Systematic Address and Control Code Transformations for Performance Optimisation of a MPEG-4 Video Decoder

M. Palkovic M. Miranda K. Denolf P. Vos F. Catthoor[+]
IMEC, Kapeldreef 75, 3001 Leuven, Belgium.
[+] Also Professor at Katholieke Univ. Leuven.
{palkovic,miranda,denolf,vosp,catthoor}@imec.be

Abstract

A cost-efficient realisation of an advanced multimedia system requires high-level memory optimisations to deal with the dominant memory cost. This typically results in more efficient code for both power and system bus load. However, also significant performance improvement can be achieved when carefully optimising the address functionality. This paper shows how the nature of this addressing code and the related control flow allows to transform the complex index, iterator and condition expressions into efficient arithmetic. We apply our ADdress OPTimisation (ADOPT) design technology to a low power memory optimised MPEG-4 decoder. When mapped on popular programmable multimedia processor architectures, we obtain a factor of 2 in performance gain.

1. Introduction

New multimedia systems based on novel compression standards typically use a large amount of data storage and transfers. However, in an embedded system, this memory space and bus load consumes most of the energy (between 50 and 80%) [8]. Therefore, optimising the global memory accesses of an application in a so-called Data Transfer and Storage Exploration (DTSE) is a crucial task for achieving effective low power and low cost realisations [4].

DTSE has a positive effect on both energy and system bus performance, largely independent of the platform where the application is being mapped on. Some steps of the DTSE methodology introduce new addressing code, containing more complex (non-linear) arithmetic than the initial one. This type of functionality is a bottleneck for the linear nature of all commercial Address Calculation Unit (ACU) architectures. Moreover, pointer level specific address optimisation techniques, like those found in conventional compilers [1] or state-of-the-art automated address optimisation approaches [13, 14, 6, 17, 18] are ineffective for non-linear indexing.

Our address optimisation methodology, implemented in the new ADOPT environment [9, 7], efficiently removes this bottleneck. The techniques are based on the use of source code level transformations, largely independent of the targeted instruction-set architecture. These optimisations also need to be complemented by more architecture specific ones. Using these techniques, the addressing code and related control flow can be optimised at the global scope, resulting in a relevant gain in execution cycles.

This paper focuses on the optimisation of the addressing code and related control flow of a MPEG-4 video decoder (described in Section 2). As the application is memory optimised [5], the gain in execution cycles due to the optimised addressing arithmetic is not hidden by data transfer overhead. Section 3 illustrates the systematic application of our high-level address optimisation technique on the MPEG-4 video decoder. When mapped on a Pentium III and TriMedia TM1000 platform, these optimisations lead to an overall performance gain as reported in Section 4.

The contributions of this paper are the relevant performance gain for a memory optimised MPEG-4 video decoder [5], running on popular multimedia processors and the systematic code transformation methodology aiming for performance gain in both address related arithmetic and control flow. Such systematic approach allows to increase design productivity when compared to more conventional ("ad-hoc") code optimisation approaches.

2. Application driver description and analysis

The MPEG-4 (natural visual) video decoder [15] is a block-based algorithm exploiting temporal and spatial redundancy in subsequent frames. A MPEG-4 Visual Object Plane (VOP) is a time instance of a visual object (i.e. frame). A decompressed VOP is represented as a group of MacroBlocks (MBs). Each MB contains six blocks of 8 x

8 pixels: 4 luminance (Y), 1 chrominance red (Cr) and 1 chrominance blue (Cb) blocks. Figure 1 shows a simple profile decoder, supporting rectangular intra-coded (I-VOP) and predictive coded (P-VOP) VOPs. An I-VOP contains only independent texture information, decoded separately by inverse quantisation and IDCT scheme. A P-VOP is coded using motion compensated prediction from the previous P or I VOP. Reconstructing a P-VOP implies adding a motion compensated VOP and a texture decoded error VOP which is a computationally and memory intensive operation [5, 12].

