High-level Control Flow Transformations for Performance Improvement of Address-Dominated Multimedia Applications

Heiko Falk
Computer Science 12
University of Dortmund
Dortmund, D 44221
Heiko.Falk@udo.edu

Cédric Ghez
DESICS
IMEC Lab.
Leuven, B 3001
ghez@imec.be

Miguel Miranda
DESICS
IMEC Lab.
Leuven, B 3001
miranda@imec.be

Rainer Leupers
ISS
RWTH Aachen University
Aachen, D 52056
leupers@iss.rwth-aachen.de

Abstract— This paper describes a set of novel high-level control flow transformations for performance improvement of typical address-dominated multimedia applications. We show that these transformations applied at the source code level can have a very large impact on execution time at the cost of limited overhead in code size for a broad range of instruction set processor families (i. e. CISC, RISC, DSP, VLIW, ...). For a profound evaluation, all transformations are applied to the C-codes of two real-life applications selected from the video and image processing domains. A detailed analysis of the effect of the transformations is done by compiling and executing the transformed programs on seven different programmable processors. The measured runtimes indicate quite significant improvements in all processor families when comparing the performance of the transformed codes to their initial version even when these are compiled using their native optimizing compilers with their most aggressive optimization features enabled. The average gains in execution time range from 40.2% and 87.7% depending on the driver, with an average overhead in code size between 21.1% and 100.9%.

I. Introduction

Typical multimedia applications often involve a large amount of transfers to the memories of an embedded system. This results in huge penalties with respect to runtime due to slow memories and bus systems as well as power consumption due to the domination of total system power by memory related power consumption [20].

In general, source code transformations are applied to remove Data Transfer and Storage overhead (*DTS*). Several examples of such methodologies have been described in related literature, e. g. [16, 4] amongst others.

These methodologies typically achieve a significant reduction of memory footprint and bus transfers but lead to a more complex addressing functionality with respect to addressing arithmetic and control flow modification. Therefore, that overhead must be removed to ensure a good total system performance, especially for programmable processors. For that purpose, several address optimization methodologies and techniques have been proposed in literature, mostly located at the assembly code abstraction level [11, 7, 17, 18], but also at the source code level [12, 14]. However, they mostly focus on the optimization of the arithmetic related issues but do not address control flow execution issues which is the contribution of this paper.

The control flow overhead introduced by DTS approaches originates e. g., when reducing the required memory footprint but also when exploiting the data reuse opportunities present in the code. However, we have observed that also real-life multimedia code can exhibit such behaviour (as it is the case for one of the two drivers used in this work) even when not processed by DTS related transformations.

With the help of if-statements within global loops, the memory footprint is reduced by loop merging transformations implemented for array references in local loops with different but still overlapping iteration domains. With the help of extra copy signals and extra (local) loops, most memory accesses presenting data reuse opportunities can be localized at the lower layers of the memory hierarchy, hence reducing system bus load. However, this results in overhead in control flow which leads to overall runtime degradation resulting mostly from stalls in the execution of the instruction pipeline but also from increased miss rates of the instruction cache.

Some overhead in arithmetic is also present due to the condition expressions of the extra if-statements and the adapted indexing, but also due to overhead originally present in the code. However, that one can be largely eliminated by first applying an (address) arithmetic optimization stage, e. g. the ones developed within the ADOPT project [14]. These optimizations are implemented at the source code level and in a processor independent way.

These address optimizing source code transformations (in fact all loop iterator dependent arithmetic including conditions over loop iterators) have been proven to be effective for multimedia applications independently of the processor architecture for which the applications have been compiled and benchmarked.

In this paper, we demonstrate that after the optimization of the address and condition related arithmetic, huge improvements in runtimes can still be achieved. The largest potential for optimization lies in the simplification of the generated control flow structures which has not been considered so far. A new systematic technique for source code

transformation is presented that allows to automatically reduce the amount of branch evaluation in the inner loop of a nest by the use of a novel *loop nest split* technique.

