Analysis of High-level Address Code Transformations for Programmable Processors

Sumit Gupta[‡] Miguel Miranda[§] Francky Catthoor[§ ¶] Rajesh Gupta[‡]

[‡]Department of Information and Computer Science, Univ. of California at Irvine.
[§]IMEC Lab., Kapeldreef 75, 3001 Leuven, Belgium. [¶]Also Professor at Katholieke Univ. Leuven.

Abstract

Memory intensive applications require considerable arithmetic for the computation and selection of the different memory access pointers. These memory address calculations often involve complex (non)linear arithmetic expressions which have to be calculated during program execution under tight timing constraints, thus becoming a crucial bottleneck in the overall system performance. This paper explores applicability and effectiveness of source-level optimisations (as opposed to instruction-level) for address computations in the context of multimedia. We propose and evaluate two processor-target independent source-level optimisation techniques, namely, global scope operation cost minimisation complemented with loop-invariant code hoisting, and non-linear operator strength reduction. The transformations attempt to achieve minimal code execution within loops and reduced operator strengths. The effectiveness of the transformations is demonstrated with two real-life multimedia application kernels by comparing the improvements in the number of execution cycles, before and after applying the systematic source-level optimisations, using state-of-the-art C compilers on several popular RISC platforms.

1. Introduction

Memory intensive applications (e.g., multimedia) require considerable arithmetic for the computation and selection of the different memory access pointers. These memory address calculations often involve linear and non-linear arithmetic expressions which are typically embedded into deeply nested loops. These usually have to be calculated during program execution under tight timing constraints, thus becoming a crucial bottleneck in the overall system performance. To handle this complex arithmetic special hardware units have been used in processors, which may be either programmable or custom hardware[1, 2].

Custom hardware generation solutions have received a lot of attention and a considerable amount of work has targeted optimising their overhead [2, 3]. However, specialised hardware units and/or custom hardware generation solutions add to the design complexity and cost. On the other hand, most programmable multimedia and DSP processors [4, 5] have specialised address calculation units that provide special addressing modes, like the auto-inc(dec)crement mode, in the instruction set architecture.

In a programmable processor context, several compiler approaches targeted at instruction-level optimisations have been proposed [6, 7, 8]. SUIF is instrumented [9] with an additional pass to do aggressive code-hoisting and some induction variable optimisations at the instruction-level. However, we show in this paper that much bigger gains can be obtained by performing these aggressive optimisations at the **source code-level** prior to using standard compilers for low-level code generation. From our experiments, we deduce that the opportunities present at the high-level are to a great extent complementary and decoupled from the ones currently exploited at the instruction-level.

Previous experience with the custom ACU oriented ADOPT approach [3], has shown that when the exploration/optimisation is done at a higher-level (e.g., the index/address expression-level) much lower hardware implementation costs and synthesis time are obtained. However, our ADOPT approach up to now was solely targeted for custom hardware generation. This paper describes experiments which demonstrate that our previous work, augmented with some additional steps, can be efficiently used in the context of a software compiler approach. This will allow optimisation of high-level address generation code by applying optimising processor independent transformations to the source code. To demonstrate that the resulting high-level code is optimised for performance it is compiled on several platforms using standard compilers and the results before and after applying the transformations are compared. These results show that significant improvements are possible.

Also, the ADOPT script can be used as the back end of data transfer and storage exploration (DTSE) approaches such as *Atomium* [10] and *Acropolis* [11] which target low power background memory organisations and silent bus behaviour. The ADOPT transformations have been verified to not counteract the memory oriented transformations performed by such DTSE approaches.

Most of the transformations applied for this work have been done manually in a systematic way. Because of the formalism applied, we believe that these can be automated in a high-level code optimiser which is our current work. Some of the steps have already been automated in prototype tools such as the address expression extraction [3] and the exploration support for the algebraic transformations [12].

