A Decoupled Architecture of Processors with Scratch-Pad Memory Hierarchy

A. Milidonis[1], N. Alachiotis[1], V. Porpodas[1], H. Michail[1], A. P. Kakarountas[1], and C. E. Goutis[1]

[1] VLSI Design Lab., Electrical & Computer Engineering Department, University of Patras, Rio, Greece
e-mail: milidon@ee.upatras.gr

Abstract

We present a decoupled architecture of processors with a memory hierarchy of only scratch–pad memories, and a main memory. The decoupled architecture also exploits the parallelism between address computation and processing the application data. The application code is split in two programs the first for computing the addresses of the data in the memory hierarchy and the second for processing the application data. The first program is executed by one of the decoupled processors called Access which uses compiler methods for placing data in the memory hierarchy. In parallel, the second program is executed by the other processor called Execute. The synchronization of the memory hierarchy and the Execute processor is achieved through simple handshake protocol. The Access processor requires strong communication with the memory hierarchy which strongly differentiates it from traditional uniprocessors. The architecture is compared in performance with the MIPS IV architecture of SimpleScalar and with the existing decoupled architectures showing its higher normalized performance. Experimental results show that the performance is increased up to 3.7 times. Compared with MIPS IV the proposed architecture achieves the above performance with insignificant overheads in terms of area.

1. Introduction

Today's modern applications require very large amounts of data that are accessed through several layers of temporal memory storage. Accessing data from the memory hierarchy determine to a large extent the system's performance.

Cache is the most common memory used for temporal data storage. However, its performance is frequently limited by a number of factors such as data are copied in fixed size blocks equal to the cache line size, conflict misses increase drastically when the number of processors or functional units fed by a memory hierarchy grow. Moreover, data are written in cascaded addresses of cache lines which reduces the cache's flexibility of avoiding conflicts.

Scratch-pad memories are used for overcoming these problems, since no conflict misses occur and data blocks written, can be of any size less than the scratch-pad's size. Moreover, accessing a scratch-pad memory costs less in power and time than cache which consumes more area.

However, comparing to cache, scratch-pad memories do not include a logic circuit responsible for performing its read and write operations in bursts. In platforms using cache memories, the data transfers between hierarchy levels are performed by cache circuits which copy line sized blocks while the processor is not concerned on how the data are fetched. In platforms with scratch-pad memory hierarchy, the processor has to perform each data block transfer between levels usually using Direct Memory Access (DMA) circuits. However for their initialization, extra cycles are needed by the processor for computing the data block source and destination addresses in the memory layers and for programming the DMAs to perform the transfers. To save these extra cycles a separate processor strongly communicating with the memory hierarchy is required.

In this paper a decoupled processor architecture is presented having a memory hierarchy of only scratch-pad memories, and a main memory. In this architecture, the first of the processors performs data transfers (Access processor) between the main memory, the layers of the scratch-pad memory hierarchy and the register file of the Execute processor and the second performs data processing (Execute processor).

The main contributions of this architecture are that the data transfers between the layers of memory hierarchy are controlled by the Access processor. This processor uses memory management compiler methods and exploits the flexibility of storing data in scratch-pad memories avoiding misses and reducing memory accesses while using prefetching techniques. By these, the execution time of applications is reduced compared to existing decoupled processors without scratch-pad memories.

This paper is organized as follows. Section 2 presents the related work. The proposed decoupled processor architecture is presented in Section 3. Experimental results are given in section 4. Finally, section 5 concludes this paper.

978-3-9810801-2-4/DATE07 © 2007 EDAA

2. Related work

Decoupled architectures have been researched in the past. In [1] the main idea of decoupling is presented. A more recent decoupled architecture with additional hardware for increasing performance is shown in [2]. A cache hierarchy exists between the main memory and the processor that computes the memory addresses. It should be noted that none of the above architectures handle scratchpad memory hierarchy addressing.