Further optimization of the addressing can be performed (even after the optimization of the arithmetic functionality) by replacing small *ring buffers* by register variables (namely, *ring buffer replacement*) such that no addressing code is required for these at all. This can be done very efficiently by avoiding unnecessary data moves between the functional units and the register files by applying novel loop unrolling techniques steered by control flow issues.

An important advantage of using source code transformations compared to compiler optimizations is that an implementation of such techniques can be done independently of architectural features of different processors and independently of intermediate representations and data structures used within the compiler. Native compilers especially designed and tuned for a processor architecture are still needed as back-end for the generation of highly optimized assembly code. Hence, the source code optimization approach is highly portable.

The code transformations proposed in this paper have been applied to two multimedia applications written in ANSI C in a systematic way, partially with the help of already implemented tools. The optimized code versions of the benchmarks have been compiled for seven different programmable processors and their runtimes have been measured on the real processor hardware.

The remainder of this paper is organized as follows: A survey of work related to address code generation and source code transformations is given in section II. Section III presents the optimizations applied to the benchmarks. An overview of the applications serving as benchmarks and the processors considered is given in section IV. Section V contains a detailed description of the measured results. Section VI summarizes and concludes this paper.

II. Related Work

Some particular DTS source code transformation methodologies are DTSE [4] and ADOPT [14] which are already mentioned previously. Besides these methodologies, many approaches have been presented for the automated generation of high-quality address code. In [11, 7, 17, 18], heuristics for the computation of memory layouts for program variables and for the address assignment problem are described. These approaches are situated at the level of address registers and post-increment/decrement operations provided by the instruction set of digital signal processors (*DSP*s). For this reason, they are implemented as compiler built-in optimizations.

A loop transformation framework is suggested by Kelly and Pugh [9]. This framework helps in representing loop transformations in a unified formal way, but does not target the problem of selection and ordering of transformations.

Liem et al. [12] have proposed a framework for the analysis of array references and for the efficient use of address generation units (*AGU*s). In their work, the analysis of memory accesses is done by profiling at the level of address registers, whereas the optimizations performed afterwards are expressed at the source code level.

Zory and Coelho [21] have investigated source code transformations in the context of factorization of algebraic expressions and efficient mapping of such expressions to special hardware capabilities of processors.

Manniesing et al. [13] present a way for transforming embedded applications especially for DSPs with instruction set extensions according to the Single Instruction Multiple Data (*SIMD*) paradigm. Source code is rewritten with the help of a transformation tool for ANSI-C [3] which is based on pattern recognition strategies.

In the domain of intermediate representations suitable for source code transformations, the SUIF [19] system is well-known. It provides a set of data structures for the representation and optimization of programs together with a parser supporting C and Fortran as well as a back-end printing out the contents of the data structures as C code.

The LANCE system proposed by Leupers [10] uses an executable intermediate representation. ANSI-C programs are represented as statements of 3-address code that are stored in the form of ANSI-C instructions. This way, optimizations based on the LANCE representation can be seen as low-level source code transformations.

III. Transformations

In this section, the source code transformations to be applied after the DTS and address optimization stages are illustrated. They focus on the optimization of the control flow and address related issues.

The transformations presented in this section are currently partly automated. These tools are based on the SUIF [19] intermediate representation. Due to the prototype nature of our tools, some manual intervention is still required. During this manual process, care has been taken to ensure that always a formal and systematic approach has been used when transforming the benchmarks. In the future, this will allow us to fully automate the entire optimization process by implementing a complete tool chain for source code transformations.

A. Loop Nest Split for Branch Evaluation Optimization

Very often, manifest if-statements are present inside loop nests in order to detect particular ranges in the complete loop iteration space. This introduces a large overhead within the loops, because on the one hand, many computations have to be done for the evaluation of the conditions. On the other hand, a very irregular control flow is generated leading to a sub-optimal performance of the processor pipeline. A typical situation[1] is depicted in fig. 1. As can be seen, two if-statements with eight conditions on the induction variables have to be evaluated in the innermost j-loop. This leads to the execution of more than 90 million if-statements throughout the entire loop nest. Since the code is already written in an optimized manner (i. e.

