typically, these applications require large storage capacities and very high throughput to achieve real time constraints. The use of distributed memory architectures allows to exploit parallelism, thus alleviating memory access bottlenecks. However, as memory bandwidth increases, the cycle budget available for address generation becomes tighter. Specialized programmable hardware address generation units (pAGUs) [1] have been used to speed-up address expression evaluation. These units benefit from code generation optimization techniques, such as induction variable detection [2], usually associated to strength reduction techniques as in software compilers [3]. However, they are costly in terms of area since they include several arithmetical units, registers and/or counters to provide enough programmability. Therefore, the use of a large number of them in a distributed memory environment becomes prohibitive. Moreover, as the number of pAGUs grows, the cost associated to the program ROMs and instruction decoders also grows, reaching a point where the area overhead introduced by address generation becomes a dominant factor in the overall data path related design cost. This is true both in terms of area, as demonstrated by the large memory management units (MMUs) which reside on MPEG type designs, and in power.

An alternative to this approach is to use many small application-specific address generation units (AGUs). In the most straightforward and usually inefficient case, this leads to a direct mapping of the address equation (AE), associated to each individual index expression, onto dedicated address hardware blocks. This strategy can also lead to a large data path overhead if not executed well, but in this case the overhead only depends on the overall contribution of the local cost in the address generators. It does not increase the complexity of the instruction ROMs and the associated decoders. Furthermore, this overhead can in practice be significantly reduced by several optimizing alternatives such as the choice of different architectural styles [4] for AE realization, and by reusing hardware by both space and/or time-multiplexing techniques.

Most of the work found in literature is concentrated on bit-level optimizations for counter/table based architectures [5, 6, 7]. More recently, work on address generation for memories containing multiple arrays which benefits from techniques relying on the inversion or rearrangement of address bits for faster offset addition, has been reported [8]. Time sharing which we will show to be crucial to reduce the overall area cost is almost not considered. Only some interactive support for multiplexing has been introduced in the ZIPPO toolbox [9] which is mainly oriented to reuse hardware at bit-level. In contrast, a set of automatable system level techniques for the address part of the equation multiplexing in DSP has been presented, showing significant saving in area cost [4] for counter/table based architectures. In this paper, we extend the previous techniques to a methodology for efficient hardware address generation, supported by the **AD**ress equation **OPT**imization and generation environment, developed at IMEC. The methodology is based on a distributed organization of AGUs. Each AGU works in a stand-alone mode fully decoupled from the system controller. This organization allows to considerably reduce the routing complexity typically present in partitioned architectures.

## 2. Address generation context and target model

ADOPT supports two different architectural styles for the generation of the AEs. One possibility is to expand the different AEs in sequences which are then realized with counters with the outputs modified by custom logic [5, 4, 10]. There, every next address is computed incrementally, therefore the resulting hardware is known as **incremental Address Generation Unit** (iAGU). This style is well suited for very regular memory accesses, resulting in a mere binary/modulo counter and some inexpensive mapping logic.

The other possible AGU architecture supported by ADOPT is motivated by the fact that in RMSP applications, the AEs constitute a direct function of the loop iterators. In this case, the AE can be mapped onto arithmetical operators and only inexpensive adders/sustractors and/or barrel shifters are then needed to implement the constant multiplications. The resulting architecture is a **customized Address Calculation Unit** (ACU) realized with custom arithmetic building blocks selected from a library. This is the preferred architectural style for irregular memory accesses that consume too much mapping logic if implemented by an iAGU. Here, use is made of a lowly multiplexed data-path style and script [11, 12], optimized for these ACUs. Therefore, the choice between the two styles is important, as shown by our earlier work [4]. Both architectural styles are illustrated in Figure 1.