2. High-level address optimisation script

We refine the high-level **Ad**dress **Opt**imisation script (ADOPT) presented in [13] to also support programmable processors. The new methodology is a platform independent approach whereby the address generation code in a given behavioral description is optimised by applying processor independent transformations. Therefore, the cost model is extended to include performance, in terms of execution cycles, in addition to power and area. The resulting behavioral code can either be synthesised into hardware or compiled as software onto a target processor.

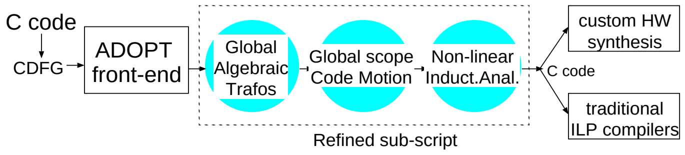

Figure 1. Processor target independent high-level address optimisation script

Our previous custom style oriented ADOPT script [13] starts off by extracting the Address Expressions (AEs) from the CDFG of the given behavioral algorithm and splitting and clustering them to maximise resource sharing. This is followed by target architecture selection, loop-invariant code hoisting (CH), induction variable analysis (IVA), and algebraic transformations (AT). The resulting AEs are then mapped onto the custom hardware processor by using specific synthesis techniques aimed at the *time-shared cluster-level* [14]. However, this ADOPT methodology does limited CH and IVA and considers these two steps isolated from each other.

In this paper, we refine the ADOPT script (see Figure 1) to propose a more detailed analysis of the interaction between the global-scope CH and IVA stages and incorporate operation cost minimisation (OCM) by exploiting algebraic transformations. The Global Algebraic Transformation Engine (GLATE) stage [12] is now performed before the other transformations. This is necessary to create more freedom at the global scope for CH, by exploring AE factorisations different from those originally present in the code. For custom processor targets, the position of the common-subexpression elimination stage at the beginning of the sub-script did not limit the search space. However, for programmable processors, exploiting beyond basic-block code invariance by CH is essential to maximally utilise the limited resources of the underlying platform. Hence, exploring factorisation alternatives increases the search space, therefore, enhancing the opportunities for other optimisations.

To properly steer GLATE to consider optimal factorisation for global scope common-subexpression elimination and CH in terms of performance cost, we extended the current model [12, 13] to consider the cheapest context-dependent cost, e.g., the cost of a constant multiplication is the cost of the corresponding add/subtract/shift network, expressed in cycles and weighted by the execution rate of the corresponding code scope.

The approach starts with a behavioral description of the application in a high-level language, e.g. *C*. The result after applying the high-level transformations (see Figure 1) is C code with optimised addressing functionality. This code can then be used either for hardware ACU generation or be compiled on a target processor using standard compilers to implement the addressing in software.

3. Address expression code extraction

The address expressions (AEs) for the indices of arrays are extracted from indexed arrays of the form $A[I_1][I_2]...[I_n]$, by linearising the multi-dimensional arrays into single-dimension linear arrays [3]. The address expression (AE) of the array indices of A can be expressed as: $AE = ((I_1*S_2 + I_2)*S_3 \ldots)*S_n + I_n$, where $I = [I_1, I_2, \ldots I_n]$ is the index vector of the array and $S = [S_1*S_2*\ldots * S_n, S_2*S_3*\ldots *S_n, \ldots S_n, 1]$ is the storage size vector of the array and S_i is the maximal size of the array in the ith-dimension. This explicit AE extraction exposes different opportunities for operation cost minimisation.

4. High-level address transformations

We now introduce the two main sub-stages of ADOPT.

4.1. Operation cost minimisation and loop-invariant code hoisting

Common Sub-expression Elimination(CSE) is an important compiler-level transformation that attempts to detect repeating sub-expressions in a piece of code, stores them in a variable and reuses the variable wherever the sub-expression occurs subsequently [15]. Since the factorisation of expressions influences CSE/CH drastically, the ADOPT script uses GLATE [12] to find optimal factorisation possibilities, remove redundant code and propagate constants at the high-level and beyond basic blocks (across loop and conditional boundaries). In contrast, traditional models used by ILP compilers typically limit the analysis capabilities for data dependencies that expand beyond the control-flow boundaries, hence limiting the optimisation capabilities to the local scope (within basic blocks).