[1] Taken from the Motion Estimation benchmark presented in section IV.

common subexpressions are eliminated and loop invariant code is moved out of the inner loops), it is obvious that the if-statements themselves represent the main bottleneck in this code.

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {
  x1=4*x;
  for (y=0; y<49; y++) {
   y1=4*y;
   for (k=0; k<9; k++) {
    x2=x1+k-4;
    for (l=0; l<9; l++) {
     y2=y1+l-4;
     for (i=0; i<4; i++) {
      x3=x1+i; x4=x2+i;
      for (j=0; j<4; j++) {
       y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3 || y3<0 || 48<y3)
        then_block_1; else else_block_1;
       if (x4<0 || 35<x4 || y4<0 || 48<y4)
        then_block_2; else else_block_2; }}}}}}
```

Fig. 1. Loop Nest before Splitting

The code fragment above seems to be suitable for the application of conventional loop unswitching as proposed by Bacon et al. [1], which is able to move if-statements out of frequently executed loops (see fig. 2).

```
for (i=1; i<n; i++) {
  a[i]=a[i]+c;
  if (x<7)
    b[i]=a[i]*c[i];
  else
    b[i]=a[i-1]*b[i-1]; }
```

```
if (x<7)
  for (i=1; i<n; i++) {
    a[i]=a[i]+c;
    b[i]=a[i]*c[i]; }
else
  for (i=1; i<n; i++) {
    a[i]=a[i]+c;
    b[i]=a[i-1]*b[i-1]; }
```

Fig. 2. Conventional Loop Unswitching

Unfortunately, this well-known technique requires the conditions of the if-statements to be loop-invariant which is not the case here. To our knowledge, no optimization technique exists which is able to reduce the control flow overhead in situations like the one shown in fig. 1. In contrast, the loop nest split technique presented in this section complements the conventional loop unswitching, since we explicitly focus on loop-variant if-statements and are able to optimize them.

We have found that a large potential for optimization can be exploited when carefully analyzing the conditions of the if-statements and the induction variables. We have developed and implemented optimization strategies based on genetic algorithms that are capable of identifying ranges of values of the induction variables where all conditions of the if-statements are proven to be true. For this purpose, a polyhedral representation of conditions in loop nests is used. In a first step, a genetic algorithm optimizes each single condition in a loop nest isolated without considering the structure of entire if-statements in the loop nest and generates optimized polyhedra for all conditions. Hereafter, these optimized polyhedra are combined to one single polyhedron representing the whole loop nest with all if-statements and conditions and thereby modelling a global search space. To this global search space, a second genetic algorithm is applied which determines the globally best solution for an entire loop nest [5, 6]. For the example given above, our technique has lead to the result that both if-statements in the j-loop are true for $x \geq 10$ or $y \geq 14$.

The goal of our loop nest split technique is to generate a supplementary loop nest covering exactly these ranges of the induction variables determined by the analysis. Therefore, the y-loop is split by duplicating the loop nest located inside the y-loop, i.e. all loops from the k- to the j-loop. The duplicated loop nest is guarded by a new if-statement ensuring that this copy is only executed for the values of x and y determined by the analysis. Since for these values, it is known that the conditions of the if-statements are true, the if-statements can be removed from the duplicated loop nest resulting in a piece of code with linear control flow. The transformation of the code from fig. 1 is illustrated in fig. 3. After our optimization, only a total number of around 7 millions of if-statements is executed.

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {
  x1=4*x;
  for (y=0; y<49; y++)
   if (x>=10 || y>=14) {      /* Splitting-If */
    for (y_new=y; y_new<49; y_new++)
     for (k=0; k<9; k++)
      for (l=0; l<9; l++)
       for (i=0; i<4; i++)
        for (j=0; j<4; j++) {
         then_block_1; then_block_2; }
    y = y_new;
   } else {
    y1+4*y;
    for (k=0; k<9; k++) {
     x2=x1+k-4;
     for (l=0; l<9; l++) {
      y2=y1+l-4;
      for (i=0; i<4; i++) {
       x3=x1+i; x4=x2+i;
       for (j=0; j<4; j++) {
        y3+y1+j; y4+y2+j;
        if (0 || 35<x3 || 0 || 48<y3)
         then_block_1; else else_block_1;
        if (x4<0 || 35<x4 || y4<0 || 48<y4)
         then_block_2; else else_block_2; }}}}}}
```

