Proceedings of the 5th WSEAS Int. Conf. on SIGNAL, SPEECH and IMAGE PROCESSING, Corfu, Greece, August 17-19, 2005 (pp132-137)

Exploiting the Distributed Foreground Memory in Coarse Grain Reconfigurable Arrays for Reducing the Memory Bottleneck in DSP Applications

GREGORY DIMITROULAKOS[1], MICHALIS D. GALANIS[2], COSTAS E. GOUTIS[3]
VLSI Design Laboratory, Electrical & Computer Engineering Department
University of Patras, Rio Campus, GREECE

Abstract: - This paper presents a methodology for memory-aware mapping on 2-Dimensional coarse-grained reconfigurable architectures that aims in the minimization of the data memory accesses for DSP and multimedia applications. Additionally, the realistic 2-Dimensional coarse-grained reconfigurable architecture template to which the mapping methodology targets, models a large number of existing coarse-grained architectures. This is exploited for quantifiyng the influnce that the architectural features have on performance improvements achieved by our methodology. A novel mapping algorithm is also proposed that uses a list scheduling technique in which the binding, routing, and scheduling phases are considered together and they are steered by a set of costs. The algorithm transfers the data reuse values in the internal interconnection network instead of being fetched in order to reduce the data transfer burden on the bus network. The experimental results show that memory accesses and execution time are reduced since the mapping methodology efficiently exploits the data reuse opportunities.

Key-Words: - Coarse Grain Reconfigurable Arrays, List Scheduling, Data Reuse, Mapping methodology distributed memory hierarchy.

1 Introduction

Coarse-grained reconfigurable architectures have been proposed for accelerating loops in multimedia and DSP applications in embedded systems. These architectures combine the high performance of ASICs with the flexibility of microprocessors. Coarse-grained reconfigurable architectures consist of a large number of Processing Elements (PEs) connected with a reconfigurable interconnect network. This work focuses on architectures where the PEs are organized in a 2-Dimensional (2D) array and they are connected with mesh-like reconfigurable networks [1]-[4]. In this paper, these architectures are called Coarse-Grained Reconfigurable Arrays (CGRAs). This type of reconfigurable architecture is increasingly gaining interest because it is simple to be constructed and it can be scaled up, since more PEs can be added to the mesh-like interconnect. Also, their coarse granularity greatly reduces the delay, power and configuration time relative to an FPGA device at the expense of flexibility [1].

The aim of mapping algorithms to CGRAs is to exploit the inherent parallelism in the considered applications for increasing performance. An increase in the operation parallelism results in a respective increase in the rate by which data are fetched from memory -called *data memory bandwidth*- which is the major bottleneck in exploiting the inherent parallelism [5]. Thus, there is an imperative need for a mapping methodology to CGRAs for reducing the memory bandwidth requirements. Furthermore, performance is not the only important factor in embedded systems design. Power consumption is equally important in most of the cases (e.g. in handheld devices). As it was shown in [5], memory contributes the most to the power consumption. Generally, power consumption can be reduced by minimizing the number of memory accesses; this is also the case for the CGRAs. So, data memory management techniques taking into account the architectural features of the CGRAs have to be developed for the widespread usage of CGRAs in embedded systems.

This paper presents an automated memory-aware mapping methodology for CGRAs that attempts to minimize the data memory bandwidth by exploiting data reuse in loops of DSP applications. This is achieved by taking advantage of the distributed foreground memory and by proper placing the operations in the architecture PEs for minimizing the number of memory data transfers and their delay cost. With this way, memory bandwidth is freeing up allowing more operations, which require memory accesses, to be executed in parallel in the CGRA; so parallelism is increased.

The large number of architectural decisions in CGRAs makes their design space very large and

complex. For this reason the proposed methodology targets on a generic CGRA template architecture which can model the majority of the existing CGRAs [1]-[4]. This template abstracts the different CGRA configurations in terms of the number and type of PEs, the interconnections among them and their memory interface. Thus, the methodology is retargetable to any type of CGRA. The experimental results illustrate how the performance improvements, by applying our methodology, are related to the architectural decisions. Additionally the experimental results showed a significant reduction in the execution time and the number of accesses to the data memory.

The rest of the paper is organized as follows: section 2 describes the related work, while section 3 presents the CGRA architecture template. Section 4 describes the proposed mapping algorithm. Section 5 presents the experimental results. Finally, conclusions are outlined in section 6.