Consider the segment of code from the *Full Search Motion Estimation* algorithm (ME) shown in Figure 2(a). This segment shows a deeply nested loop (3-levels), containing two array accesses. The address expression equations have already been extracted in the code shown in Figure 2(a) and the arrays have been linearised as explained in Section 3. The transformed code after factorised CSE using GLATE is shown in Figure 2(b). Variables, *cse*10 and *cse*100 are produced by the factorisation engine, and reused in subsequent scopes. Finally, Figure 2(c) shows how *cse*10 is moved out of the loops in which it has no dependencies with the loop variables. Similar optimisation happen on *cse*1 and *cse*2. The results in Section 5 demonstrate how these transformations lead to a significant improvement in performance in terms of execution cycles.

```
for (j_1=-8; j_1<=8; j_1++) {
 for (k_11=-4; k_11<=4-1; k_11++) {
  for (l_11=-4; l_11<=4-1; l_11++)
   Ad[((208+j_1)*257+8+k_11)*257+ 16+j_1+l_11] =
   A[(8+k_11)*257+16+j_1+l_11];
 }
 Delta += A[3096] -
        Ad[((208+j_1)*257+4)*257+ 16+j_1-4];
 for (l_12=-4+1; l_12<=4-1; l_12++)
  Delta += A[3100+l_12] -
         Ad[((208+j_1)*257+4)*257+16+j_1+l_12];
}                (a)
```

```
for (j_1=-8; j_1<=8; j_1++) {
 for (k_11=-4; k_11<=4-1; k_11++) {
  for (l_11=-4; l_11<=4-1; l_11++) {
   cse10 = (33025*j_1+6869616)*2;
   cse100 = l_11+k_11*257+1032;
   Ad[cse100+cse10] = A[cse100+1040+j_1];
  }
 }
 Delta += A[3096]-Ad[cse10];
 for (l_12=-4+1; l_12<=4-1; l_12++)
  Delta += A[3100+l_12]-Ad[l_12+4+cse10];
}                (b)
```

```
for (j_1=-8; j_1<=8; j_1++) {
 cse10 = (33025*j_1+6869616)*2;
 //cse1 to cse2 obtained after code motion
 cse1 = 1040+j_1;
 cse2 = 4+cse10;
 for (k_11=-4; k_11<=4-1; k_11++) {
  //cse101 obtained after code motion
  cse101 = k_11*257+1032;
  for (l_11=-4; l_11<=4-1; l_11++) {
   cse100 = l_11+cse101;
   Ad[cse100+cse10] = A[cse100+cse1];
  }
 }
 Delta += A[3096]-Ad[cse10];
 for (l_12=-4+1; l_12<=4-1; l_12++)
  Delta += A[3100+l_12]-Ad[l_12+cse2];
}                (c)
```

Figure 2. Full-search ME kernel: (a) original segment; (b) after operation cost minimisation; (c) after loop-invariant code hoisting

4.2. Non-Linear operator strength reduction

In this section, we demonstrate techniques to transform the polynomial expressions generated from DTSE approaches [10, 11], so as to achieve non-linear operator strength reduction, whereas standard compilers only target linear operator strength reduction [15].

The transformations attempt to reduce the number of non-constant multiplications and replace them by cheaper add/accumulate operations. Constant divisions and multiplications can be further reduced using conventional linear IVA. Multimedia applications, especially in image rendering and 3D graphics rendering algorithms, are riddled with non-linear arithmetic and the use of multiplies and divides. However, the address generation logic typically cannot afford the area and power intensive integer multipliers. After operation strength reduction, the address arithmetic can be efficiently mapped to architectural extensions such as auto-increment hardware and add-shift-load chains [6].