Scratchpad management techniques are presented in [5]. [7] describes Scratchpad compiler techniques that outperform techniques applied in cache. [3] and [6] present run time techniques, applied in uniprocessor platforms. In [14] the benefits on hiding memory latency are presented, using DMA combined with a software prefetch technique and a customized on-chip memory hierarchy mapping. In [8] there is a study showing that decoupled processors with caches outperform uniprocessors with caches and that this speedup is increased as the memory latencies increase.

In [3] run time management of scratchpad is presented. The platform used contains one level of cache, one of scratch pad, a DMA engine, and a controller that controls the DMA engine. The computation of memory addresses are done by the processor that also handles processing data, which means the architecture of [3] is not decoupled.

Finally, in [4] an SRAM memory architecture is used on a VLIW processor while a distributed address generation architecture with a loop acceleration unit calculates the memory addresses of data to be transferred to/from the SRAMs. The start addresses of the data blocks that are to be transferred are generated at compile time which increases the instruction code size significantly. This is avoided in our architecture by assigning the delivering of data to the Access processor which will calculate each time the new start addresses of the data blocks executing repeatedly one code routine, thus keeping the instruction code size small.

Continuing in [4], the use of address generators for fetching data to the functional units contributes a lot on increasing the performance of applications with stream computations but in the case of random data accessing there shall be serious initialization overheads. The transfer from the main memory to L1 scratch-pad is not on the paper's scope. In our work the complete transfer of data from main memory to the register file of the Execute processor and vice versa is performed by the Access processor.

3. Proposed Architecture

The decoupled Architecture of processors is shown in Fig. 1. It consists of a main memory, a memory hierarchy of various levels of SRAMs (next are referred as Scratch-pad memories), DMAs for the transfers of data between the above levels, an Access Processor, an Execute processor and a small buffer used for communication between the two processors.

The DMAs are as many as the levels of memory hierarchy and are used for the transfer of data in bursts between them.

The task for the transfer of data between the layers of memory hierarchy is critical for the system's performance. Using scratch-pad in our architecture, data can be placed at any of its memory location. The block sizes transferred to/from scratch-pad can be of any size. For our memory hierarchy a compiler is used for applying memory management techniques. The Access processor executes the compiler's output taking design time and also run time decisions, increasing the system's performance.

The Access processor is as an integer processor. Its main operations are the computation of the addresses in the

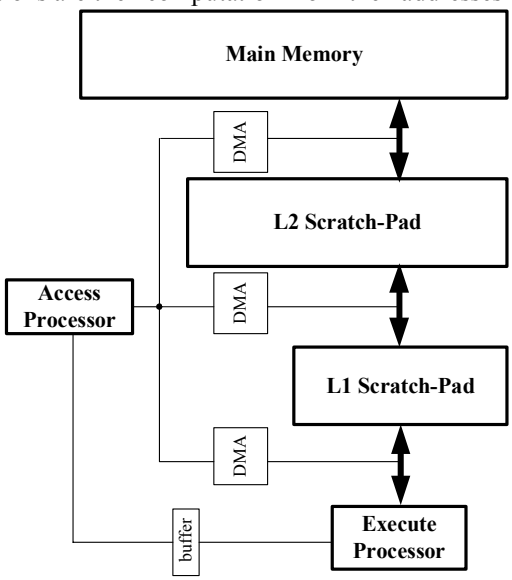

Fig. 1 Proposed Architecture

memory hierarchy for storing the application's data types and the control of the DMAs for the transfers of data between the levels of memory hierarchy.

Computation of the addresses need very small amount of data compared to the data of the Execute processor (e.g. the base address of a data type stored in a level of memory hierarchy). For that reason the Access processor does not need a separate data memory hierarchy. It uses one memory hierarchy for storing data and program without having serious performance overhead.