2 Related Work

Although several CGRA architectures have been proposed in the past few years [1]-[4], a small number of mapping methodologies consider the limited memory bandwidth problem. For reducing the memory bandwidth requirements in the KressArray [6] architecture the mapping algorithm stores the data reuse values in a globas register file. To reduce the number of memory accesses in PACT-XPP [4] when array references inside loops read subsequent element positions, the compiler only reads one element per iteration and generates shift registers to store the other values. However in these works the distributed foreground memory is not exploited neither an exploration of the design space is performed.

Mei et al. [7], proposed a modulo scheduling algorithm for mapping loops on a generic coarse grained reconfigurable architecture and in [8] this approach was applied to an MPEG-2 decoder. In [9] an exploration was performed for different architecture alternatives using that modulo scheduling algorithm for revealing the design space trade-offs. To our best knowledge none of the [7]-[9] works considers a mapping methodology for exploiting the distributed PE local memory network and the data reuse opportunities for alleviating the data bandwidth bottleneck. Although the local register files are used for storing the data values produced and consumed due to data dependences during the mapping phase, they are not used for storing the data reused values from data reuse opportunities, which are present at the source code level. This is due to the fact that the mapping phase takes as input a low level intermediate representation of the applications' loop bodies which is inadequete for array subscript analysis.

A methodology for reducing the accesses to the data memory in pipelined execution of an application was given in [10]. However, in that work there wasn't any exploration performed to show how the performance is influenced by the architecture's characteristics.

3 Generic Architecture Template

In this section, the generic reconfigurable template considered in this work is described. Since it is based on characteristics found in the majority of the 2D coarse-grained reconfigurable architectures [1]-[4], it can be used as a realistic model for mapping computational intensive applications to such type of architectures. The proposed architecture template is shown in Fig. 1 and consists of 4 basic parts: (a) the PEs organized in a 2D interconnect network, (b) the configuration memory, (c) the memory interface, and (d) the main data memory. Each PE is connected to its nearest neighbours while there are cases [3] where there are also direct connections among all the PEs across a column and a row.

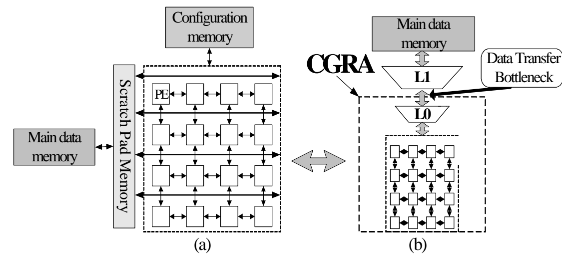

Fig. 1. (a) CGRA architecture template, (b) Memory hierarchy.

In the proposed template, each PE contains one Functional Unit (FU), which it can be configured to perform a specific word-level operation, each time. Typical operations supported by the FU are: ALU, multiplication, and shifts. The FU in the proposed CGRA template also supports predicate operation. Thus, loops containing conditional statements are supported by the CGRA template through "if-conversion" process [12]. For storing intermediate values between computations and memory fetched values, a local RAM exist inside a PE. The PE's input operands can come from three different sources: (a) from the same PE, (b) from another PE, and (c) from the memory buses. A reconfiguration register inside each PE stores control values that determine how the FU, the storage unit, the interconnections among the PEs, and the multiplexers are configured. The configuration memory of the CGRA (Fig.1a) stores the whole

Proceedings of the 5th WSEAS Int. Conf. on SIGNAL, SPEECH and IMAGE PROCESSING, Corfu, Greece, August 17-19, 2005 (pp132-137)

configuration for setting up the CGRA for executing the application. Additionally, configuration caches distributed in the CGRA and reconfiguration registers inside the PEs are used for the fast reconfiguration of the CGRA.

The CGRA's memory interface (Fig.1b) consists of: (a) the memory buses, (b) the scratch pad memory [11] which is the first level L1 of the CGRA's memory hierarchy, and (c) the base (zero) memory level, called L0, which is formed by the local RAMs inside the PEs. As it happens in the majority of the existing CGRA architectures [2], [3] the PEs of our template architecture, residing in a row share a common bus connection to the scratch-pad memory.The L1 serves as a local memory for quickly loading data in the PEs of the CGRA. The interconnection network together with the L0 acts as a high-bandwidth foreground memory, since during each cycle several data transfers can take place through different paths in the CGRA. Our methodology focuses on reducing the memory accesses in the L1 level by exploiting the L0 level structure. Typically, CGRA architectures [1]-[3] have local storage units inside a PE. However, our approach is the first one that utilizes the L0 for reducing the memory bottleneck.