Consider the code segment taken from the *GSM code book lookup* shown in Figure 3(a). The polynomial expression for *cse*1 has a very expensive multiplication between two loop dependent multiplicands. However, the sequence generated by this variable can be analysed at compile time since it depends the loop variables whose bounds and stride

```
ivtmp = ipos[3];
for (i3=ivtmp; i3<40; i3+=5) {
 cse1 = i3*(i3-1)/2;
 if (i3<=i0) {
  if (i3==i0) rdm = h2[cse2];
  else rdm = rr[cse4+i3];
 } else rdm = rr[cse1+i3];
 rrv[i3/5] = s;
}               (a)
```

```
ivtmp = ipos[3];
//initialisation of ivpol
ivpol = ivtmp*(ivtmp-1)/2-5*ivtmp;
for (i3=ivtmp; i3<40; i3+=5) {
 //ivpol replaces 'i3*(i3-1)/2' in 'cse1'
 ivpol += 5*i3-15;
 if (i3<=i0) {
  if (i3==i0) rdm = h2[cse2];
  else rdm = rr[cse4+i3];
 } else rdm = rr[ivpol+i3];
 rrv[i3/5]= s;
}               (b)
```

```
ivtmp = ipos[3];
ivpol = ivtmp*(ivtmp-1)/2 - 5*ivtmp;
ivlin1 = 5*ivtmp-15; //initialise ivlin1
ivlin2 = ivtmp/5; //initialise ivlin2
for (i3=ivtmp; i3<40; i3+=5) {
 ivpol += ivlin1;
 //ivlin1 replaces '5*i3-15' in 'ivpol'
 ivlin1 += 25;
 if (i3<=i0) {
   if (i3==i0) rdm = h2[cse2];
   else rdm = rr[cse4+i3];
 } else rdm = rr[ivpol+i3];
 //ivlin2 replaces 'i3/5' in 'rrv[i3/5]'
 rrv[ivlin2++]= s;
}               (c)
```

Figure 3. GSM algorithm: (a) original segment; (b) after polynomial induction variable replacement; (c) after linear IV replacement

is known. As shown in the transformed code in Figure 3(b), the polynomial IV, *ivpol*, is an *accumulator* with a constant multiplication which generates the same sequence as *cse*1. Constant multiplications and divisions can then be removed by conventional linear induction analysis techniques as shown in Figure 3(c). For instance, the polynomial IV, *ivpol*, in Figure 3(c) is being generated by adding a linear IV, *ivlin*1, instead of the earlier $5 * i3$ multiplication. Another example is the linear IVs: *ivlin*2 replacing $i3/5$ in Figure 3(c). Similar polynomial IVA opportunities are present throughout the GSM code.

A comprehensive overview of techniques to generate polynomial IVs using a linear IV is presented by Wolfe *et al.* [16]. They demonstrate that polynomials of successively higher degree can be obtained by augmenting them with the polynomial IV of the previous degree. We have obtained huge gains by applying these transformations on source-level code as discussed in the results section.

5. Experimental framework and results on real-life demonstrators

We have performed the high-level transformations described in the previous sections on two already (manually) optimised real-life demonstrators: the full search ME algorithm and the GSM code-book lookup-search. The segments of code used from these kernels are responsible for more than the 60 % of their overall performance cost.

For our experiments, we have used general-purpose mi-

Behav.Code:	SUN UltraSparc 5 (128 MB)				HP PA-RISC 2.x (1500 MB)		
Full Search	gcc-2.95.1 -O3 -mtune=ultrasparc		cc -c -xO5 -xtarget=ultra2i		gcc-2.91.6 -O3		native cc comp.
Motion Estimation	SW Mult Emul.	Mult. Enabled	SW Mult Emul.	Mult. Enabled	SW Mult. Emul.	Mult. Enabled	Mult. Enabled
Initial	151.3	151.3	96.6	96.6	132.3	132.3	113.9
Algebraic Trafos	125.7 (20.3%)	125.7 (20.3%)	86.0 (12.3%)	86.0 (12.3%)	91.5 (45.5%)	90.9 (45.5%)	107.5 (5.8%)
+ Code Hoisting	121.7 (24.3%)	121.7 (24.3%)	82.7 (16.7%)	82.7 (16.7%)	88.2 (49.9%)	88.2 (49.9%)	105.6 (7.8%)
+ Ind. Anl.(Lin)	114.5 (32.1%)	114.5 (32.1%)	85.8 (13.5%)	85.8 (13.5%)	88.5 (49.4%)	88.5 (49.4%)	105.5 (7.8%)

