Systematic speed-power data-layout exploration for cache controlled embedded multimedia applications

M. Miranda C. Ghez C. Kulkarni F. Catthoor* D. Verkest

IMEC Lab., Kapeldreef 75, Leuven, Belgium
*Also Professor at the Katholieke Universiteit Leuven, Belgium

{miranda,ghez,catthoor,verkest}@imec.be

ABSTRACT

The ever increasing gap between processor and memory speeds has motivated the design of embedded systems with deeper cache hierarchies. To avoid excessive miss rates, instead of using bigger cache memories and more complex cache controllers, program transformations have been proposed to reduce the amount of capacity and conflict misses. This is done by complicating the memory index arithmetic code which results in performance degradation. However, when these are complemented by high-level address code transformations, the overhead introduced can be largely eliminated a compile time. In this paper, the clear benefits of the combined approach is illustrated on two real-life applications of industrial relevance, using popular programmable processor architectures and showing important gains in energy (up to a factor 3 less) with a relatively small penalty in execution time (up to a 30%). The results of this paper leads to a systematic trade-off (supported by tools) between memory power and CPU cycles which has up to now not been feasible for the targeted systems.

Categories and Subject Descriptors

H.4.m [**Xxxxxxx**]: Yyyyyyyyy; D.2 [**Zzzzzzzz**]: Ttttttttt

1. INTRODUCTION

Multi-media systems such as medical image processing and video compression algorithms, typically use a very large amount of data storage and transfers. This is especially a problem for low-power embedded system realisations because the memories and bus transfers are responsible for most of system energy waste (between 50 and 80%). Therefore, optimizing the global memory accesses of an application in a so-called Data Transfer and Storage Exploration (DTSE) [1] stages is a crucial task for achieving low power realizations. These optimisations have both platform independent and specific phases. The first phase have a positive influence on both energy and performance (irrespective of the architecture) since it removes redundant data transfer and storage and it improves the locality of production/consumption of the data. However, this phase need to be complemented by a platform specfic one to exploit the opportunities created. This is especially true for highly constrained architectures like those found in programmable platforms due to resource limitations.

In multimedia, the vast majority of programmable processor use cache controllers to manage their on-chip memories. These controllers decide at run-time when and how the data is copied from the higher memory hierarchy layers to the cache architecture [2]. In this way the designer does not have to worry about managing the memory hierarchy, especially for the dynamic part of the application. However, this can become a run-time bottleneck especially due the cache misses when the required data is no longer present in the on-chip memory. These misses translates into an excess of system bus load and energy because of the traffic of data between main memory and the cache (hierarchies). This results into cycles wasted when the processor is stalled waiting for the data to arrive and in energy spent in reloading the cache and it limits the performance of shared-memory multiprocessor based implementations.

All this unnecessary overhead can be avoided at the source-level code. However, it requires knowledge of the underlying platform architecture. This knowledge can be exploited using platform-aware transformations. Optimisations oriented to minimise the miss rate in the data cache (D-cache) are those exploiting the limited lifetime of the data during program execution (namely inplace optimisations [3]) and those exploiting the memory data organisation freedom [4, 5]. This reduction in miss rate leads to important savings in energy (see Section 3). However, they typically require to re-write the index expressions of the affected data arrays, thus resulting in a much more complex addressing functionality. This potentially large overhead in addressing can lead to factors overhead in execution time and to hide the benefit in execution time of the more efficient cache utilisation.

The new addressing code is dominated by index expressions of piece-wise linear nature and it becomes a bottleneck for the linear nature of most commercial Address Calculation Unit (ACU) architectures. Therefore, it should be removed before going to the more pointer level specific address optimisation techniques like those found in conventional [6] or state-of-the-art compilers [7]. Moreover, conventional automated address optimisation approaches [8, 9, 10, 11, 12] do not allow this for address code containing modulo/division operations. To remove this large bottleneck we have developed efficient address optimisation techniques. These are based on the use of source code level transformations which are largely independent of the targeted instruction-set architecture [13, 14], but which need also to be complemented by more architecture specific ones [15]. Using this techniques the addressing code is globally optimised, resulting in factors overall improvement in execution cycles when compared to their data-organised versions.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada