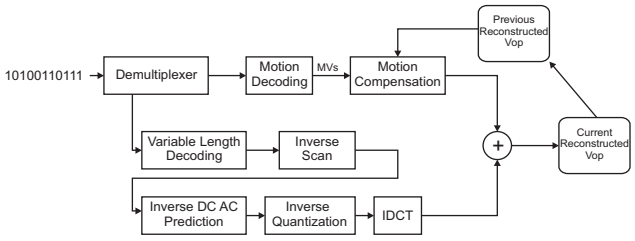


Figure 1. MPEG-4 simple profile natural visual decoding [5]

Due to flexibility requirements of the MPEG-4 standard, an instruction-set processor architecture is preferred over a fully custom one [2]. Meeting real-time constraints in such architectures, requires an optimal mapping of the application on the target architecture. At IMEC, a considerable effort has been spent on optimising the MPEG-4 decoder code at the C level for memory usage [5]. Nevertheless, important optimisation techniques oriented to e.g. optimise the addressing arithmetic of this application are not explored yet and are the focus of this paper.

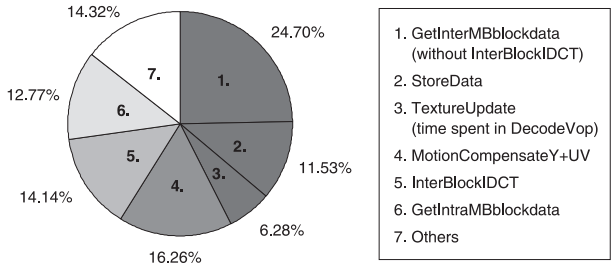
To bound the exploration space and select appropriate candidates for our methodology (see Section 3), the functions with an important cycle-count are targeted by measuring the time of the function and its descendants. It is necessary to focus on both characteristics because functions can be either time consuming or often called. The analysis is performed on Pentium III using the Rational Quantify environment [11].

The identified application core loops over two conditional branches as shown in Figure 2. After applying memory related transformations, the "if" branch of the application core takes about a 65% of execution time, where the "else" branch takes only about a 15% of execution time for a test frame sequence of average complexity.

Four functions are selected as the most time consuming of our application (see Figure 2): the IDCT computation called by the `GetInterMBblockdata` function, the bitstream parsing called by many underlying functions, the ACDC reconstruction called by the `GetIntraMBblockdata` function and the motion compensation (`MotionCompensateY+UV` functions and their descendants).

```
DecodeVop()
{
  ...
  loop
  {
    ...
    if(expr) {    // This branch takes a 65% of
                  // the total number of cycles.
      if (comp < 4)
        MotionCompensateY();
      else if (comp == 4)
        MotionCompensateUV();
      else /*comp == 5*/
        MotionCompensateUV();
      GetInterMBblockdata();
      ...
      StoreData();
    }
    else {        // This branch takes a 15% of
                  // the total number of cycles.
      GetIntraMBblockdata();
      StoreData();
    }
    ...
  }
  ...
}
```

(a)



(b)

Figure 2. (a) Illustration of MPEG-4 video decoder core; (b) Breakdown in execution time for Pentium III @500MHz

3. High-level address and control flow optimisation of the MPEG-4 video decoder

Our high-level address optimisation script comprises two major steps, relatively independent of the processor architecture. Both are source level code transformations. The first stage reduces the amount of address arithmetic related operations as well as the number of times these are executed (i.e. it reduces the redundant address arithmetic present in the code) [9]. During the second stage, the strength or complexity of the address arithmetic is reduced by transforming the complex operations into simpler ones for execution while preserving the functionality [7].

This section describes our optimisation script for the `GetIntraMBblockdata` function and its descendants. Similar optimisation steps are applied to the `GetInterMBblockdata` function and its descendants.

Before applying the different address optimisation steps, all multi-dimensional arrays need to be single-dimensional. During this step, a benefit is obtained in execution time

by avoiding a three-level memory indirection dependency chain for the fetched data. Finally, the address code is extracted.

3.1. Exposing the opportunities for address arithmetic optimisation by means of control flow transformations

Most multimedia applications are typically structured in a similar program hierarchy: functions containing loops with conditionals inside. As this hierarchy hides the opportunities for program transformations, especially for the address related code, it is crucial to increase the initially available search space in the code for arithmetic optimisations (arithmetic cost minimisation, loop-invariant code motion, etc.). The search space of the video decoder kernel is increased at different places.