Table 1. Full Search ME : number of Kcycles improvement and cumulative speed-up (%).

croprocessor based platforms (*Sun UltraSparc* and *HP PA-Risc 2.x*) instead of DSP compilation and simulation environments. The reason is state-of-the-art optimising compilers for these platforms are known to be more powerful than those available for DSPs. However, we believe that these transformations are platform independent and can easily be used for DSP processors as well, especially since several new DSP architectures are VLIWs [4, 5]. The code before and after the transformations is compiled with maximum optimisation flags enabled using GNU's *gcc 2.9x* portable compiler on both the Sun and the HP platform and Sun Solaris 2.7 *cc* compiler and HP-UX's 10.x *cc* compiler. By using two different compilers on each platform and two different benchmark algorithms, we have tried to verify that the transformations are independent of the internal idiosyncrasies of the optimising compilers and the algorithms.

The improvement in the number of cycles that the two algorithms take for execution when compiled using the portable compiler and the native compiler for both SUN and HP platforms is shown in Tables 1 and 2. The tables show the number of cycles of execution and the cumulative speed-up (measured as % overhead in number of cycles of the non-optimised version over the optimised one). Each table compares the number of cycles when the code is compiled using software emulation for the integer multiplies by adds and shifts, i.e., by disabling the use of the multiplier and when the code is compiled so as to use the hardware integer multiply instruction. This is done since it is desirable to reduce or eliminate multipliers from the ACU so as to reduce area and power consumption. There is no flag in HP's native cc compiler to disable the generation of multiply instructions, so the HP cc results in Tables 1 and 2 only show the results with the multiplier enabled.

For the Motion Estimation kernel, Table 1 compares the results when the applications are compiled after AT and operation cost minimisation OCM (second row), and after applying the loop-invariant CH (third row). The table shows that most speed-up is due to the AT exploration phase on both architectures. After the application of OCM and loop-invariance transformations on the ME algorithm, a significant (upto 50%) speed-up is achieved.

For the GSM kernel, Table 2 compares the results for when the algorithm is compiled after exploring AT at the global scope (second row), and then, followed by global-scope CH (third row). In this driver, CH was applied not only across loop boundaries but also across data-dependent conditional boundaries, where subexpressions which are defined inside certain conditional basic blocks have been moved unconditionally up-front in the code and reused in all the remaining occurrences. To decide where to move the subexpressions, we have respected the procedural execution of the basic blocks. By profiling, we have decided when the gain of unconditionally computing the subexpressions is bigger than the cost of conditionally computing them. Table 2 also shows the cumulative numbers after applying the IVA, both polynomial and linear. Its evident from the table that for this kernel most of the speed-up is due to IVA.

Note that when performing linear IVA at the source-level, the number of cycles for the ME algorithm is at times increased marginally, (see last row in Table 1) perhaps because of register spillage due to the added registers required to store the induction variables. Conventional compilers are capable of doing simple linear IVA by converting multiplies into adds and shifts and they can control the register spillage overhead. However, linear IVA can be effectively exploited with DSPs due to their zero-cycle overhead auto-increment addressing modes in the ACU [6].