The Execute Processor is a processor that can handle both integer and floating point numbers. Its main task is the processing of application's data. It does not contain any load/store instructions in its instruction set since this task is assigned to the Access processor and the DMAs. The DMA between L1 scratch-pad and the Execute processor (Fig. 1), executes instructions provided by the Access processor for placing data from L1 scratch-pad to the Execute processor's

register file and vice versa. Having this functionality the Execute processor focuses only on processing application's data and since the proposed architecture is decoupled these data are prefetched in its register file which leads to increased performance as the delay of data to be fetched is most of the times eliminated.

Communication between the Access and Execute processors is a critical task in order for the system to operate efficiently. For that reason it must be sure that the correct data are placed in the Execute processor's register file before the execution of the set of instructions corresponding to these data. Also, it must be sure that the action of sending data from the Execute processor's register file to L1 scratch-pad has been concluded before new data are written in the same registers replacing the old ones. For that reason the two processors communicate through the buffer in Fig.1 with a simple handshake protocol.

They must also communicate in the case where the execution path in the Control Data Flow Graph changes (e.g. when a branch is executed in which different data are required for each case of the branch). The above buffer is used in this case by the two processors for declaring which of the execution paths is followed.

The size of the buffer is small (16 bytes are enough) since it is needed for writing information about data that exist in the Execute processor's register file and for currently executed branches. Also it is not accessed as frequently as the register file of the Execute processor but only when a data block has been transferred or a critical branch occurs which does not comprise a significant overhead to the system's performance.

An example of the code executed by the decoupled processors is shown in Fig.2. The code calculates the average values of sixteen numbers repeatedly. In the example there are two data types used, the input data type whose values are averaged and the output data type which contain the result of the average process. When the Execute processor requests a block to be transferred to/from its register file, it writes the value 1 to the communication buffer when the input data type is requested and 2 for the output data type. The Access processor reads these values and executes the transfer for the correct data type. In effect the above is an implementation of handshake protocol.

The Execute processor (Fig. 2) using the LOAD_com command (line: 1) sends an interrupt request to the Access processor for transferring the first eight numbers to the register file and writes the value 1 to the first register of the buffer (BUF[1]) to declare to the Access processor that a new block of data is requested from the input data type. In line 3 there is a check whether the Access processor has transferred the requested block to the Execute processor's register file (the Access processor writes the value 0 to BUF[1] when the transfer has been completed). This is done using while loops (lines 3, 13). In the case that the Access processor has not written the zero value to the first register of the buffer, the processing of the next eight numbers is stalled.

```
Execute Processor
1  :LOAD_COM;                 Access Processor
2  :while(1){                 1:  while(1){
3  : while(BUF[1]!=0){ }
4  : LOAD_COM;                2:  Transfer(M_2_L2);
5  : R0=R0+R1;
6  : R0=R0+R2;                3:  Transfer (L2_2_M);
7  : R0=R0+R3;
8  : R0=R0+R4;                4:  Transfer (L2_2_L1);
9  : R0=R0+R5;
10: R0=R0+R6;                 5:  Transfer (L1_2_L2);
11: R0=R0+R7;
12: R0=R0+R8;                 6:  }
13: while(BUF[1]!=0){ }
14: LOAD_COM;                 7:  Int_Routine1:
15: R0=R0+R11;                8:  Transfer (L1_2_RF);
16: R0=R0+R12;
17: R0=R0+R13;                9:  Int_Routine2:
18: R0=R0+R14;                10: BUF[1]=BUF[1]-BLOCKSIZE;
19: R0=R0+R15;
20: R0=R0+R16;                12: Int_Routine3:
21: R0=R0+R17;                13: Transfer (RF_2_L1);
22: R0=R0+R18;
23:  while(BUF[2]!=0){ }      14: Int_Routine4:
24: R9=R0/16;                 15: BUF[2]=BUF[2]-BLOCKSIZE;
25: STORE_COM;
26:}
```