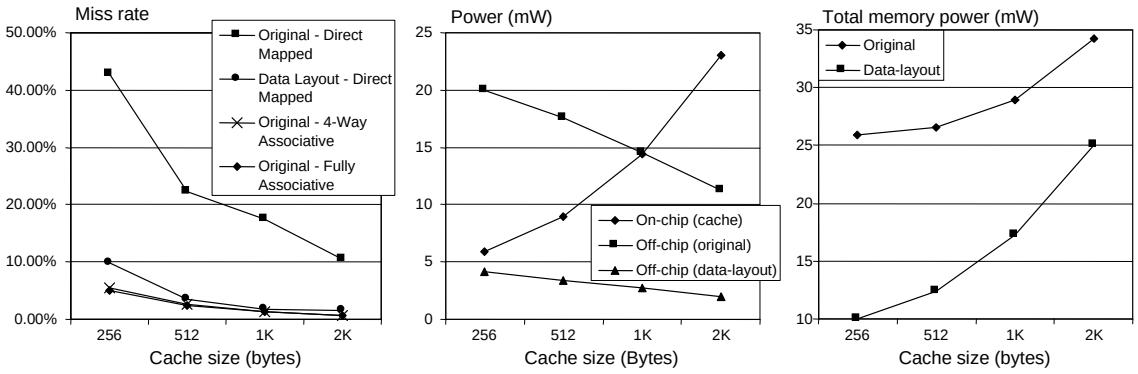

Figure 1: D-cache miss-rate evolution for the MPEG4 Motion Estimation driver for different cache sizes and controller types, including off-chip/on-chip power breakdown and total power evolution for different cache sizes (for a direct mapped cache controller).

The clear benefits of the combined approach to aggressively improve cache utilisation by source code transformations which introduce complex addressing which is then reduced again by a high-level address optimisation stage, has to our knowledge not been studied in the literature. We will provide a systematic approach to achieve that in this paper. That approach is illustrated on two real-life application drivers of industrial relevance, using three popular programmable processor architectures, showing important gains in cycle count and energy consumption.

2. APPLICATION DRIVERS DESCRIPTION

Due to flexibility requirements of the implementation an instruction-set processor architecture is preferred over a fully custom one. The architectures which we target in this paper, consist of multi-media (extended) processors (such as the Philips TriMedia, Pentium-III MMX) but also more general RISC architectures such as the HP-PA800 architecture. For realistic embedded implementation we assume their "core" versions which have a relatively small local (L1) on-chip cache memory to reduce the energy per L1 cache access (which is our focus here). These processors are connected to a (distributed) shared off-chip memory (hierarchy). Threfore, meeting real-time constraints in such architectures, requires an optimal mapping of the application onto the target architecture.

Our exploration approach is illustrated on two real-life application drivers: a Cavity Detector algorithm and Full Search Motion Estimation kernel for CIF video frames. The Cavity Detector is an image processing application used mainly in the medical field for detecting tumor cavities in computer tomography pictures [16]. The initial Cavity Detector algorithm consists of three modules expressed in about 100 lines of flat (non-hierarchical) C-code. Each module has one image frame (of 1280×1000 pixel) as input and one as output. When this initial algorithm is mapped into an embedded system, the system bus load and power requirements are quite high. The main reason is that each of the functions reads an image from shared background memory, and writes the result back to this memory.

At our laboratory, much effort has been already spent on optimising both drivers at the source-level code, mainly by applying the platform independent transformation stage of the DTSE methodology [17, 18]. In this paper we will study the data cache related trade-offs during the platform specific mapping stage.

3. DATA-LAYOUT OPTIMISATION TECHNIQUES FOR D-CACHES

A *cache miss* happens whenever a block of data requested by the CPU is not found in the cache. Therefore, the fist time a block has be copied into the cache can also be considered as a miss (namely a *compulsory miss*). However, only those misses related to blocks that have been previously allocated in the cache lead to an overhead. In this case, two possible situations exist [2]: (1) when the cache cannot contain all the blocks needed during execution, *capacity misses* will occur because of blocks being discarded and later retrieved; (2) if the block placement strategy is set associative or direct mapped, *conflict misses* (in addition to compulsory and capacity) will occur because a block can be discarded and later retrieved if too many blocks map to its set.

For the D-cache, optimisations are available which are oriented to minimise both capacity and conflict misses. Capacity misses can be greatly reduced by exploiting the limited life-time of the data (inplace oriented optimisations [3]) and the conflict misses by applying data-layout oriented transformations [5].

For our experimental setup we obtain first the transformed C code from our prototype data layout transformation tool [20]. Thus, we now have the initial (reference) and the data layout transformed codes. We have used the SimpleScalar simulator tool set [19] for simulating cache performance for varying cache sizes. The cache is simulated using the faster and functionally correct sim-cache simulator for measuring cache performance.