The addressing code containing arrays of constants within the index expressions hides factorisation opportunities. For instance, Figure 3 illustrates one case where the presence of the `Xtab`, `Ytab`, `Ztab`, `Xpos` and `Ypos` arrays of constant values block the possibilities for finding common factors along the different loop iterations.

```
int *DC_store;

DC_store = (int*)malloc(LB*6*15*sizeof(int));
...

int Xtab[6] = { 1, 0, 3, 2, 4, 5 };
int Ytab[6] = { 2, 3, 0, 1, 4, 5 };
int Ztab[6] = { 3, 2, 1, 0, 4, 5 };
int Xpos[6] = {-1, 0,-1, 0,-1,-1 };
int Ypos[6] = {-1,-1, 0, 0,-1,-1 };

loop
{
  block_A = comp == 1 || comp == 3 ?
  DC_store[0 + 15*Xtab[comp] + 90*(((mb_num/MB)*MB
  + (mb_num%MB + Xpos[comp]))%LB)] : mg*8;
  block_B = comp == 3 ? DC_store[0 + 15*Ztab[comp]
  + 90*(((mb_num/MB + Ypos[comp])*MB
  + (mb_num%MB + Xpos[comp]))%LB)] : mg*8;
  block_C = comp == 2 || comp == 3 ? DC_store[0
  + 15*Ytab[comp] + 90*(((mb_num/MB + Ypos[comp])*MB
  + mb_num%MB)%LB)] : mg*8;

  mb_num++;
}
```

Figure 3. Piece of code from the MPEG-4 video decoder core with conditionals limiting the search space for address arithmetic optimisation.

In order to substitute the elements of the arrays by their actual values, it is necessary to "unfold" the conditions depending on the variables indexing these, i.e. the induction variable `comp`. The focus is first on the conditional structure present in the body of the loop. In the example of Figure 3, several redundant condition tests are present, e.g. for `comp=3`, all three condition expressions are tested and executed. Such condition structure of the code causes a control flow overhead and limits the search space (for propagation of constants contained in arrays). To avoid this, the control flow is organised to test the `comp` value only once and to collect the bodies for the same `comp` value together (see Figure 4). In the next step, the propagation of constants of arrays for `Xtab`, `Ytab`, `Ztab`, `Xpos` and `Ypos` is performed.

```
int Xtab[6] = { 1, 0, 3, 2, 4, 5 };
int Ytab[6] = { 2, 3, 0, 1, 4, 5 };
int Ztab[6] = { 3, 2, 1, 0, 4, 5 };
int Xpos[6] = {-1, 0,-1, 0,-1,-1 };
int Ypos[6] = {-1,-1, 0, 0,-1,-1 };

loop
{
  if (comp==1) {
    block_A = DC_store[0 + 15*Xtab[comp]
    + 90*(((mb_num/MB)*MB
    + (mb_num%MB + Xpos[comp]))%LB)];
    block_B = block_C = mg*8;
  }
  else if (comp==2) {
    block_A = block_B = mg*8;
    block_C = DC_store[0 + 15*Ytab[comp]
    + 90*(((mb_num/MB + Ypos[comp])*MB
    + mb_num%MB)%LB)];
  }
  else if (comp==3) {
    block_A = DC_store[0 + 15*Xtab[comp]
    + 90*(((mb_num/MB)*MB
    + (mb_num%MB + Xpos[comp]))%LB)];
    block_B = DC_store[0 + 15*Ztab[comp]
    + 90*(((mb_num/MB + Ypos[comp])*MB
    + (mb_num%MB + Xpos[comp]))%LB)];
    block_C = DC_store[0 + 15*Ytab[comp]
    + 90*(((mb_num/MB + Ypos[comp])*MB
    + mb_num%MB)%LB)];
  }
  else
    block_A = block_B = block_C = mg*8;

  mb_num++;
}
```

Figure 4. Piece of code from the MPEG-4 video decoder core with rewritten control-flow aiming to increase the search space.

3.2. Removing redundancy in the address arithmetic code

The address arithmetic code obtained after the address extraction step is still characterised by complex address expressions as shown in Figure 4. During ADOPT's Arithmetic Cost Minimisation (ACM) step, the expressions can be factorised and simplified by exploiting algebraic and modulo transformations. The goal is to minimise the number of operation instances. For our application, this step involves, among other classical optimisations, constant propagation, dead code elimination and common sub-expression elimination, all performed at the global scope. For other applications, it is sometimes necessary to re-factorise the initial arithmetic to expose the opportunities for such optimisations [7]. Before applying this step, it is important to expose the search space for arithmetic exploration properly (see Section 3.1).