Fig. 3. Loop Nest after Splitting

B. Ring Buffer Replacement

As already mentioned in section I, circular copy signals can be introduced for data-reuse optimization, therefore many array accesses are performed using modulo operators. This very often leads to the effect that many small arrays are involved where the dividend depends on the induction variable of a loop, as shown on the left side of fig. 4. In these cases, arrays serve as ring buffers that are accessed in a circular way within loops.

It is the goal of the ring buffer replacement techniques presented in this section to scalarize these arrays, i. e. to replace them by a set of scalar variables. Every access to an element of the original array is replaced by the corresponding scalar variable as principally shown on the right side of fig. 4.

```
int ip[3];                          int ip0,ip1,ip2;
for(x=..;..;x++){                   for(x=..;..;x++){
 ip[x % 3] = code;                   ...
 a = ip[(x-2) % 3]*68;               ip_x = code;
 a += ip[(x-1) % 3]*99;              a = ip_y*68;
 a += ip[x % 3]*68;}                 a += ip_z*99;
                                     a += ip_x*68;}
```

Fig. 4. Ring Buffer Replacement

Conventional optimizations for array scalarization already exist (see e. g. [1, 15]), but they are unable to handle the circular access to the ring buffer. In contrast, our techniques consisting of a scalarization and an unrolling step are able to replace the ring buffers and model the circular access to its elements. The main advantages of the techniques presented here are firstly that complex addressing arithmetic is removed from the code. Since the memory addresses of scalar variables are determined at compile time, no costly address calculations have to be performed by a processor at runtime. This holds for all kinds of processors, even for DSPs with complex address generation units. The second advantage of the ring buffer replacement is that the newly inserted scalar variables are automatically fed into the register allocation phase of a compiler. Thus, the compiler can decide to keep the scalars in processor registers leading to faster and less energy-consuming accesses to the ring buffer. In the following two subsections, the scalarization and unrolling steps mentioned above are illustrated.

B.1. Ring Buffer Scalarization

In a first phase, the elements of a ring buffer are scalarized so that explicit addressing expressions like e. g. `(x-2) % 3` (see left side of fig. 5) and the implicit addition with the base address of the array are removed from the code.

Special care has to be taken to model the circular access to the elements of the ring buffer correctly. This is done by inserting some copy instructions at the beginning of a loop as shown on the right side of fig. 5. With the help of these copy instructions, the contents of the scalars is shifted

```
int ip[3];                          int ip0,ip1,ip2;
for(x=..;..;x++){                   for(x=..;..;x++){
 ip[x % 3] = code;                   ip2 = ip1;
 a = ip[(x-2) % 3]*68;               ip1 = ip0;
 a += ip[(x-1) % 3]*99;              ip0 = code;
 a += ip[x % 3]*68;}                 a = ip2*68;
                                     a += ip1*99;
                                     a += ip0*68;}
```

Fig. 5. Ring Buffer Scalarization

by one position in every loop iteration and consequently models the cyclic behavior of ring buffers.

Obviously, this scalarization step leads to a slight overhead, because additional instructions and data transfers are inserted. This overhead can not be removed using standard copy propagation techniques, but we are able to eliminate it with the help of the loop unrolling technique proposed in the following section.