4 Proposed methodology

The block diagram of the proposed mapping methodology is shown in Fig. 2. The input is the application description in C language. We have utilized the front-end of the SUIF2 compiler infrastructure [13] to create the intermediate representation (IR) of SUIF2. We have used existing and we have developed new SUIF2 passes for performing analysis and transformations on applications' loops. More specifically, data-flow analysis is used to identify the data reuse opportunities in the IR. Transformations in the IR are performed to create the Data Dependence Graph (DDG). Passes for dead code elimination, common sub-expression elimination and if-conversion transformations, have also been utilized. The considered analysis and transformation flow is enclosed in the dashed line of Fig. 2.

4.1 Mapping methodology description

As shown in Fig. 2, the first input to the mapping algorithm is a DDG, representing the loop body, with the extra information concerning the data reuse among operations. We call this graph Data Dependence Reuse Graph (DDRG). The DDRG is a directed graph $G(V, E, E_R)$, where: V is the set of DDRG nodes representing the operations of the loop body. Each DDRG node is annotated with the type of operation, its priority and the number of memory operations it requires. E is the set of data edges showing data dependencies among the operations.

Finally, E_R are non-directional edges showing when data reuse exists among the DDRG nodes. The E_R edges are further annotated with the names of variables that are common to the operations that connect. We call, the subset of operations in the DDRG that have E edges sinking into a specific node v, Data-Dependence-Predecessors (DDP) for that node. Furthermore, we call the subset of operations in the DDRG that are connected with a node v via E_R edges, Data-Reuse Predecessors (DRP) for that node.

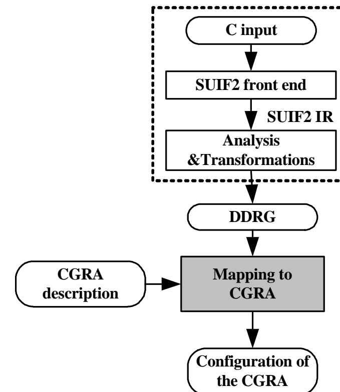

Fig. 2. Mapping methodology for CGRAs.

The description of the CGRA architecture is the second input to the mapping phase. The CGRA architecture is modeled by a undirected graph, called CGRA Graph, $G_A(V, E_I)$. The V is the set of PEs of the CGRA and E_I are the interconnections among them. The CGRA architecture description includes parameters, like the local RAMs' size inside a PE, the memory buses to which each PE is connected, the bus bandwidth and memory access times.

The task of mapping applications to coarse-grained reconfigurable architectures is a combination of scheduling operations for execution [14], mapping these operations to particular PEs, and routing data through specific interconnects in the CGRA. The PE selection for scheduling an operation, and the way the input operands are fetched to the specific PE, will be referred to hereafter as a Place Decision (PD) for that specific operation. Each PD has a different impact on the operation's execution time and on the execution of future scheduled operations. For this reason, a cost is assigned to each PD to incorporate the factors that influence the scheduling of the operations. The target of the proposed mapping algorithm is to find a cost-effective PD for each operation. The

proposed memory-conscious mapping algorithm is described in the pseudocode shown in Fig. 3.

```
// SOP : Set with operations to be scheduled
// G    : Application's DDRG
// QOP: Queue with ready to schedule operations
SOP = V;
AssignPriorities(G);
p = Minimum_Value_Of_Mobility; // Highest priority
while (SOP != o ){
 QOP = queue ROP(p);
 do {
  Op = dequeue QOP;
  (DDPPEs, RTime) = Predecessors(Op);
  (DRPPEs) = Predecessors_R(Op);
  do{
     Choices= GetCosts(DDPPEs,DRPPEs, RTime);
     RTime++;
  } while( Resource_Congestion(Choices) );
  Decision = DecideWhereToScheduleTimePlace(Choices);
  ReserveResources(Decision);
  Schedule(Op);
  SOP = SOP – Op;
 } while( QOP != o );
 p = p+1;
}
```

Fig. 3. Memory-aware mapping algorithm.