After the propagation of the constants of arrays `Xtab`, `Ytab`, `Ztab`, `Xpos` and `Ypos`, the different common sub-expressions within the control-flow scopes are detected and stored in variables (i.e. `cse`) as shown in Figure 5.

```
loop
{
  if (comp==1) {
    block_A = DC_store[90*(mb_num%LB)];
    block_B = block_C = mg*8;
  }
  else if (comp==2) {
    block_A = block_B = mg*8;
    block_C = DC_store[90*(mb_num%LB)];
  }
  else if (comp==3) {
    cse = 90*(mb_num%LB);
    block_A = DC_store[30 + cse];
    block_B = DC_store[cse];
    block_C = DC_store[15 + cse];
  }
  else
    block_A = block_B = block_C = mg*8;

  mb_num++;
}
```

Figure 5. Illustration of ADOPT's ACM step in the MPEG-4 video decoder

This step is complemented by an Aggressive Code Hoisting (ACH) step aiming at removing unnecessary common computations of sub-expressions. These are hoisted across loops and overlapping control flow branches as high as possible in the loop and control flow hierarchy [9].

3.3. Levering the complexity of modulo address operations

The last step, Non-linear Operator Strength Reduction (NOSR), focuses on substituting the non-linear expressions that generates piece-wise linear address sequences by a combination of conditionals and induction variables. In our example the expression `mb_num%LB` (see Figure 5) is one of these cases, where `mb_num` is a loop iterator and `LB` is a data dependent value. This expression can be replaced by an induction variable `p_mb_num` that is conditionally reset whenever it reaches the `LB` value (see Figure 6) [7]. Note that although `LB` is data dependent, its value is defined at the start of a loop and it remains constant during the whole loop execution.

3.4. Matching the target instruction set ACU and controller execution

Processor specific transformations are necessary to match the addressing code to the targeted ACU instruction set architecture. This stage contains both data (arithmetic) and control-flow oriented transformation types. As the Pentium processor is a highly pipelined architecture, control flow transformations aiming at the reduction of the amount of branch predictions are very important.

The MPEG-4 video decoder has a condition, with large corresponding branches in terms of instructions. It is contained in the main loop of the video decoder kernel (see Figure 2, Figure 7a). As the condition expression is independent on the loop (always the same test is done over each loop iteration), it is possible to push the loop down across

```
p_nb_num = 0;
loop
{
  if (p_nb_num>=LB)
    p_nb_num-=LB;

  if (comp==1) {
    block_A = DC_store[90*p_nb_num];
    block_B = block_C = mg<<3;
  }
  else if (comp==2) {
    block_A = block_B = mg<<3;
    block_C = DC_store[90*p_nb_num];
  }
  else if (comp==3) {
    cse = 90*p_nb_num;
    block_A = DC_store[30 + cse];
    block_B = DC_store[cse];
    block_C = DC_store[15 + cse];
  }
  else
    block_A = block_B = block_C = mg<<3;

  p_mb_num++;
}
```

Figure 6. Illustration of ADOPT's NOSR (piece-wise linear) step in the MPEG-4 video decoder

conditional branches without violating data dependencies. The goal is to perform the iterations of the loop inside the branches and not outside (see Figure 7b). In this way, the impact of the branch prediction is minimised, since the prediction is happening only once instead of for every iteration of the loop.

```
loop {
  if (expr) {
    body1
  }
  else {
    body2
  }
}
     (a)
```

```
if (expr) {
  loop
    body1
}
else {
  loop
    body2
}
     (b)
```

Figure 7. Illustration of processor specific transformation in the MPEG-4 video decoder core: (a) initial code; (b) transformed code

Other type of processor specific transformations incorporated for Pentium III and TriMedia TM1000 comprise the use of customised library functions like `memcpy` and `memset` and the use of an auto-increment pointer feature. Calling highly optimised library functions for setting and copying large data blocks brings a significant gain. However, this approach cannot always be applied since it requires a row-major access pattern in both source and destination arrays.

4. Experimental framework and results

This section, measures the performance impact of the address and control flow optimisations. The memory optimised code, obtained by applying the Data Transfer and Storage Exploration (previously reported in [5]) on the MPEG-4 verification model, is used as the reference for this work.

4.1. Test sequences description

The testbench contains three video bitstreams with different motion complexity. Mother and Daughter (m&d) is a typical head and shoulders sequence with little motion, Foreman (for) is a real life sequence with medium motion and Calendar and Mobile (c&m) is a highly complex sequence with multiple, heterogeneous motion. All three sequences are heavily dominated by P-VOPs. Therefore, the ACDC reconstruction (see Section 2) has a much smaller impact in performance that the motion compensation itself [12].