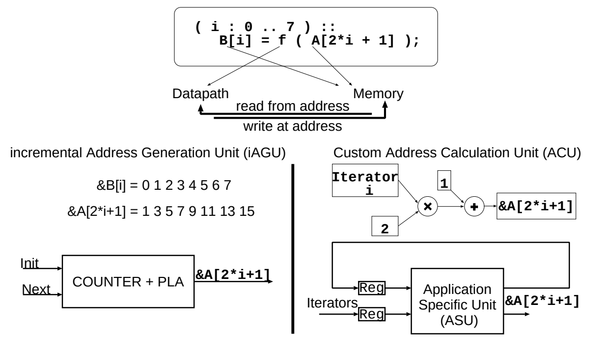
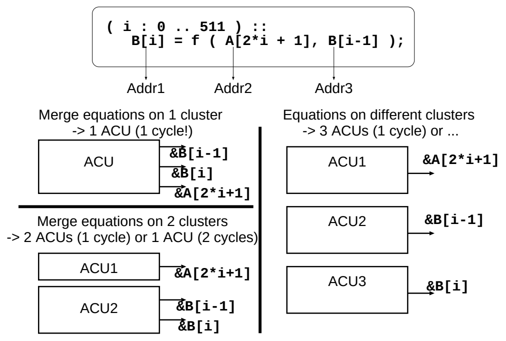


**Figure 1. Illustration of address optimization and hardware mapping task.**

Instead of performing scheduling first, followed by the traditional FU, mux and register allocation stages [13], we have shown that it is better to first identify similar clusters in the algorithm. For address generation, the clusters typically correspond to the AE and hundreds of them are present. These clusters are then assigned to shared data-path units in an optimized way [11], followed by the actual definition of the composition of these ACUs. The last two subtasks are very difficult to perform manually, so user support is needed here. Only then, more traditional detailed scheduling and multiplexer/foreground memory allocation takes place which are also very error-prone and time consuming for the designer.

### 2.1. Exploration in address generation

For both target styles it is possible to perform a number of high-level architectural optimizations, including in particular address equation/cluster splitting, merging and sharing. The assignment of two or more address equations/sequences to the same hardware unit creates possibilities for optimization by different algebraic transformations: common subexpression elimination, factorization, dead-code removal, etc. Here, depending on similarity measures, the merging happens whenever two address equations [14] or sequences [4] are sufficiently similar to motivate their evaluation on a common hardware not multiplexed in time. In this paper, we will concentrate on the customized ACU style for which a simple example of the different possibilities is shown in Figure 2. For realistic designs, a very large search space is typically available (see section 5).



( i : 0 .. 511 ) ::
B[i] = f ( A[2*i + 1], B[i-1] );
Addr1
Addr2
Addr3
Merge equations on 1 cluster
-> 1 ACU (1 cycle!)
ACU
&B[i-1]
&B[i]
&A[2*i+1]
Merge equations on 2 clusters
-> 2 ACUs (1 cycle) or 1 ACU (2 cycles)
ACU1
&A[2*i+1]
ACU2
&B[i-1]
&B[i]
Equations on different clusters
-> 3 ACUs (1 cycle) or ...
ACU1
&A[2*i+1]
ACU2
&B[i-1]
ACU3
&B[i]

**Figure 2. Illustration of options for address equation merging.**

When more cycles are available to realize the equations (which is usually the case in practice), also the freedom to use time-multiplexing should be explored. Use is made of efficient data-path clustering and assignment to minimize the time-multiplexed address unit cost. When two or more AEs are matching very well, they can be shared on the same, partly controllable address unit. This means that usually some multiplexers are present and that some building blocks operate under different modes (like a shifter which can multiply with $2^{-2}$ or $2^{-4}$). The similarity measure used to select the best matching address clusters is related to the one used for sharing (see [11]). The different sharing options are illustrated for a simple case in Figure 3. It should be stressed that this exploration is orthogonal to the merging search space so both optimizations can be combined.