Fig. 2 Decoupled Processor pseudo-code example

Continuing, the Execute processor requests the next block of data using the LOAD_com command (line: 4) and while the Access processor transfers (prefetch) it to the register file in different locations than the first block, the Execute processor calculates the first 8 sums of the first block. The same technique for writing data to the Execute processor's register file from L1 scratch-pad is used for transferring them to the opposite direction. In the example of Fig. 2 the R9 register contains the value of output data type which must be stored in L1 scratch-pad. The Execute processor sends an interrupt request to the Access processor for uploading the value of R9 and writes to the second register of the buffer the value 2 using the STORE_COM command (line: 25). Before writing the new value to R9 it must be sure that the Access processor has already sent the previous average number to L1 scratch-pad (line:23). If the second register of the buffer is zero then the Execute processor writes the next average value to R9 (line: 24) else it waits.

The Access processor transfers data through the main memory to L1 scratch-pad using a polling technique. In Fig. 2 this is shown in pseudo-code in the Access processor code (lines: 1-6). It also receives two kinds of interrupts. One from the Execute processor for writing/reading new data from L1 scratch-pad to its register file (lines: 8, 13) and vice versa and one from the lower DMA indicating that the last transfer is completed and the registers in the buffer must be updated (lines: 10, 15). Using interrupts for transferring data to/from the register file from/to L1 scratch-pad ensures that the stalling time of the Execute processor until data arrive will be minimized.

It is important to notice that the program of the Execute processor is much less sized compared to the corresponding one of a uniprocessor as there are no load instructions for every new data word but only a request for a block of eight words. This decreases even more the application's execution time. Also the code size is not increased by the amount of instructions used for the two processors' communication as indicated in Fig. 2 since it is very small compared to the application's total code size.

Observing the architecture of the decoupled processor in Fig. 1 it can be concluded that transferring of data through the memory levels of hierarchy is a task separated from processing and assigned to the Access processor which uses prefetching techniques. The Access processor provides a flexible way of placing data to the levels of the memory hierarchy with no misses following memory management techniques. Also, this memory hierarchy can operate in a pipelined way since the Access Processor may transfer every available data and not a distinct data type through the whole path at a time, decreasing significantly the total transfer time from the Execute Processor to the Main Memory. Data are transferred in blocks usually larger in size than the cache line sizes making the instructions for these transfers less and avoiding getting the processing load of the Access processor high. The Execute processor is not occupied with accessing memory but processes the application data which are prefetched in its register file reducing the memory hierarchy's delay.

4. Experimental Evaluation

In this section we present the simulation method and the experimental results of comparing the proposed architecture with the MIPS IV architecture. Also our experimental results are compared to the architectures in [2] and [4].

4.1 Simulation Method

We used the SimpleScalar simulator [9] for measuring the MIPS IV performance. Table 1 shows the SimpleScalar configuration used for our measurements. For fairness, the MIPS IV processor may execute 2 instructions in parallel. By this way the MIPS processor is able to perform load/store operations in parallel with data processing operations. All benchmarks for running in the SimpleScalar simulator were compiled by the gcc compiler using optimization level –O3. For evaluating our architecture we created its behavioral VHDL model using in-order processors for the Execute and Access ones having instruction set identical to that of MIPS IV, one Arithmetic Logic Unit (ALU) and one multiplier. Each processor executes one instruction per cycle. We implemented the front end of a C compiler using lexical and syntax analysis tools and an automated flow for generating the assembly code of the processors. The C codes of the benchmarks that were input to our compiler's front end were optimized manually.

The actual delay times of the memories used are provided from the Cacti model [10]. In the experiments of Fig. 3 and Fig. 4 we used L1 and L2 memories with sizes of 4KB and 16KB correspondingly. In the experiments of Fig. 5 and Fig. 6 we used the memory sizes given above for our architecture while for SimpleScalar the size for L1 cache varied from 4KB to 32KB and for L2 cache from 16KB to 128KB. The main memory is an SRAM of 256KB since this was the largest size that our benchmark required. Its latency is also provided by the Cacti model. In most systems the off chip memory size requirements are much larger and their latencies are also increased. However, as proved in [8] the more the memory latencies increase the better the decoupled processors performance become, comparing to uniprocessors' performance. For that reason the gains of our architecture become larger. For our DMAs we used circuits with a simple functionality for transferring data in bursts. For this operation a more simplified version of the loop unit of [4] which can only execute one loop instruction per cycle is used having 1 cycle latency for each new address calculation.