Table 1. Characteristic of the testbench video bitstreams

Test case	Number of VOPs	Bitrate (kbps)	Encoded framerate (fps)
Mother and Daughter	300	120	30
Foreman	300	450	25
Calendar and Mobile	150	2000	15

Table 1 lists the number of VOPs, bitrate (kilobits per second) and the encoded framerate (frames per second). The bitrate (kbps) also gives an idea of the complexity required during the decoding phase. The smaller the number of kbps while encoding, the smaller the effort required on the decoding phase. The results are for the CIF (Common Interchange Format) (358 x 288) image size. On both platforms, the native compilers with all low-level optimisations enabled are used for the experiments. This makes our transformations work on top of the traditional low-level compiler optimisations.

4.2. Speeding up the MPEG-4 video decoder by address optimisations

To better illustrate the impact of the ADOPT steps, the memory optimised reference code is split into an Original and a DTSE version (see Figure 8). On Trimedia TM1000, this breakdown is not available.

The memory bottleneck initially present in the Original version results in a large power and system bus load overhead. Applying DTSE removes this and consequently results in an improved performance.

However, our formalised ADOPT steps bring a crucial gain when applied as a next optimisation stage (the Addr.Opt. version), especially when meeting real-time constraints is an issue (e.g. 30 frames per second (fps) for m&d). In Figure 8, this gain varies from 60% to 40% of the total, decreasing with the input stream complexity.

With exactly the same testbench, our transformations result in a performance improvement of more than a factor of two on top of the optimisation effect described by Denolf *et al.* [5]. The authors report similar performance as

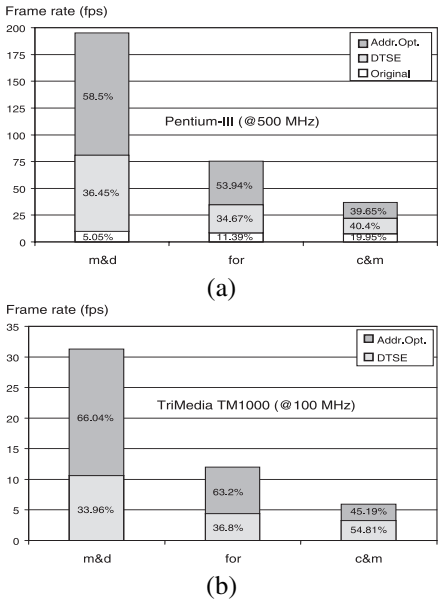


Figure 8. Breakdown in frame rate due to optimisations performed at IMEC: (a) Pentium III; (b) TriMedia TM1000

the one obtained in the state-of-the-art [3, 10], stressing the non-straightforward character of the comparison. The performance logically depends on the platform and the coding characteristics of the test sequences. Unfortunately, insufficient details about the testbench in [3, 10] are provided to make a detailed comparison.

4.3. Breakdown between the different address optimisation stages

During the address optimisation stage, several types of optimisations are applied. In general, the gain can be decomposed in two main stages (see Figure 9): an architecture independent and an architecture specific address optimisation stage.

The architecture independent stage is further divided in two substages: one collecting both ACM+ACH stages, the other reporting the gain obtained by applying our NOSR stage to replace the complex modulo operations by a combination of pointers and conditions.

The processor independent stage is followed by a processor specific address arithmetic optimisation stage which brings significant gain (50% over the total address optimisation gain for m&d test sequence). However, its impact decreases with the test sequence complexity increase (35% for for and 30% for c&m over the total address optimisation gain).

When comparing the breakdown obtained for TriMedia TM1000 with the one for Pentium III, a higher impact of

ADOPT's NOSR step is observed. This is due to the lack of modulo acceleration in the TriMedia TM1000 processor that could eventually speed-up the computation of the these operations [16]. Also, the architecture specific optimisations have a larger impact in this architecture, due to the larger variety of functional units (up to 27) [16] when compared to the Pentium III architecture. This clearly shows the importance of adapting the code to the instruction set of the targeted architecture as modern multimedia processors have highly specialised instructions.