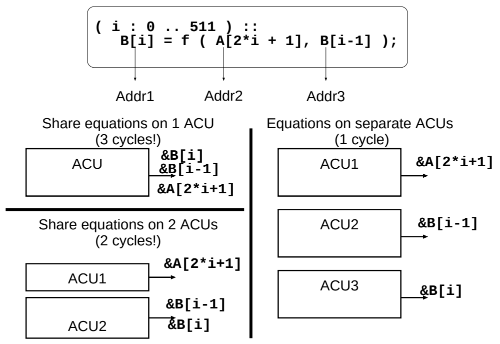


**Figure 3. Illustration of options for address equation sharing.**

## 3. Control strategies for address generation

Counter based architectures require two control lines generated from a global or a local controller to obtain the next memory address, and to provide an initial state to the AGU. For ACUs the iterator state values can be supplied either by the system controller (**absolute addressing**) or by a local controller (**incremental addressing**), thus contributing to the addressing cost. However, for iAGUs the local controller is somehow embedded. Note, that in most applications a mixed architectural style for addressing will be adopted in order to arrive at area efficient solutions. When selecting a controller strategy for address generation, three different styles can be identified for manifest memory accesses. Figure 4a shows a possible implementation obtaining the actual state of each iterator ($i, j, k$) for ACUs and the control lines for sequencing the counter/logic based units from the system controller. However, this alternative will lead to an unacceptable routing cost for complex designs, as the number of index expressions and iterators grows. The second style (Figure 4b) reduces partly the routing complexity by delegating the generation of the iterator state value to a local controller. However, the system controller still has to take care of sequencing the different memory accesses by distributing **res**et and **inc**rement signals among the different AGUs. The third style (Figure 4c) completely decouples the system controller from address generation units, and the addressing of memories and data-path is synchronized by means of the **system clock**. This last style is the one selected by ADOPT due to the possibility to provide parallelism at lower routing cost than the others two (see Figure 5). Furthermore, we also exploit the possibility of using an architecture based on distributed local controllers to localize the addressing hardware, hence reducing further the routing cost. Also, use of hierarchy is made when implementing the distributed controllers in order to reuse hardware and to optimize area overhead.

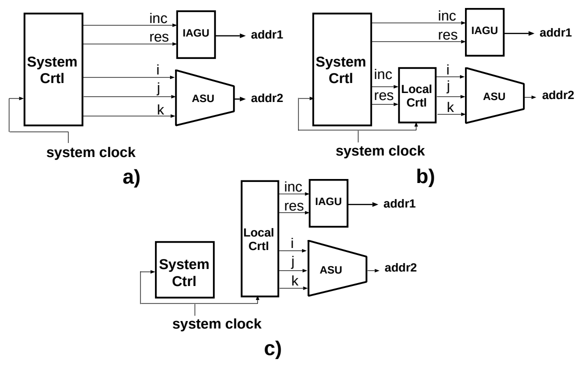


**Figure 4. Different controller strategies for hardware address generation.**

The proposed methodology for the synthesis of the local controllers is based on a sequential expansion of the nested loops which are present in the algorithm. The result is then expressed in a hardware description language like VHDL for subsequent logic synthesis. Figure 6 shows an example of how to model the behavior of the different iterators by means of a hierarchical FSM model. There, a projection of the nested structure over the abstract time is assumed, with the most inner iterator acting as the dimension where time is projected. Note that basically, every iterator is only in charge of detecting whether its immediate lower level neighbor is already at the upper bound, and only the most inner one is incremented along the abstract cycle time. The equivalent behavior of the resulting FSM can then be expressed by means of both Moore or Mealy machine models. In this way, it is very easy to translate any loop structure from the system level (algorithm) to an equivalent RTL level on a syntactical translation basis.

In Figure 6 the signal `NEXT` for controlling time evolution in the FSM description can be easily derived from the system cycle “clock” or it can be generated by another local controller. In the former case there is no hierarchy involved in the controller generation process (iterators `j,i` in Figure 6). The conditions needed to describe the activation of this signal in VHDL can be easily derived after distributing the overall cycle budget along the different nested loops. On the other hand, it is very well possible that in order to reuse controller hardware, a hierarchical controller matching the nested loop hierarchy is adopted (iterator `k` in Figure 6). In this case, the hierarchy of the resulting hardware can be easily provided at the behavioral level by selecting at which level of the loop hierarchy it is decided to stop the sequential expansion of loops. In this way every hierarchical section will generate the required control signal to activate the sequencing of the subsequent section.

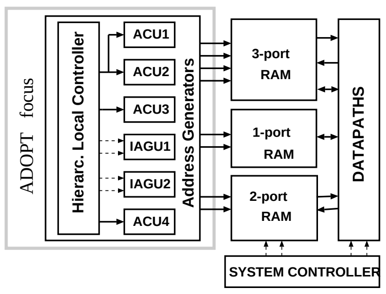
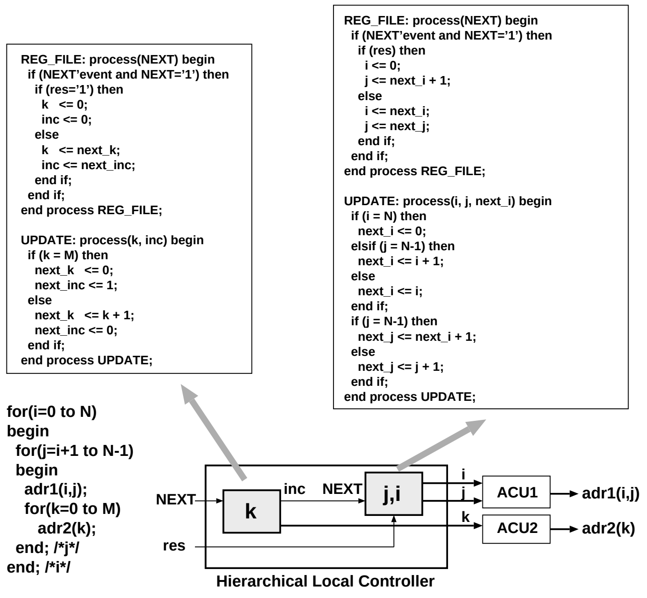