Table 1. SimpleScalar configuration

Integer alu	1	**Fetch ifqsize**	2
Integer mult	1	**Load/Store Queue**	8
Issue width	2	**RUU**	16
Decode width	2	**Branch Prediction**	Perfect

In [12] a method for estimating the MIPS IV area according to its configuration is given. The technology used was 0.5 micron. We applied this method for 0.18 micron. After estimating the area of each architecture we compared them calculating their ratio for avoiding any method variations between the two architectures. The differences of our architecture with the SimpleScalar's in terms of area are that in our case there are two in order processors with no Register Update Unit (RUU), the Access processor is integer and has no floating point unit and the data memory hierarchy does not include tag arrays. The dominant components for area are the memories and not the processors. For these reasons the area in the decoupled architecture is slightly incremented by 5.1% compared to the one in the SimpleScalar's architecture.

Our benchmark suite consists of five benchmarks: full_search, a motion estimation code; mxm, an integer matrix multiplication program (it contains one initialization and one multiplication nest); detect_roots, a kernel of cavity detector which is a medical image application; rasta_filt, a filtering routine for speech recognition application from MediaBench[11]; and sub_4 a kernel of QSDPCM (Quad-Tree Structured Difference Pulse Code Modulation) [13] which is an inter-frame compression technique for video images.

4.2 Simulation Results

Fig. 3 illustrates the speedup gained using the proposed architecture compared with SimpleScalar's performance according to the configurations given in the previous section. In applications where the temporal locality becomes more dominant (e.g. matrix multiplication) the performance becomes up to 3.7 times higher than SimpleScalar's. This is because the use of scratch-pad allows better compiler methods than those achievable by cache hierarchy, due to cache misses. Also cache makes unnecessary transfers by fetching data through the memory hierarchy that are not needed for processing. This is because used and not used data may be stored together in a cache line and transfers are done in blocks. In cache memory prefetching is done for data blocks having the size of a cache line. They are transferred according to the rules of modulo addressing in which cache misses occur, decreasing performance. In our architecture prefetching is done more flexibly since no misses occur. Also, the blocks transferred in bursts have sizes of multiple cache lines for decreasing their latency. This is more important to the highest levels of the memory hierarchy in which the non-burst response delay is larger.

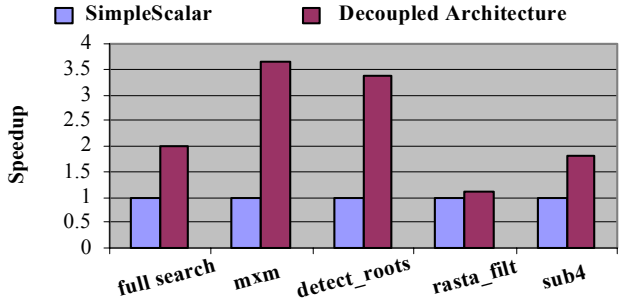

Fig. 3 Normalized Performance of Decoupled Architecture and SimpleScalar

The relative area delay product cost of MIPS IV and of the proposed architecture has been estimated using the delay reports of the Fig. 3 and estimating the area as described in the simulation method section.