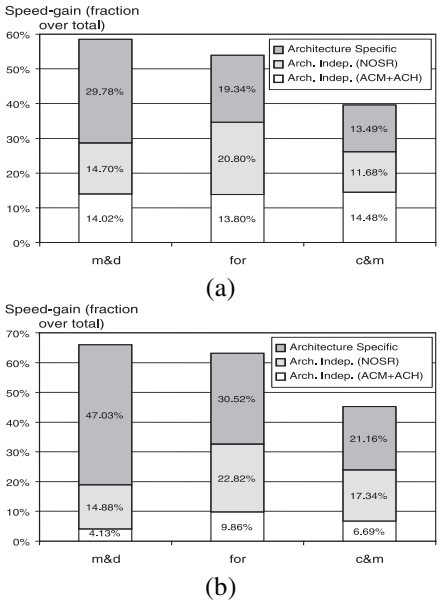


Figure 9. Breakdown in speed up due to address optimisations: (a) Pentium III; (b) TriMedia TM1000

5. Conclusions

A memory optimised MPEG-4 video decoder offers freedom to exploit address optimisations at the source level code. We have systematically applied our high-level address optimisation methodology on this application. As a result, the execution time of the most critical function is reduced on average by a factor of two for the sequences and the implementation platforms select. This has lead to a considerable overall speed-up when compared to the reported state-of-the-art results.

References

[1] A. V. Aho, R. Sethi, and J. D. Ulmann. *Compilers: principles, techniques and tools*. Addison-Wesley, 1986.

[2] S. Bauer et al. The MPEG-4 Multimedia Coding Standard: Algorithms, Architectures and Applications. *Journal of VLSI Signal Processing*, 23(1):7–26, October 1999.

[3] F. Casalino, G. Di Cagno, and R. Luca. MPEG-4 Video Decoder Optimisation. In *Proc. IEEE International Conference on Multimedia Computing and Systems*, volume 1, pages 363–368, Los Alamitos, US, 1999.

[4] F. Catthoor, K. Danckaert, C. Kulkarni, and T. Omnes. *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, chapter Data transfer and storage architecture issues and exploration in multimedia processors. Marcel Dekker, Inc., New York, 2000.

[5] K. Denolf, P. Vos, J. Bormans, and I. Bolsens. Cost-efficient C-Level Design of an MPEG-4 Video Decoder. In *Workshop on Power and Timing Modeling, Optimization and Simulation*, Goettingen, Germany, 13-15 September 2000.

[6] C. Gebotys. DSP address optimisation using a minimum cost circulation technique. In *Proc. Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 100–104, 1997.

[7] C. Ghez, M. Miranda, A. Vandecappelle, F. Catthoor, and D. Verkest. Systematic high-level Address Code Transformations for Piece-wise Linear Indexing: Illustration on a Medical Imaging Algorithm. In *Proc. IEEE Workshop on Signal Processing Systems (SIPS2000)*, Lousiana, USA, 2000.

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

[9] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta. Analysis of high-level address code transformations for programmable processors. In *Proc. ACM Design Automation & Test in Europe Conf. (DATE)*, Paris, March 2000.

[10] G. Hovden and N. Ling. On Speed Optimisation of MPEG-4 Decoder for Real-Time Multimedia Applications. In *Proc. 3rd International Conference on Computational Intelligence and Multimedia Applications (ICCIMA'99)*, volume 1, pages 399–402, Los Alamitos, US, 1999.

[11] http://www.rational.com/products/vis_quantify/index.jtmpl.

[12] J. Kneip, B. Schmale, and H. Moller. Applying and Implementing the MPEG-4 Standard. *IEEE Micro*, 19(6):64–74, Nov.-Dec. 1999.

[13] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Proc. Intl. Conf. on Computer-Aided Design (ICCAD)*, 1996.

[14] C. Liem, P. Pauling, and A. Jerraya. Address calculation for retargetable compilation and exploration for instruction-set architectures. In *Proc. Design Automation Conference (DAC)*, pages 597–600, 1996.

[15] MPEG subgroups. MPEG-4 Overview — (V.18 — Singapore Version). ISO/IEC JTC1/SC29/WG11 N4030, Singapore, March 2001. http://www.cselt.it/mpeg/standards/mpeg-4/mpeg-4.htm.

[16] Philips Electronics North America Corporation. *TM1000 Preliminary data book*, 1997.

[17] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Proc. Design Automation Conference (DAC)*, 1997.

[18] S. Udayanarayanan and C. Chakrabarti. Address Code Generation for Digital Signal Processors. In *Proc. Design Automation Conference (DAC)*, pages 353–358, 2001.