**Figure 5. Distributed memory architecture using ADOPT for address generation synthesis.**



```
REG_FILE: process(NEXT) begin
  if (NEXT'event and NEXT='1') then
    if (res='1') then
      k   <= 0;
      inc <= 0;
    else
      k   <= next_k;
      inc <= next_inc;
    end if;
  end if;
end process REG_FILE;

UPDATE: process(k, inc) begin
  if (k = M) then
    next_k   <= 0;
    next_inc <= 1;
  else
    next_k   <= k + 1;
    next_inc <= 0;
  end if;
end process UPDATE;
```

```
REG_FILE: process(NEXT) begin
  if (NEXT'event and NEXT='1') then
    if (res) then
      i <= 0;
      j <= next_i + 1;
    else
      i <= next_i;
      j <= next_j;
    end if;
  end if;
end process REG_FILE;

UPDATE: process(i, j, next_i) begin
  if (i = N) then
    next_i <= 0;
  elsif (j = N-1) then
    next_i <= i + 1;
  else
    next_i <= i;
  end if;
  if (j = N-1) then
    next_j <= next_i + 1;
  else
    next_j <= j + 1;
  end if;
end process UPDATE;
```

```
for(i=0 to N)
begin
  for(j=i+1 to N-1)
  begin
    adr1(i,j);
    for(k=0 to M)
      adr2(k);
  end; /*j*/
end; /*i*/
```

**Figure 6. Behavioral description for a hierarchical controller.**

This methodology can also be applied for the synthesis of the local controllers that generate the increment and reset signals needed in counter based architectures. Since the sequencing of the addresses is always synchronized with respect to the iterator that defines the scope where the corresponding AE is located. Therefore, when this iterator has to be incremented, a pulsed signal can be activated to increment the associated iAGU. The generation of the reset for the counter can be either synchronized with the global power-on reset or with the reset condition of the outermost iterator involved in the corresponding AE. Furthermore, it is also very straightforward to generate a local controller to be shared by different AGU architectures (iAGU and ACU), thus providing a very flexible control structure for mixed architectural styles within the same AE (see subsection 2.1).

## 4. Interaction between address generation, data-path synthesis and memory management

It is impossible to combine data storage, data-path and controller related tasks into a unique solution process since most of the individual tasks are NP-complete on their own, and the current CAD techniques are already very time-consuming for realistic designs. Combining more tasks into a single one would be fatal for complexity and run times. This clearly motivates the splitting of these issues in the full design methodology. The main motivations for why high-level memory management (HLMM) should take place before Address Generation (AG), data-path mapping (DPM) and controller synthesis (CS) are:

1.- The Area and Power effects of (array signal related) memory decisions are (much) larger than the data-path/control related issues. This has been shown on several designs in the past and is also confirmed by analyzing related literature [15, 16].

2.- The search space for AG, DPM and CS is not too heavily constrained by putting the HLMM issues first. It still leads to overall feasible solutions. This was illustrated for the DPM related issues in terms of parallelism introduction for meeting the throughput [17].

3.- The effect of the HLMM constraints on the actual outcome for AG, DPM and CS is relatively small, so near-optimal results are usually still feasible. Exceptions to this can only occur when relative expensive re-computation, look-ahead transformations and the like have to be performed, but this special case can be easily checked in advance.

A second though related issue is why AG is done after HLMM but before DPM. Actually, this is not fully necessary because for the custom data-path style it would be possible to combine both the (optimized) clustered address expressions and the data-path related clusters in one description which then needs to be assigned to ASUs and mapped to hardware (extended architecture synthesis path). Experience shows however that in most cases, this mixing of the two parts will not result in better designs.