Existing processors were used to measure the performance values. The code versions have been compiling the native compilers and enabling their most aggresive speed oriented optimisation features. The processors used are Hewlett-Packard's PA-RISC 8000, Intel's Pentium-MMX-III, and TriMedia's TM1000.

After cache simulations we have observed that both drivers are dominated by misses in the D-cache (the miss rate for I-cache is about three orders of magnitude smaller than for D-cache). Therefore, we have focussed on optimising the D-cache miss rate by applying main memory data-layout oriented program code transformations. Simulations have been performed using different cache controller types: a 4-way associative cache for the non optimised version and a direct mapped for both optimised and non-optimised versions. Also, different capacity sizes (with the number of cache lines ranging from 256 till 2024) have been evaluated. Figure 2 and 1 shows the miss rate after simulations for both controller types.

For the non-optimised code, the miss rate obtained using a direct mapped controller is rather large when we compare it to the one obtained when using 4-way associative controller. However, when using the cache optimised code, the miss rate behaves equally well for both controller types. This is clearly an important result because it enables the use of the simpler direct mapped cache controllers for embedded applications, instead of the more costly, and power

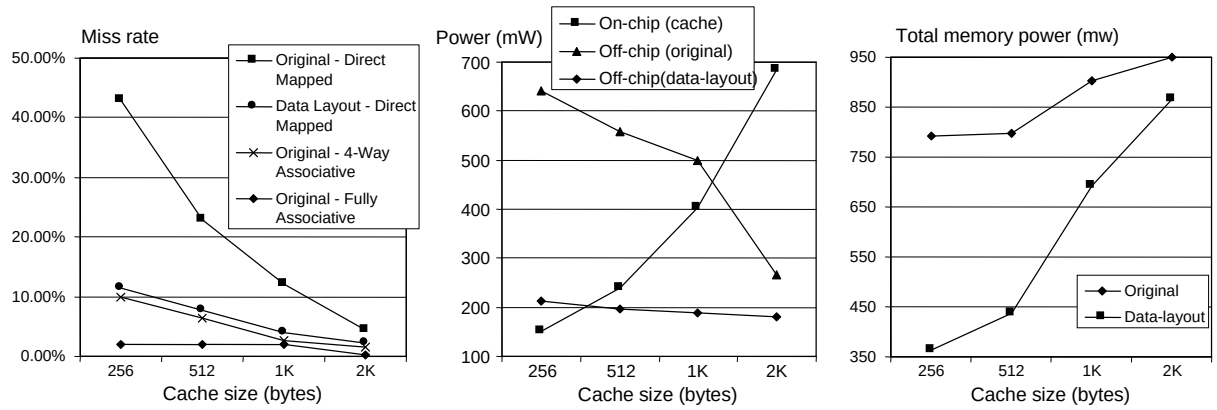

Figure 2: Data-cache miss-rate evolution for the Cavity Detector driver for different cache sizes and controller types, including off-chip/on-chip power breakdown and total power evolution for different cache sizes (for a direct mapped cache controller).

hungry, set-associative controller types.

4. HIGH-LEVEL OPTIMISATION OF THE INDEX/ADDRESS COMPUTATION

Cache oriented transformations require the introduction of complex modulo and integer division operations in the index expressions. This is because, transformations exploiting the limited lifetime of the data requires complex modulo operations in the address code to express in time the folded nature of the relation between the original and the transformed address spaces in main memory.

```
for(x=1; x<N-2; x++) {
  for(y=1; y<M-2; y++) {
    for(k=-1; k<1; k++) {
      ....
      A[x][y] += B[x+k][y]*C[abs(k)];
      ....
    }
    A[x][y] /= tot; }}
```

(a)

```
for(y=0;y<M+2;y++) {
  for(x=0;x<n+2;x++) {
    ....
    if(x>=0 && x<N && y>0 && y<M-1)
      D[x%3] = B[(y*N+x%3)%160+(y*N+x%3)/160*256+ 96];
    ....
    if(x-1>=1 && x-1<=N-2 && y>=1 && y<=M-2)
      for(k=-1;k<=1;k++)
        acc+=D[(x-1+k)%3]*C[abs(k)];
    acc/=tot; }}
```

(b)

Figure 3: Illustration of data cache oriented transformations in the Cavity Detector driver: (a) initial code; (b) transformed code.