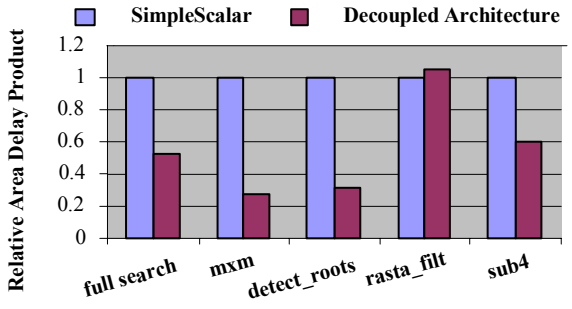

Fig. 4 Comparison between Decoupled Architecture and SimpleScalar in terms of area delay product

Fig. 4 shows that the decoupled architecture costs less in most of the cases except of the rasta_filt case where the cost is slightly higher. From this experiment it is concluded that our architecture performs higher in factors larger than 2 and 3 in most cases without any serious area penalty.

We also examined whether the SimpleScalar architecture using larger conventional caches can outperform the proposed decoupled architecture of processors.

For this we kept the decoupled architecture's memory sizes of L1 and L2 scratch-pads constant at 4KB and 16KB respectively and increased the data cache sizes of SimpleScalar for L1 from 4KB to 32KB and L2 from 16KB to 128KB.

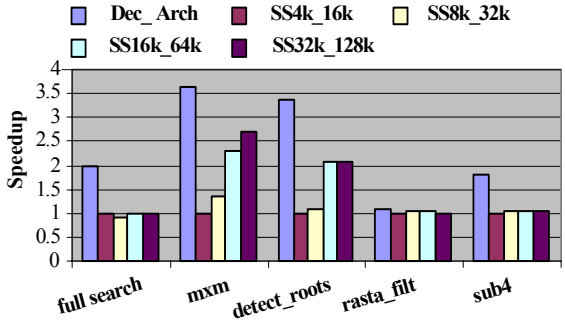

Fig. 5 Normalized Performance of Decoupled Architecture and SimpleScalar when caches of variable sizes are used

The results are given in Fig. 5 where in most cases the SimpleScalar's performance is stable as the memory sizes increase and in other cases is increased until a certain memory size and then becomes stable. The conclusion of these experiments is that the decoupled processor is faster than SimpleScalar in all memory sizes used.

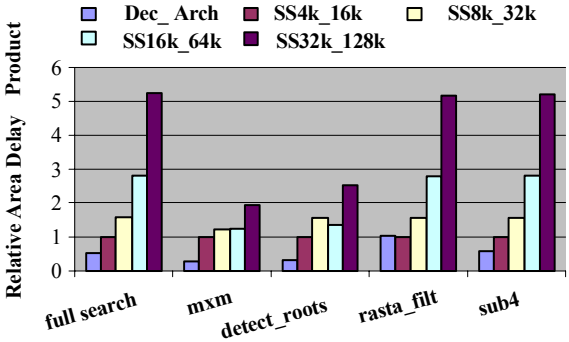

Fig. 6 Area delay product comparison between Decoupled Architecture and SimpleScalar with caches of variable sizes

The relative area delay product for the above variations of memory sizes for SimpleScalar was estimated as above and is shown in Fig. 6. This product becomes far better for the decoupled architecture as the memory sizes in the SimpleScalar's architecture increase.

In the following, we perform a kind of comparison between our results and those of other existing architectures. Since the information about their area was not

available for defining the total hardware cost as with MIPS IV, we converted the results of their measurements in order to be relevant to our simulation method as described below.

In MediaBreeze [2] the 16 times maximum speedup is a result of hardware acceleration units included in the Access processor. More specifically, in each clock cycle four address, five loops (five branch instructions and five increment instructions), three load and one store instructions are computed by the hardware, which corresponds to 18 instructions per cycle. In applications with many loop nests and complex addressing such as motion estimation this hardware has significant impact on increasing performance.

Continuing in [2], a prefetching engine is introduced which increases the speedup from 18 to 28 times. For converting this result to be relevant to our simulation method, we divide it by a factor of 9 since it is assumed (as a worst case) that all 18 extra execution units will operate on 50% of the execution cycles. The relevant increment of the maximum performance is 3.07 and the average value of all the results in [2] is 0.9 which are less compared to ours (3.7 and 2.2 respectively). The average values for [2] here and [4] below are estimated from the corresponding figures in these papers. The higher speedup of our architecture is because we use a more flexible memory hierarchy in which no data misses occur. Also, prefetching is done in our architecture more efficiently since data are transferred through each memory level guided by memory management techniques, executed by the Access processor.