It should be stressed though that when data-path and address generation issues are postponed, extensive use of (hardware and software) pipelining at the system level is required to end up with a feasible design. Still, this is in general much more effective than simply splitting the system in several subsystems (e.g. memory and data-path partition) and allocating part of the total cycle budget to each of them. Our relatively simple design approach allows to allocate the full cycle budget (with a small "interface tolerance") to the memory and I/O access. When the memory and communication organization is fixed, this will provide sufficient data at the appropriate "cycles" to the ports of the data-path black box. Within these bandwidth and ordering constraints, the data-path task can then be performed. If this task starts with the appropriate data-path and parallelism oriented transformations (see [17] for motivation, model and examples), the search space available to the data-path synthesis is still sufficient to arrive at both feasible and efficient architectures, even with stringent timing requirements.

## 5. Results for a Motion Estimation application

The methodology presented in this paper is tested on one of the two main nested loops of a full 2D motion estimation application. The algorithm is part of a video encoder application targeted to frame sizes of `256x256` pixels, with a `8x8` pixels macroblock size, and using an IDCT block of `4x4` pixels. Figure 7 describes a pseudo-code description of the selected algorithm. Note, the hierarchical loop structure and the complexity associated with the data-path related addressing which is much bigger than the other arithmetic and control related operations.

```
#define M 8
#define N 8
#define m 4
#define n 4
#define INF 4

(for T=1 to INF)
begin
  Delta=0;
  optDelta=0;
  (for j=-N to N)
  begin
    (for k_1=-m to -m-1)
    begin
      (for l_1=-n to n-1)
      begin
         if (T==1) then
            Ad[T][M][N*2+j][M+k_1][N*2+j+l_1]=0;
         else
            Ad[T][M][N*2+j][M+k_1][N*2+j+l_1]=A[T-1][M+k_1][N*2+j+l_1];
         end if;
      end;
      Delta=A[T][M*2-m][N*2-n]-Ad[T][M][N*2+j][M-m][N*2+j-n]+Delta;
    end;
    (for l_2=-n+1 to n-1)
      Delta=A[T][M*2-m][N*2+l_2]-A[T][M*2-m][N*2+l_2]+Delta;
    (for k_3=-m+1 to m-1)
    begin
      Delta=A[T][M*2+k_3][N*2-n]-Ad[T][M][N*2+j][M+k_3][N*2+j-n]+Delta;
      (l_3=-n+1 to n-1)
        Delta=A[T][M*2+k_3][N*2+l_3]-Ad[T][M][N*2+j][M+k_3][N*2+j+l_3]+Delta;
    end;
         optDelta=Delta+optDelta;
  end;
end;
```

**Figure 7. Pseudo-code description of the test-vehicle.**

Several experiments have shown the great impact that the different architectural alternatives can have over the overall cost figure. All the experiments mentionated here are constrained to generate every memory address in one clock cycle. Therefore, only AEs belonging to different loop scopes can be time-multiplexed on the same hardware unit.

The first experiment is targeted to obtain an overall area figure corresponding to an implementation where every AE is mapped into a separate customized ACU. Actually, this figure will serve as a reference for further optimizing alternatives. The controller strategy used for this example is based on a unique non-hierarchical local controller that is fed with the different iterator values to every ACU. Table 1 displays the area figure for every ACU as well as the area consumed by the local controller and the overall cost consumption. The area figures are in $mm^2$ and the target technology is 0.7$\mu m$ CMOS standard cell.

| Expression | Area |
|---|---|
| `1:Ad[T][M][N*2+j][M+k_1][N*2+j+l_1]` | 0.5949 |
| `2:A[T-1][M+k_1][N*2+j+l_1]` | 0.3221 |
| `3:A[T][M*2-m][N*2-n]` | 0.1181 |
| `4:Ad[T][M][N*2+j][M-m][N*2+j-n]` | 0.4337 |
| `5:A[T][M*2-m][N*2+l_2]` | 0.1587 |
| `6:Ad[T][M][N*2+j][M-m][N*2+j+l_2]` | 0.4871 |
| `7:A[T][M*2+k_3][N*2-n]` | 0.1680 |
| `8:Ad[T][M][N*2+j][M+k_3][N*2+j-n]` | 0.5415 |
| `9:A[T][M*2+k_3][N*2+l_3]` | 0.2906 |
| `10:Ad[T][M][N*2+j][M+k_3][N*2+j+l_3]` | 0.5949 |
| local controller | 0.1124 |
| total | 3.8200 |