On the other hand, data-layout oriented transformations require to create gaps of unused memory locations within array of data allocated in main memory, as well as to allocate in memory different arrays in an interleaved manner. Threfore, these data-layout oriented transformations also require modifications in the original addressing code by introducing complex modulo and integer divisions operations. Figure 3 illustrates the effect of such transformations on a piece of code representative of the Cavity Detector driver. The combined result is that nested modulo expressions are even present.

As a result, for both application drivers described in Section 3, a large overhead in addressing is created. For both drivers, this overhead is mostly due to the more complex index expressions (mainly piece-wise linear) introduced during the inplace [3] and data-layout [5] DTSE oriented optimisation steps. In both drivers, a considerable number of costly modulo operations are executed and traditional compilers completely fail to efficiently eliminate them. Hence, the execution time increases considerably: a factor between 2-20 depending on the application and the target processor architecture.

```
for(y=0;y<10;y++){
  for(x=0;x<100;x++){
    if(x>1) A[(y%3)*3
              +(x-2)%3]=...
    if(x>4) ...=A[(y%3)*3
              + (x-5)%3]
}}
```

(a) Initial

```
for(y=0;y<10;y++){
  v_y = (y%3)*3;
  for(x=0;x<100;x++){
    v_yx = (x-2)%3 + v_y;
    if(x>1) A[v_yx] = ...
    if(x>4) ... = A[v_yx];
  }}
```

(b) Index expression optimisation

```
for(y=0;y<10;y++){
  if(p_y>=9) p_y-=9;
  for(x=0;x<100;x++){
    if(p_x>=3) p_x-=3;
    v_yx = p_x + p_y;
    if(x>1) A[v_yx] =...
    if(x>4) ...= A[v_yx];
    p_x++;}
  p_y+=3;}
```

(c) Address pointer generation

Figure 4: Illustration of processor independent address transformations for the Cavity Detector driver: (a) initial code; (b) after algebraic and modulo oriented transformations; (c) after execution cost oriented transformations.

Our novel design script allows to remove this overhead at the source-level code by reducing the complexity of the index arithmetic and the amount of address operations [15]. This high-level optimisation stage consists on two main stages. The first one exploits algebraic and modulo properties to reduce the number operation instances. This stage does a refactorisation of addressing expressions [21] (containing also modulo/division operations) that compilers are not capable to perform (see Fig 4b).

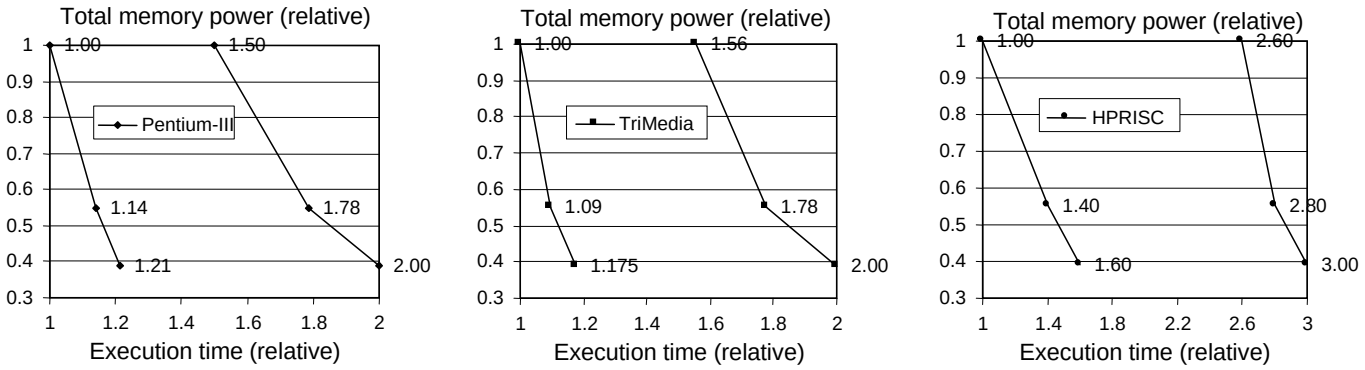

Figure 5: Power and performance trade-offs for the Motion Estimation kernel for different processors and memory data-layouts. Lines at the left represent trade-offs before the address optimisation phase and at the right after it.

When complementing this with aggressive code hoisting techniques across both loops and conditionals, the number of executed operations is minimised. At that point, the second stage can focus on reducing the execution cost of the (remaining) modulo/division operations by transforming these onto a efficient combination of linear induction variables and conditional code (see Fig 4c) [15].