The algorithm is initialized by assigning to each DDRG node a value that represents its priority. The priority of an operation is calculated as the difference of its As Late As Possible (ALAP) minus its As Soon As Possible (ASAP) value [14]. This result is called mobility. Also variable p, which indirectly points each time to the most exigent operations, is initialized by the minimum value of mobility. In this way operations residing in the critical path are considered first in the scheduling phase. During the scheduling phase, in each iteration of the while loop, QOP queue takes via the ROP() function the ready to be executed operations which have a value of mobility less than or equal to the value of variable p. The first do-while loop schedules and routes each operation contained in the QOP queue one at a time, until it becomes empty. Then, the new ready to be executed operations are considered via ROP() function which updates the QOP queue.

The Predecessors() function returns (if exist) the PEs where the Op's DDP were scheduled (DDPPEs) and the earliest time (RTime) at which the operation Op can be scheduled. The RTime equals to the maximum of the times where each of the Op's DDP finished executing tf.

$$RTime(Op) = \max_{i=1,\ldots,|DDP(Op)|}\left(tf\left(Op_i\right)\right) \text{ where } Op_i \in DDP(Op) \quad (1)$$

The Predecessors_R() returns (if exist) for a given operation Op the place where its DRP were executed (DRPPEs). The function GetCosts() returns the possible PDs and the corresponding costs for the operation Op in the CGRA, in terms of the Choices variable. It takes as inputs the earliest possible schedule time (RTime) for the operation Op along with the PEs where the DDP (DDPPEs) and DRP (DRPPEs) have been scheduled. The function Resource_Congestion() returns true if there are no available PDs due to resources constraints. In that case RTime is incremented and the GetCosts() function is repeated until available PDs are found.

The DecideWhereToScheduleTimePlace() function analyzes the mapping costs from the Choices variable. The function firstly identifies the subset of PDs with minimum delay cost and from them the ones with minimum memory cost. Finally, from the resulting PD subset selects the one with minimum interconnection cost as the one which will be adopted. The function ReserveResources() reserves the resources (bus, PEs, storage and interconnections) for executing the current operation on the selected PE. Finally, the Schedule() records the scheduling of operation Op.

4.2 Mapping costs

The Choices variable includes the delay, interconnection, and the memory overhead costs. The delay cost for placing operation Op in a specific PE_x refers to the operation's earliest possible schedule time there. As shown in eq. (2), it is the sum of RTime plus the maximum of the times required to route the predecessor operands (DDP or DRP) to the PE_x. The PEs, where the predecessors have been scheduled, are denoted by PE_{Opi} where $i \in P(Op)$ and P(Op) stands for the predecessors of operation Op.

$$Choices.\text{dly}\left(PE_x, Op\right) = RTime\left(Op\right) + \max_{i=1,\ldots,|P(Op)|} tr(PE_{Op_i} \rightarrow PE_x) \quad (2)$$

The interconnection overhead refers to the interconnections that must be reserved in order to transfer the operands to the destination PE. As shown in eq. (3), it is the sum of the CGRA interconnections which were used to transfer the predecessor operands. Higher interconnection overhead causes future scheduled operations to have larger execution start time due to conflicts.

$$Choices.\text{intercon}\left(PE_x, Op\right) = \sum_{i \in P(Op)} PathLength\left(PE_{Op_i} \rightarrow PE_x\right) \quad (3)$$

A greedy approach was adopted for calculating the time (tr) (eq.(2)) and the number of interconnections (eq.(3)) required for routing an operand. For each operand the shortest paths, which connect the source and destination PE, are identified. From this set of paths, the one with the minimum routing delay is selected. The length and delay of the selected path gives the delay and interconnection costs through eq.2 and 3, respectively. Finally, the memory overhead cost

refers to the number of memory accesses by which the current operation contributes.

$$Choices.\text{memory}(PE_X, Op) = total_memory_accesses \quad (4)$$

5 Experimentation

In this section, we present the experimental results from applying the proposed mapping methodology steps on a representative CGRA architecture. We have developed in C++ a prototype compiler framework that realizes our mapping algorithm.

5.1 Experimental Setup

The experimental setup considers a 2D CGRA of 16 PEs connected in a 4x4 array. The PEs are directly connected to all other PEs in the same row and same column, as in a quadrant of Morphosys [3], through vertical and horizontal interconnections. Each PE has a local RAM of size 64 words; This size was selected after we have investigated by experimentation that this is the minimum size needed for not increasing the schedule length. There is one FU in each PE that can execute any operation in one clock cycle. The granularity of the FU is 16-bit, which is the word size. The direct connection delay among the PEs is zero cycles. Also, two buses per row are dedicated for transferring data to the PEs from the scratch-pad memory. Each bus transfers one word per memory cycle. The experiments consider scenarios for different values of the scratch-pad's latency.