We have also found that a similar effect happens in the GSM code. The GSM code is dominated by non-linear arithmetic, with 2.8K integer multiplications and 1.16K constant divisions being initially executed at run time. After the ADOPT transformations, the new code contains just 154 multiplications and 154 constant divisions (needed for the initialisation of the different induction variables). Table 3 shows the reduction in number of calls to the SW-emulated multiply and divide functions at each stage of the transformation script and the cumulative savings as percentage reduction. Note that although the effective number of calls to constant multiplications is not reduced when performing linear IVA, a significant reduction (up to 86%) in calls to constant divisions is observed.

This experiment demonstrates how applying linear induction analysis may be closely interlinked to the steps and passes being performed by the compiler, especially when reducing the strength of the constant multiplications. However, the rest of the steps, namely global-scope CSE/CH, non-linear IVA, and possibly even linear IVA for division replacement, remain for the most part complementary and well decoupled.

In terms of absolute number of cycles of code execution, the native compiler, which is typically highly optimised for the target processor, performed better than the portable

Behav.Code:	SUN UltraSparc 5 (128 MB)				HP PA-RISC 2.x (1500 MB)		
GSM Codebook	gcc-2.95.1 -O3 -mtune=ultrasparc		cc -c -xO5 -xtarget=ultra2i		portable gcc-2.91.6 -O3		native cc comp.
Look-up Search	SW Mult. Emul.	Mult. Enabled	SW Mult. Emul.	Mult. Enabled	SW Mult. Emul.	Mult. Enabled	Mult. Enabled
Initial	15.94	9.79	17.91	10.01	20.88	10.87	9.26
Algebraic Trafos	15.93 (0.0%)	9.81 (-0.2%)	17.91 (-0.0%)	10.01 (-0.0%)	21.01 (-0.5%)	10.84 (0.3%)	9.45 (-1.9%)
+ Code Hoisting	14.93 (6.8%)	9.08 (7.9%)	16.87 (6.2%)	9.45 (5.9%)	18.73 (11.5%)	9.78 (11.1%)	8.55 (8.3%)
+ Ind.Anl.(Poly)	15.01 (6.2%)	8.68 (12.9%)	15.09 (18.6%)	8.28 (20.9%)	16.75 (24.6%)	8.40 (29.4%)	7.41 (25%)
+ Ind.Anl.(Lin)	12.93 (23.3%)	8.45 (15.9%)	13.60 (31.7%)	7.45 (34.4%)	15.64 (33.5%)	7.21 (50.7%)	7.01 (32.1%)

Table 2. GSM codebook: number of Mcycles improvement and cumulative speed-up (%).

Behav.Code:	SUN ULTRASparc 5 (256 MB)				HP PA-RISC 2.x (1500 MB)	
GSM Codebook	portable gcc 2.95.1		native-cc		portable gcc 2.91.6	
Look-up Search	Multiplies	Divides	Multiplies	Divides	Multiplies	Divides
Alg.Trafos	1554	188	1665	6.5	1634	188
+ Code Hoisting	1359 (12.5%)	188 (0.0%)	1548 (7.6%)	6.5 (0.0%)	1417 (13.2%)	188 (0.0%)
+ Ind.Anl. (Polynoml.)	1146 (26.5%)	188 (0.0%)	1321 (26.0%)	6.5 (0.0%)	1146 (29.8%)	188 (0.0%)
+ Ind.Anl. (Linear)	1146 (26.5%)	25 (86.7%)	1139 (46.2%)	6.5 (0.0%)	1146 (29.8%)	25 (86.7%)

Table 3. GSM: reduction in num. of Kcalls & cumulative saving(%) in SW integer multiplies & divides

one for both drivers. However, after applying the ADOPT script, we note that both the compilers perform equally well in terms of the absolute number of cycles, irrespective of the platform. Also, when we migrated from gcc 2.8 to gcc 2.9, we found that the absolute number of cycles improved, but contrary to expectation, the speed-ups after our transformations also *improved*. Clearly, an aggressive, processor independent, source-to-source pre-compiler can greatly help improve compilation results by exploring optimisations at the global scope and remove the effects of syntactic variance and coding styles. The large search space at the instruction-level makes it more difficult to find global optimisation opportunities, hence, decreasing exploration productivity.