B.2. Loop Unrolling for Ring Buffers

Conventional loop unrolling (see e. g. Muchnick [15]) is a very common technique for reducing loop overhead. The determination of the so-called *unrolling factor* is normally done by an analysis of size and number of iterations of a loop so that an explosion of code size is avoided. In contrast, loop unrolling in the context of ring buffer replacement is steered by the number of copy instructions inserted by the scalarization step described in section B.1. The left side of fig. 6 shows 3 variables created by the ring buffer scalarization. The contents of variables `ip1` and `ip0` is copied to `ip2` and `ip1`, and a new value is assigned to `ip0`. These two copy instructions can be removed completely by unrolling the loop with factor 3 and adapting definitions and uses of the `ip`x variables. The resulting code is shown on the right side of fig. 6.

```
for(x=..;..;x++){                   for(x=..;..;x+=3){
 ip2 = ip1;                          ip2 = code;
 ip1 = ip0;                          a = ip1*68;
 ip0 = code;                         a += ip0*99;
 a = ip2*68;                         a += ip2*68;
 a += ip1*99;                        ip1 = code;
 a += ip0*68;}                       a = ip0*68;
                                     a += ip2*99;
                                     a += ip1*68;
                                     ip0 = code;
                                     a = ip2*68;
                                     a += ip1*99;
                                     a += ip0*68;}
```

Fig. 6. Loop Unrolling for Ring Buffers

IV. Experimental Setup

For evaluating the effects of the transformations, two multimedia applications from the image processing domain have been used as test drivers. The first one is a cavity detection algorithm (*CAVITY*, [2]) mainly used in the medical field for detecting tumour cavities in computer tomography pictures. The second benchmark is an MPEG 4 full search motion estimation kernel (*ME*, [8]) used in video en/decoding algorithms. For clearly demonstrating that the techniques presented in this paper are not necessarily depending on previous DTS-like optimizations, only the CAVITY benchmark has passed the DTSE transformations [4]. In contrast, both drivers are address optimized [14] so that the memory and addressing optimized versions serve as starting point for the performance oriented transformations described in this paper.

It is obvious that strong interdependencies exist between the ring buffer scalarization and our loop unrolling technique. Because of these relations, these two transformations should be executed sequentially without any other transformation in between. Since clearly no interdependencies exist between the loop nest split technique of section A and the ring buffer scalarization / loop unrolling, the former transformation can principally be executed before or after the latter ones. Still, we have decided to do the loop nest splitting before any other transformation. This is only due to the fact that the analysis routines for the loop nest splitting are easier to implement when unrolled loops don't need to be considered.

All code versions generated during the entire transformation process were compiled with the highest degree of optimization for an Intel Pentium 3, Sun UltraSparc-III, HP-9000, TriMedia TM-1000, Texas Instruments TMS320C6x and an ARM7TDMI core both in 32-bit arm-mode and 16-bit thumb-mode. For the Pentium, the Visual-C++ compiler was used, for all other platforms we used the highly optimizing compilers provided by the processor manufacturers (e.g. Sun Workshop, TI Code Generation Tools). Runtimes were measured by executing and profiling the compiled programs on existing hardware using either available workstations or evaluation boards. The usage of real hardware is advantageous because all measurements can be done within a very small time frame without depending on simulation tools which are known to be imprecise and slow. Furthermore, the available caches of all processors (except TI C6x and ARM7TDMI) are considered automatically when measuring the on-board execution times.

V. Results

The tables I and II show the runtimes of the benchmarks on the different processors. Runtimes are measured for the memory / address optimized version of a benchmark (designated as "Reference Code" in the tables) and after each of the transformations presented in section III. The last row in these tables shows the total improvements measured when applying the transformations presented in this paper. For this purpose, the fastest code version generated during transformation (highlighted in **bold** font) is compared with the reference code. Empty rows denote missing code versions where a transformation could not be applied since that benchmark did not expose an opportunity. For all processors, the runtimes in the following tables are given in CPU seconds.

As table I shows, the loop nest split and ring buffer scalarization steps are the ones leading to the highest gains for the CAVITY benchmark on all platforms (between 21.8% and 66.3% altogether). The subsequent transformations lead to further improvements, the best runtimes vary between 32.5% (SUN) and 74% (TriMedia) of the runtimes before transformation. On the average over all processors, the runtimes have been improved by 40.2%.