We have used 13 characteristic DSP applications written in C code. Their characteristics are given in Table 1. More specifically, the second column refers to the number of operations in the application's loop body, the third one refers to the average number of Instructions Per Cycle (IPC) and the fourth one refers to the percentage of CGRA's PE utilization.

Table 1. Application's characteristics

Program	#of Ops	IPC	CGRA util%	Description
fft	95	5.6	34.8	radix-4 FFT
idct_ver	61	5.5	34.7	vertical pass (8X8 IDCT)
idct_hor	79	5.6	35.3	horizontal pass (8x8 IDCT)
fdct_ver	60	5.0	31.3	vertical pass (8x8 FDCT)
fdct_hor	61	5.1	31.8	horizontal pass (8x8 FDCT)
wav_ver	64	5.3	33.3	vertical pass (2D-DWT)
wav_hor	54	3.9	24.1	horizontal pass 2D-DWT
lat_synth	18	3.0	18.8	lattice synthesis filter
lat_anal	18	2.3	14.1	lattice analysis filter
iir	39	4.9	30.5	iir filter
volterra	27	3.9	24.1	iir filter
wdf7	44	4.9	30.6	iir filter
nc	56	4.3	26.9	noise canceling

5.2 Experimental Results

Fig. 4 shows the performance comparison for mapping the designs on the CGRA, with and without exploiting data reuse opportunities for reducing the memory accesses. Also, for this experiment, we unroll the loops in the designs by 10 and we consider 4 scenarios concerning the scratch pad's latency. Above the bars, the percentages of performance improvement are shown. In almost all cases, the designs are executed faster when data reuse opportunities are exploited.

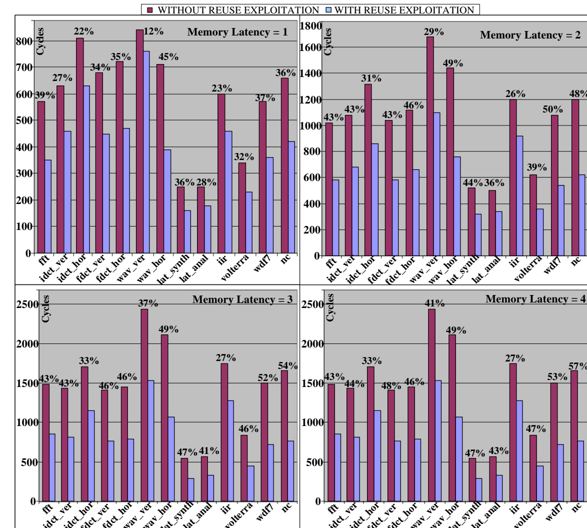

Fig. 4. Performance comparison with and without data reuse exploitation.

The performance improvements become larger as the memory latency value increases. The performance increase, which is greater as memory access latency increases, is due to the following reason. When the memory fetch cycle duration increases, there is a greater possibility for two operations which require memory access to conflict, thus reducing operation parallelism. This conflict causes operations to shift-down in time, thus further increasing the schedule length.

The reduction in memory accesses and the L0 utilization is illustrated in Table 2, for scratch-pad's latency equal to 1 and unroll factor equal to 10. The reduction in memory accesses explains the increase in performance, shown in Fig. 4. We free the memory bandwidth by reusing common data values internally in the CGRA and not fetching them from the scratch-pad memory. This increases the operation parallelism since more operations requiring memory accesses can be run in parallel. So, operations shift-up in time and the schedule length decreases.

For the same case Table 3 gives the CGRA utilization with and without taking into account reuse opportunities. As it can be concluded a significant enhancement in the operations parallelism, expressed by the CGRA utilization, is achieved when data reuse is exploited.