In order to compare our results with those in [4] a conversion of the measurements is also needed since the simulation method is different. The processor in [4] has 8 Function Units (FUs) running in 1 GHz and its performance is compared with XScale which is a RISC processor with 1 FU running on 400 MHz. We convert those simulation results by dividing them by a factor of 10 (8*1GHz/400MHz*50%) since it is assumed (as a worst case) that the execution units will operate on 50% of the execution cycles. In [4] the maximum difference in performance between the two processors occurs for the benchmark GAU, where the architecture in [4] outperforms XScale nearly 20 times. Its converted value to our simulation method is 2 and is below our maximum speedup (3.7). The average values for these testbenches used for MIPS IV and our architecture are 1.2 and 2.2 respectively which shows that our architecture is almost twice as fast as that of MIPS IV.

5. Conclusions

In this paper we have presented a decoupled architecture of processors having only a scratch-pad memory hierarchy. The processors synchronization is done through protocol. We implemented the behavioral VHDL, the compiler's front end and the automated flow for generating the assembly language for the proposed architecture. The experiments we preformed show that the system's performance is increased 2.2 times on average and 3.7 times maximum more than in MIPS IV architecture and also outperform other existing decoupled architectures, while the cost in terms of area is inconsiderable. More work is currently done, focusing on the energy consumed by the proposed architecture.

References

[1] J. E. Smith, "Decoupled Access/Execute Architectures", Proceedings of the 9th International Symposium on Computer Architecture, pp. 112-119, May 1982.

[2] D. Talla, L. K. John, "MediaBreeze: A Decoupled Architecture for Accelerating Multimedia Applications" ACM Computer Architecture News, ACM Press, ISSN 0163-5964, pp. 62-67, vol. 29. no. 5, December 2001.

[3] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, J. Mendias, "An integrated Hardware/Software Approach For Run-Time Scratchpad Management", Proceedings of the 41st annual conference on Design automation, June 07-11, 2004, San Diego, CA, USA.

[4] B. Mathew, A. Davis, "A Loop Accelerator for Low Power Embedded VLIW Processors", Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, September 08-10, 2004, Stockholm, Sweden.

[5] P. R. Panda, F. Catthoor, et al. Data and memory optimizations for embedded systems. ACM TODAES, April 2001.

[6] Kandemir M., et al."A Compiler Based Approach for Dynamically Managing Scratch-pad Memories in Embedded Systems", IEEE Trans. on CAD. 23 (2), pp.:243 – 260. Feb 2004

[7] M. T. Kandemir, A. Choudhary."Compiler-directed scratch pad memory hierarchy design and management", DAC (2002) New Orleans, USA

[8] L.Kurian, T.Hulina, L.D. Coraor, "Memory Latency Effects in Decoupled Architectures", IEEE Trans. on Computers, Vol. 43 No. 10, October 1994

[9] D. Burger and T.M. Austin, "The simplescalar toolset, Version 2.0," Comp. Sciences Dept, UW, Tech. Rep., June 1997.

[10] G. Reinman and N. Jouppi. An integrated cache timing and power model. Technical report, Compaq Western Research Lab,1999

[11] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems" International Symposium on Microarchitecture (1997)

[12] M. Steinhaus, R. Kollaz, J. L. Larriba-Pey, T. Ungerer, M. Valero,"Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models", Workshop on Complexity-Effective Design, June 30, 2001, Göteborg, Sweden

[13] P. Stobach. "A new technique in scene adaptive coding", European Signal processing Conference (EUSIPCO), 1998.

[14] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, A. Thanailakis: A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck. IEEE Trans. VLSI Syst. 14(3): 279-291 (2006)