6. Conclusions

In this paper, we propose a script for advanced high-level address optimisation consisting of a well defined sequence of source-level transformations. Through our experimental results, we verify that we achieve significant speed-ups, up to 50%, in terms of number of cycles, and savings, up to 86%, in the number of calls to multiply/divide integer units. Moreover, we demonstrate that these transformations can be applied at the source-level, followed by the use of standard compilers. We identify among the proposed transformations those which counteract with the compiler and those which are complementary to the compiler steps. Finally, we have also demonstrated that the results obtained after applying the optimisations are near optimal, independent of the compiler and platform selected. This clearly demonstrates the need for a highly aggressive and portable, processor independent, source-to-source pre-compiler targeted at address computation code optimisation.

Although we have verified the effectiveness of our transformations only with RISC processor architectures whereas the final target architectures for such algorithms are DSPs, these source-level transformations are to the most extent, platform-independent and will easily extend to DSPs, especially contemporary VLIW-like DSP processors [4, 5]. However, more experimentation is required to determine the interactions that architectural features of RISCs such as caches and large register sets have with our transformations.

7. Acknowledgements

The authors would like to acknowledge the support of the National Science Foundation under grant CCR-9806898 and DARPA under grant DARPA/ITO DABT63-98-C-004. This research has been supported by the SMT and IT-IMAGE projects of the Flemish Community - IWT.

References

[1] K.Kitagaki, T.Oto, T.Demura, Y.Araki, T.Takada, *A new address generation unit architecture for video signal processing*, Visual Communications and Image Processing,1991.
[2] P.Lippens, J.Van Meerbergen, *et al. Phideo: A silicon compiler for high speed algorithms*, Europ. Conf. for Design Automation, 1991.
[3] M.Miranda, F.Catthoor, M.Janssen, H.De Man, *ADOPT: Efficient hardware address generation in distributed memory architectures*, Intl. Symp. on System Synthesis, 1996.
[4] Texas Instruments, *TI TMS320C6x User's Guide*.
[5] Philips Semiconductor, *Trimedia TM1000 Programmable media processor databook*.
[6] R.Leupers, P.Marwedel, *Algorithms for Address Assignment in DSP Code Generation*, Intl. Conf. on Computer-Aided Design, 1996
[7] A.Sudarsanam, S.Liao, S.Devadas, *Analysis and evaluation of address arithmetic capabilities in custom DSP architectures*, Design Automation Conference, 1997.
[8] B.Wess, *Minimisation of data address computation overhead in DSP programs*, Design Automation for Embedded Systems, no. 4, 1999.
[9] S.M. Pujare, C.G. Lee, P. Chow, *Machine-Independent Compiler Optimizations for the UofT DSP Architecture*, Intl. Conf. on Signal Proc. Apps. and Tech., 1995
[10] 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*, Kluwer Academic Publishers, 1998
[11] K. Danckaert, F. Catthoor, H. De Man, *System-level memory management for weakly parallel image processing*, EuroPar Conf., 1996.
[12] M.Janssen, F.Catthoor, H.De Man, *A specification invariant technique for operation cost minimisation in flow-graphs*, Intl. Symp. on High-level Synthesis,1994
[13] M.Miranda, F.Catthoor, M. Janssen, H.De Man, *High-Level Address Optimisation and Synthesis Techniques for Data-Transfer Intensive Applications*, IEEE Trans. on VLSI Systems, no.4, vol.6, Dec. 1998.
[14] S.Note, W.Geurts, F.Catthoor, H.De Man, *Cathedral-III: Architecture driven high-level synthesis for high throughput DSP applications*, Design Automation Conference, 1991.
[15] A.Aho, R.Sethi, J.Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.
[16] M.Gerleck, E.Stoltz, M.Wolfe, *Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form*, ACM Trans. Progrm. Languages and Systems:17, Jan. 1995.