**Table 1. Area results when mapping every AE in separated ACUs.**

Next, we wanted to see the effect on the area cost by merging different AEs into the same ACU. Here, the area saving is motivated by exploiting space-multiplexing possibilities among operators of very similar clusters (see 2.1) after performing global algebraic optimizations. A total number of 5 ACUs resulted after computing the similarity degree [14] among the different clusters. Table 2 shows the different groups of clusters (AEs) that were assigned to the same hardware unit. Here, the strategy used to generate the controller resulted in a distributed architecture of 3 local controllers with two of them implemented hierarchically (loc_ctrl_2 and loc_ctrl_3). Note, that about 29% in area saving is achieved by using merging techniques among similar clusters.

| Expression (merged) | Area | Expression (shared) | Area |
|---|---|---|---|
| ACU1: 1 ∪ 2 | 0.9167 | ACU1: 1+4+6+8+10 | 1.0933 |
| ACU2: 3 ∪ 5 | 0.1960 | ACU2: 2+3+5+7+9 | 1.1331 |
| ACU3: 4 ∪ 6 | 0.5411 | loc_ctrl_1: ACU1 | 0.1124 |
| ACU4: 7 ∪ 9 | 0.2518 | loc_ctrl_2: ACU2 | 0.1124 |
| ACU5: 8 ∪ 10 | 0.6489 | total | 2.4512 |
| loc_ctrl_1: ACU1 | 0.0705 | | |
| loc_ctrl_2: ACU2+ACU3 | 0.0581 | | |
| loc_ctrl_3: ACU4+ACU5 | 0.0729 | | |
| total | 2.6979 | | |

**Table 2. Area results after AE merging and sharing respectively.**

Table 2 also displays the area results when sharing the same harware unit among different AEs. Again, clusters with high similarity degree among them were selected as candidates for exploiting time-multiplexing possibilites. Note, that the maximum number of address equations that have to be evaluated at the same time per loop scope is two. Therefore a minimum of two ACUs are needed to provide all the address equations within one clock cycle by constraining the assignment of AEs to ACUs so that they don't share the same loop scope. The controller strategy consisted in this case in two non-hierarchical local controllers, each of them controlling one ACU. Again a significant saving in the total area overhead is achieved, a 35% this time.

However, the best results are obtained when combining both alternatives (merging and sharing) whenever an optimal assignment of clusters (AEs) to hardware units occurs. The optimal assignment is achieved by first grouping similar clusters for spatial optimization (hardware merging) followed by a second grouping of the resulting assignment for time-multiplexing optimization (hardware sharing). The first grouping resulted into four groups containing the best matching possibilities. Next, the time-multiplexing then has to deal with the optimal assignment of the four merged groups (and the remaining two AEs) onto two different ACUs. Again this process needs to be steered by the degree of similarity among groups to achieve maximum matching. Table 3 displays the area figures for the resulting ACUs and for the local controllers. There, $4 \cup 6$ represents a group of AEs (4 and 6) merged into the same cluster, and $(4 \cup 6)+(8 \cup 10)$ two clusters of merged AEs sharing the same hardware resource. Note that now the reduction of area overhead reaches a maximum 52% while still meeting a very tight cycle budget requirement (1 clock cycle per memory access).

| Expression | Area | Expression | Area |
|---|---|---|---|
| ACU1: 1+(4∪6)+(8∪10) | 0.9568 | loc_ctrl_1: ACU1 | 0.1124 |
| ACU2: 2+(3∪5)+(7∪9) | 0.6671 | loc_ctrl_2: ACU2 | 0.1124 |
| total | 1.7363 | | |

**Table 3. Area results after optimal merging and sharing of different AE into ACUs.**

## 6 Conclusions

A novel methodology for the generation and optimization of address equations for distributed memory architectures within tight cycle budgets, is presented. The methodology is based on different high-level optimizing architectural alternatives such as algebraic optimizations of the address expressions, and efficient data-path clustering and assignment to minimize the space/time-multiplexed address unit cost. In order to considerably reduce the controller routing complexity, a methodology for the synthesis of locally distributed hierarchical controllers targeted to address generation is proposed. The principles have been demonstrated on a realistic test-vehicle, obtaining a significant reduction in area overhead, and very promising results for complex applications.