High-level address transformations also have a positive effect on miss rate for both I-caches and D-caches. We have simulated the cache behaviour and we have observed an average of a 10% improvement in miss rate for the D-chace for the drivers described in Section 2. However, this improvement is limited ahen compared by the ones achieved using data-layout oriented transformations (see Section 5

Table 1: Array-size/power/speed trade-offs for the Motion Estimation.

Data-layout trade-off	Array-size Overhead	Avrg.Power Gain	Avrg.Speed Degradation
Light	30 %	45 %	14-40 %
Aggressive	3X	60 %	18-60 %

Table 2: Array-size/power/speed trade-offs for the Cavity Detector.

Data-layout trade-off	Array-size Overhead	Avrg.Power Gain	Avrg.Speed Degradation
Light	2 %	45 %	8-13 %
Aggressive	2X	65 %	24-87 %

5. TRADE-OFF BETWEEN PERFORMANCE, POWER AND MEMORY SIZE

We now discuss how the above technique can be applied to achieve important trade-off's. For both drivers we have started with an initial algorithm which has not been optimized for data transfer and storage.

In the first step, we obtain the cache miss rates for different cache sizes and using a simpler direct mapped cache controller. This is shown in Figure 2 for the Cavity Detector algorithm and in Figure 1 for the Motion Estimation kernel. Once we have obtained the miss rates for different cache sizes, we compute the total (off-chip plus on-chip) required power for each driver. We observe that for both drivers a cache size with 256 bytes consumes the least total power. For smaller cache sizes the total consumed power will be dominated by the accesses (due to misses) to the off-chip memory. For larger sizes of the cache the total power will be dominated by the accesses to the on-chip memory.

In a next step we choose the power optimal cache size of 256 bytes to decide on a trade-off between size (the overhead in memory space in the data layout optimization process as compared to the initial algorithm) and the reduction in miss rate as shown in Tables 2 and 1. Thus depending on the design constraints, the designer can now either choose a lower power solution with some overhead in size and vice-versa. For the Cavity Detector algorithm we have decided to choose two versions representative of extreme points in the search space and observe the trade-offs: one with a 2% memory usage overhead (labeled "Light") and one with a $2X$ (labeled "Aggressive") usage overhead. For the Motion Estimation kernel the points selected are responsible for a 30% overhead in memory usage (labeled "Light") and a larger $3X$ overhead in memory usage (labeled "Aggressive").

In the final stage, we perform address optimizations [15], so as to remove the overhead in addressing operations introduced in the data layout optimization process. Figures 6 and 5 show that we are able to largely remove the overhead in addressing. This is visible by comparing the trade-off achieved before applying the high-level address optimisation phase and (shaded line) and after it (solid line).

For the Cavity Detector algorithm the "Light" data-organised version delivers a reduction of almost a factor of 2 in cache related power with respect to the globally transformed version by trading-off a 10% of the CPU cycles and a 2% of the array size. However, for the Motion Estimation kernel we can achieve a larger reduction in cache related power (almost a factor 3) with respect to the initial version by trading-off between the 21% of the CPU cycles and a 30% of the array size.

Note that the approach we have used in this work is mostly power centric namely, we first optimize for power then size and lastly for performance. But the above technique can be used for a performance centric approach too. In that case, the designer should first choose the cache size with least number of misses and then optimize for power by trading-off performance (and/or size) with power.

6. CONCLUSIONS

In this paper we have shown how to efficiently combine memory data-layout and address oriented transformations in a systematic way for mapping data-intensive applications into architectures containing cache memories. The clear benefits of the combined approach has been illustrated by two real-life applications of industrial relevance, using popular programmable processor architectures. Important gains in energy (up to a factor 3 less) with limited impact on executed CPU cycles are reported.