In the case of the ME benchmark, a significant speed-up is achieved by performing loop nest splitting (see table II). Hereafter, the runtimes vary between 9.2% and 18.3% of the referenced code version. This surprising result is due to the fact that our analysis shows that all if-statements in the heart of a 7-fold loop nest evaluate to *true* for more than 92% of the iteration space. The fastest runtimes reach from 8.6% (ARM) up to 17.4% (C6x). The total average improvement of the transformations presented in this paper lies around 87.8%.

Tables III and IV show the evolution of code sizes during the transformation of both benchmarks. Code sizes are represented by the number of assembly instructions generated by the compilers for the different source code versions. As can be seen from table III, the code sizes of the CAVITY benchmark increase between 75.7% (HP) and 127.3% (Pentium) after the application of loop nest splitting, ring buffer scalarization and loop unrolling. On the average, the transformations presented in this paper lead to an increase by 100.9%. This increase by a factor of two is very moderate, since one would expect an increase by a factor of six due to loop nest splitting (factor 2) and loop unrolling (factor 3). In the case of the ME benchmark, the code size increases are even more moderate (see table IV). Here, the application of loop nest splitting and unrolling makes the code grow between only 5.7% (ARM thumb) and 63% (HP). After the whole transformation process, an average code size increase of 21.1% is observed.

These numbers show that modern optimizing compilers are not capable of achieving improvements comparable to the gains obtained by the source code transformations presented here. This is due to the fact that compilers concentrate on standard optimizations. In contrast, the techniques described here go far beyond this scope and are application domain specific in such a way, that particular properties of the code generated by memory and address optimization phases are recognized and exploited successfully.[2]

[2] Note however, that our techniques are general enough so that the execution of these DTS stages is not a necessary precondition. For example, the application of the loop nest splitting on the ME benchmark is possible without DTSE (see also [5, 6]).

TABLE I
RUNTIMES OF CAVITY

[CPU sec.]	Pentium	Sun	HP	TriMedia	TI C6x	ARM7 thumb	ARM7 arm
Reference Code	0.97	0.83	0.6	5.17	10.09	4.58	5.34
Loop nest split	0.75	0.37	0.36	**3.83**	8.48	3.73	4.63
Ring buffer scal.	0.66	0.28	0.41	4.04	6.27	3.15	4.09
Loop unrolling	**0.63**	**0.27**	**0.33**	3.91	**5.73**	**2.94**	**3.79**
Improvement (%)	35.05	67.47	45	25.92	43.21	35.81	29.03

TABLE II
RUNTIMES OF MOTION ESTIMATION

[CPU sec.]	Pentium	Sun	HP	TriMedia	TI C6x	ARM7 thumb	ARM7 arm
Reference Code	1.95	1.29	1.9	20.45	19.97	351.17	420.79
Loop nest split	0.26	**0.17**	0.27	2.14	3.66	33.57	38.93
Ring buffer scal.							
Loop unrolling	**0.25**	0.19	**0.27**	**2.08**	**3.47**	**32.18**	**36.16**
Improvement (%)	87.18	86.82	85.79	89.83	82.61	90.84	91.41

TABLE III
CODE SIZES OF CAVITY

[#Asm instrs.]	Pentium	Sun	HP	TriMedia	TI C6x	ARM7 thumb	ARM7 arm
Reference Code	513	766	511	1790	612	608	473
Loop nest split	885	1047	670	2535	1103	958	751
Ring buffer scal.	719	879	592	2480	708	757	600
Loop unrolling	1166	1399	898	4035	1163	1224	964

TABLE IV
CODE SIZES OF MOTION ESTIMATION

[#Asm instrs.]	Pentium	Sun	HP	TriMedia	TI C6x	ARM7 thumb	ARM7 arm
Reference Code	324	372	430	1570	416	730	416
Loop nest split	358	423	512	1680	497	774	450
Ring buffer scal.							
Loop unrolling	352	425	701	2095	478	772	447

VI. CONCLUSIONS

In this article, we present a set of several high-level control flow transformations for runtime improvement of address-dominated multimedia applications. The optimizations proposed here have been applied to two different image processing applications. The generated source codes have been compiled and executed on a set of seven different processors. The average gains in execution time range from 40.2% up to 87.8%, with an average overhead in code size between 21.1% and 100.9%.