## References

[1] K.Kitagaki, T.Oto, T.Demura, Y.Araki, and T.Takada. A new address generation unit architecture for video signal processing. *Visual Cmmunications and Image Processing'91:Image Processing*, Vol. 1606:891–900, 1991.

[2] M.Gerleck, E.Stoltz, and M.Wolfe. Beyond induction variables: detecting and clasifying sequences using a demand-driven ssa form. *ACM Trans. Programming Languages and Systems*, 17(1):85–122, Jan. 1995.

[3] A.Aho, R.Sethi, and J.Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.

[4] M.Miranda, F.Catthoor, and H.De Man. Address equation multiplexing for real time signal processing applications. In *VLSI Signal Processing, VII*, pages 188–197, New York, 1994. J. Rabaey, P. Chau and J. Eldon (eds.). IEEE Press.

[5] D.Grant, P.Denyer, and I.Finlay. Synthesis of address generators. In *Proc. IEEE Intl. Conf. Comp. Aided Design*, pages 116–119, Santa Clara CA, Nov. 1989.

[6] D.Grant and P.Denyer. Address generation for array access based on modulus m counters. In *Proc. 2nd ACM/IEEE Europ. Design Automation Conf.*, pages 118–122, Amsterdam, The Netherlands, Feb. 1991.

[7] P.Lippens, J.Van Meerbergen, A.Van der Werf, W.Verhaegh, B.McSweeney, J.Huisken, and O.McArdle. Phideo: A silicon compiler for high speed algorithms. In *Proc. 2nd ACM/IEEE Europ. Design Automation Conf.*, pages 436–441, Amsterdam, The Netherlands, Feb. 1991.

[8] H.Schmit and D.Thomas. Address generation for memories containing multiple arrays. In *Proc. IEEE Intl. Conf. on Computer-Aided Design*, pages 510–514, San Jose, CA, Nov. 1995.

[9] D.Grant, J.Van Meerbergen, and P.Lippens. Optimization of address generator hardware. In *Proc. 5th ACM/IEEE Europ. Design and Test Conf.*, pages 325–329, Paris, France, Feb. 1994.

[10] B.Vanhoof, I.Bolsens S.De Troch, L.Philips, J.Vanhoof, and H.De Man. Design of a voice coder with the cathedral-II silicon compiler. In *Proc. of the User Forum and EuroASIC prizes, EDAC-EuroASIC-93*, pages 150–153, Paris, France, Feb. 1993.

[11] W.Geurts, F.Catthoor, and H.De Man. Time constrained allocation and assignment techniques for high throughput signal processing. In *Proc. 29th ACM/IEEE Design Automation Conf.*, pages 124–127, Anaheim, CA, June 1993.

[12] S.Note, W.Geurts, F.Catthoor, and H.De Man. Cathedral iii: Architecture driven high-level synthesis for high throughput dsp applications. In *Proc. 28th ACM/IEEE Design Automation Conf.*, pages 597–602, San Francisco CA, June 1991.

[13] M.C.McFarland, A.C.Parker, and R.Camposano. The high-level synthesis of digital systems. *Proc. of the IEEE, special issue on "The future of computer-aided design*, 78(2):301–318, Feb. 1990.

[14] J.M.Janssen, F.Catthoor, and H.De Man. A specification invariant technique for operation cost minimization in flow-graphs. In *Proc. 7th ACM/IEEE International Symposium on High-level Synthesis*, pages 146–157, Niagara-on-the-Lake, Canada, May 1994.

[15] I.Verbauwhede, F.Catthoor, J.Vandewalle, and H.De Man. Background memory management for the synthesis of algebraic algorithms on multi-processor dsp chips. In *Proc. VLSI'89, Intl. Conf. on VLSI*, pages 209–218, Munich, Germany, Aug. 1989.

[16] T.H.Meng, B.Gordon, E.Tsern, and A.Hung. Portable video-on-demand in wireless communication. special issue on "Low power design" of the *Proceedings of the IEEE*, 83(4):659–680, April 1995.

[17] F.Catthoor, W.Geurts, and H.De Man. Loop transformation methodology for fixed-rate video, image and telecom processing applications. In *Proc. Intl. Conf. on Application-Specific Array Processors*, pages 427–438, San Francisco CA, Aug. 1994.