Table 2. Reduction in memory accesses and L0 utilization

Program	# of accesses		reduction %	L0 Utilization %
	No Reuse	Reuse		
fft	340	220	35.3	16.8
idct_ver	330	162	51.0	13.4
idct_hor	360	222	38.3	12.5
fdct_ver	300	150	50.0	15.4
fdct_hor	300	150	50.0	13.4
wav_ver	640	320	50.0	32.6
wav_hor	560	280	50.0	15.5
lat_synth	120	60	50.0	6.2
lat_anal	120	60	50.0	7.7
iir	450	330	26.7	17.0
volterra	200	90	55.0	9.3
wdf7	350	170	51.4	15.9
nc	400	162	59.5	17.0

Table 3. Improvement in CRGA utilization

Program	CGRA Utilization %		improvement%
	No Reuse	Reuse	
fft	42.8	69.6	62.6
idct_ver	60.5	82.9	37.0
idct_hor	61.0	78.4	28.5
fdct_ver	55.1	83.3	51.2
fdct_hor	53.0	81.1	53.0
wav_ver	47.6	52.6	10.5
wav_hor	47.5	86.5	82.1
lat_synth	45.0	70.3	56.2
lat_anal	45.0	62.5	38.9
iir	40.6	53.0	30.5
volterra	49.6	73.4	48.0
wdf7	48.2	76.4	58.5
nc	53.0	83.3	57.2

Fig.5 illustrates the L0 Utilization in respect to the unroll factor. As it is illustrated the L0 utilization is proportional to the unroll factor.

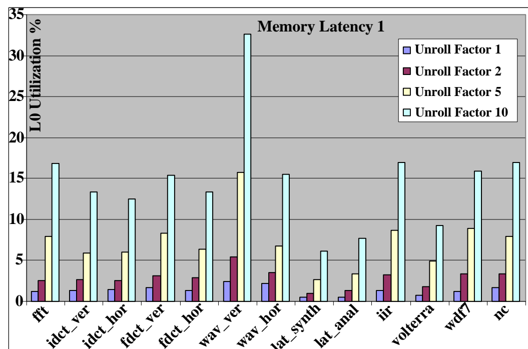

Fig. 5. L0 Utilization in respect to Unroll Factor.

The presented experiments show that a significant improvement in both performance and memory accesses is achieved with the proposed methodology. Additionally the performance improvents increase for larger values of the memory latency and the register file size.

6 CONCLUSIONS

A memory-conscious mapping for CGRA architectures was presented in this work. Significant improvements in performance have been achieved by exploiting the storage units inside the PEs. Additionally the impact that the architectural decisions have on the performance improvements achieved by our methodology have also been explored.

References:

[1] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective", in Proc. of ACM/IEEE DATE '01, pp. 642-649, 2001.

[2] T. Miyamori and K. Olukutun "REMARC: Reconfigurable Multimedia Array Coprocessor", in IEICE Trans. On Information and Systems, pp. 389-397, 1999.

[3] H. Singh, L. Ming-Hau, L. Guangming, F.J. Kurdahi, N. Bagherzadeh, E.M. Chaves Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Communication-Intensive Applications", in IEEE Trans. on Computers, vol. 49, no. 5, pp. 465-481, May 2000.

[4] Pact Corporation, "The XPP white Paper", Technical report, www.pactcorp.com, 2004.

[5] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. Achteren, and T. Omnes, "Data Accesses and Storage Management for Embedded Programmable Processors", Kluwer Academic Publishers, 2002.

[6] Reiner W. Hartenstein and Rainer Kress, "A datapath Synthesis System for the reconfigurable datapath architecture",ASP-DAC 05, Article No. 77

[7] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, "Exploiting Loop-Level Parallelism on Coarse-grained Reconfigurable Architectures Using Modulo Scheduling", in Proc. of ACM/IEEE DATE '03, pp. 255-261, 2003.

[8] B. Mei, et.al, "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture, A Case Study", in Proc. of ACM/IEEE DATE '04, pp. 1224-1229, 2004.

[9] B. Mei A. Lambrechts, et.al.,"Architecture Exploraiton for a Reconfigurable Architecture Template", IEEE Design and Test 2005 Vol.22, No.2 pp.90-101

[10] J. Lee and K. Choi, "Compilation Approach for Coarse-grained Reconfigurable Architectures", in IEEE Design & Test of Computers, vol. 20, no. 1, pp. 26-33, Jan.-Feb. 2003.

[11] P. R. Panda, N. Dutt, and A. Nicolau, "Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration", Kluwer Academic Publishers, 1999.

[12] K. Kennedy and R. Allen, "Optimizing Compilers for modern architectures", Morgan Kauffman Publishers, 2002.

[13] M. W. Hall et al., "Maximizing multiprocessor performance with the SUIF compiler", Computer, vol. 29, pp. 84-89, 1996.

[14] G. De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill, International Editions, 1994.

[15] Texas Instruments Inc., www.ti.com, 2004.