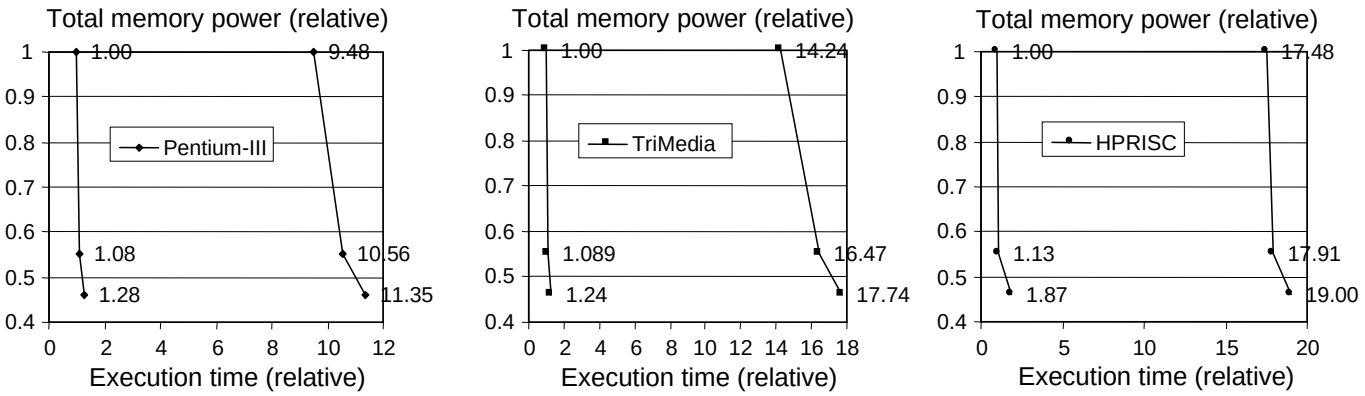

Figure 6: Power and performance trade-offs for the Cavity Detector algorithm for different processors and memory data-layouts. Lines at the left represent trade-offs before the address optimisation phase and at the right after it.

The results of this paper leads to a systematic trade-off (supported by tools) between memory power and speed which has up to now not been feasible for the targeted systems.

7. REFERENCES

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

[2] D.Patterson, J.Hennessey, "Computer architecture : A quantitative approach", *Morgan Kaufmann Publ.*, San Francisco, 1996.

[3] E. De Greef, F. Catthoor, H. De Man, *Program Transformation Strategies for Memory Size and Power Reduction of Pseudo-regular Multimedia Subsystems* IEEE Trans. on Circuits and Systems for Video Technology Vol. 8, No. 6, pp. 719-733, October 1998.

[4] P.Panda, N.Dutt, A.Nicolau, SOMETHING MORE RECENT *Memory data organisation for improved cache performance in embedded processor applications* Proc. Intnl. Symp. on System Synthesis (ISSS), 1996.

[5] C.Kulkarni, F.Catthoor, H.De Man, Advanced Data Layout Optimization for Multimedia Applications, Proc. Workshop on Parallel and Distributed Computing in Image, Video and Multimedia Processing (PDIVM'00), IPDPS'2000, May 2000.

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

[7] S.Muchnick, *Advanced Compiler Design and Implementation* Morgan Kaufmann Publishers, 1997.

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

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

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

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

[12] S.Udayanarayanan, C.Chakrabarti, "Address Code Generation for Digital Signal Processors", *Proc. ACM Design Automation Conf.*, pp. 353-358, Las Vegas, June 2001.

[13] M.Miranda, M.Janssen, F.cattjoor, H.De Man, "High-level address optimisation and synthesis techniques for data transfer intensive applications", *IEEE Trans. VLSI Systems*, XXXXXX

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

[15] 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. Workshop on Signal Processing Systems (SIPS)*, 2000.

[16] M.Bister, Y.Taeymans, J.Cornelis, "Automatic Segmentation of Cardiac MR Images", *Computers in Cardiology*, IEEE Computer Society Press, pp.215-218, 1989.

[17] K. Danckaert, F. Catthoor, H. De Man, *Platform independent data transfer and storage exploration illustrated on a parallel cavity detection algorithm*. Proc. PDPTA (CSREA Conf. on Parallel and Distributed Processing Techniques and Applications), 1999.

[18] Kristof Denolf, Peter Vos, Jan Bormans and Ivo Bolsens, Cost-efficient C-Level Design of an MPEG-4 Video Decoder *Proc. Workshop on Power and Timing Modeling, Optimization and Simulation*, 2000.

[19] D.Burger, T.Austin, "The Simplescalar Toolset", version 2.0, online document available at http://ww.cs.wisc.edu/ mscalar/simplescalar.html.

[20] C.Kulkarni, M.Miranda, C.Ghez, F.Catthoor, H.De Man, "Cache Conscious Data Layout Organization For Embedded Multimedia Applications", *Proc. 4th ACM/IEEE Design and Test in Europe Conf.*, Munich, Germany, March 2001.

[21] J.M.Janssen, F.Catthoor, H.De Man, *A specification invariant technique for operation cost minimisation in flow-graphs*, Proc. Intln. Symp. on High-level Synthesis, pp. 146-157, 1994.