The fact that we achieve significant improvements over seven different processors and optimizing compilers clearly shows that our approach of compiler-independent source code transformations is highly performant. A higher potential for optimization can be exploited when taking into account the underlying processor organization constraints during a subsequent processor-dependent transformation phase. This is part of our future work.

REFERENCES

[1] D. F. Bacon, S. L. Graham and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), Dec. 1994.

[2] M. Bister, Y. Taeymans and J. Cornelis. Automatic Segmentation of Cardiac MR Images. *IEEE Journal on Computers in Cardiology*, 1989.

[3] M. Boekhold, I. Karkowski and H. Corporaal. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. *High Performance Computing and Networking conference, Amsterdam*, Apr. 1999.

[4] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg et al. *Data access and storage management for embedded programmable processors.* Kluwer Acad. Publ., Boston, 2002.

[5] H. Falk. Control Flow Optimization by Loop Nest Splitting at the Source Code Level. Research Report 773, University of Dortmund, Oct. 2002.

[6] H. Falk and P. Marwedel. Control Flow driven Splitting of Loop Nests at the Source Code Level. In *Design, Automation and Test in Europe (DATE)*, Munich, Mar. 2003.

[7] C. Gebotys. DSP Address Optimization Using A Minimun Cost Circulation Technique. In *International Conference on Computer-Aided Design (ICCAD)*, San Jose, Nov. 1997.

[8] S. Gupta, M. Miranda, F. Catthoor and R. Gupta. Analysis of High-level Address Code Transformations for Programmable Processors. In *Design, Automation and Test in Europe (DATE)*, Paris, Mar. 2000.

[9] W. Kelly and W. Pugh. A Unifying Framework for Iteration Reordering Transformations. In *First International Conference on Algorithms and Architectures for Parallel Processing*, pages 153 – 162, Brisbane, Apr. 1995. IEEE.

[10] R. Leupers. *Code Optimization Techniqes for Embedded Processors - Methods, Algorithms and Tools*. Kluwer, Boston, 2000.

[11] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *International Conference on Computer-Aided Design (ICCAD)*, San Jose, Nov. 1996.

[12] C. Liem, P. Paulin and A. Jerraya. Compilation Methods for the Address Calculation Units of Embedded Processor Systems. *Journal of Design Automation for Embedded Systems*, 2(1), Jan. 1997.

[13] R. Manniesing, I. Karkowski and H. Corporaal. Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition. In *Lecture Notes in Computer Science*, Berlin, 2000. Springer.

[14] M. Miranda, F. Catthoor, M. Janssen and H. de Man. High-level Address Optimisation and Synthesis Techniques for Data-Transfer Intensive Applications. *IEEE Transactions on VLSI Systems*, 6(4):677 – 686, Dec. 1998.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[16] P. R. Panda, N. Dutt and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Acad. Publ., Boston, 1999.

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

[18] S. Udayanarayanan and C. Chakrabarti. Address Code Generation for Digital Signal Procesors. In *Design Automation Conference (DAC)*, Las Vegas, June 2001.

[19] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson et al. An Overview of the SUIF Compiler System. *http://suif.stanford.edu/suif/suif1*, 1995.

[20] S. Wuytack, F. Catthoor, L. Nachtergaele and H. De Man. Power Exploration for Data Dominated Video Applications. In *International Symposium on Low Power Electronics and Design (ISLPED)*, Monterey, Aug. 1996.

[21] J. Zory and F. Coelho. Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Code. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Paris, Oct. 1998.