Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study

Bingfeng Mei[†‡], Serge Vernalde[†], Diederik Verkest[†‡⋆], Rudy Lauwereins[†‡]
[†]IMEC vzw, Kapeldreef 75, B-3001, Leuven, Belgium
[‡] Department of Electrical Engineering, Katholieke Universiteit Leuven, Leuven, Belgium
[⋆] Department of Electrical Engineering, Vrije Universiteit Brussel, Brussel, Belgium

Abstract

Coarse-grained reconfigurable architectures have seen growing importance recently. Design tools and methodology are essential to their success. Based on our previous work on modulo scheduling algorithms and a novel architecture with tightly coupled VLIW/reconfigurable matrix, we present a C-based design flow using an MPEG-2 decoder as a design example. The application is mapped to the architecture in less than one person-week starting from a software implementation. The kernel and overall speedup over the reference VLIW are 4.84 and 3.05 respectively. The case study shows that our methodology and architecture can deliver a competitive package in terms of design efforts and performance over other programmable architectures.

1 Introduction

Reconfigurable architectures have seen growing importance among both academic research and commercial applications in the past few years. Particularly, coarse-grained reconfigurable architectures gain much attention [1, 7, 8, 10, 11]. They often consist of tens to hundreds of functional units (FUs), which are capable of executing word- or subword-level operations instead of bit-level ones found in common FPGAs. This *coarse* granularity greatly reduces the delay, area, power and configuration time compared with FPGAs, however, at the expense of flexibility.

The target applications of these architectures, e.g., telecommunications and multimedia, often spend most time executing a few *time-critical code segments* with well-defined characteristics [12]. So the performance of a whole application may be improved considerably by mapping these critical segments, typically *loops*, on reconfigurable architectures. However, an application not only comprises regular kernels, but also a large part of control-intensive, irregular code which is difficult to map on reconfigurable architectures. Therefore, most reconfigurable systems consist of a reconfigurable matrix and a processor, typically a RISC, to execute the entire application. Designing for such systems is similar to a HW/SW co-design problem.

Although many architectures are proposed, few provide design methodology and tools which can both exploit high parallelism and deliver a software-like design experience. Many lack efficient tools to map the kernel to the reconfigurable matrix [7, 11], while some don't have good support for co-design [1, 8, 10]. In our previous work [6], we solved the problem of mapping a kernel to a family of reconfigurable architectures by a novel modulo scheduling algorithm. In other work [5], we proposed a new architecture, called ADRES (Architecture for Dynamic Reconfigurable Embedded Systems), which tightly couples a VLIW and a reconfigurable matrix, resulting in many advantages over common reconfigurable systems with loosely coupled RISC/reconfigurable matrix. In this paper, we demonstrate a C-based design flow taking full advantage of the scheduling algorithm and the ADRES features using an MPEG-2 decoder as an example. The case study shows our methodology can design an application with efforts comparable with software development while still achieving the high performance expected from reconfigurable architectures.

The paper is organized as follow. Section 2 gives an overview of the ADRES architecture. Section 3 explains the C-based design flow. Section 4 describes how we map the MPEG-2 decoder, including mapping results and some comparisons with the VLIW architecture. Section 5 discusses related work. Section 6 concludes the paper.

2 ADRES Architecture Overview

Fig. 2 describes the system view of the ADRES architecture. It is similar to a processor with an execution core connected to a memory hierarchy. The ADRES core (fig 1) consists of many basic components, e.g., FUs and register files (RF), which are connected in a certain topology. The whole ADRES core has two functional views: the VLIW processor and the reconfigurable matrix. The reconfigurable matrix is used to accelerate the dataflow-like kernels in a highly parallel way, whereas the VLIW executes the non-kernel code by exploiting instruction-level parallelism (ILP). These two functional views share some resources because their executions will never overlap with each other thanks to the processor/co-processor model.

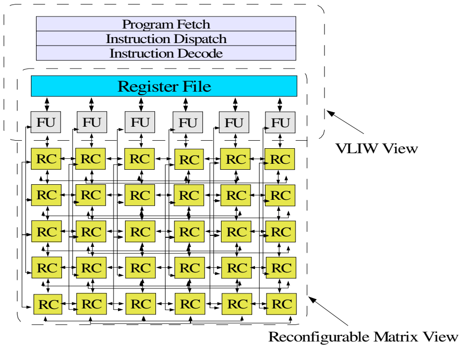


Figure 1. ADRES core

For the VLIW part, several FUs are allocated and connected together through one multi-port register file, which is typical for a VLIW architecture. The VLIW FUs are also connected to the memory hierarchy, depending on available ports. For the reconfigurable matrix, apart from the FUs and RF shared with the VLIW processor, there are a number of *reconfigurable cells* (RC) which basically comprise FUs and RFs too (fig. 3). The FUs can be heterogeneous supporting different operation sets. To remove the control flow inside loops, the FUs support predicated operations [6, 4]. The multiplexors are used to direct data from different sources. The configuration RAM stores a few configuration contexts locally, which can be loaded on cycle-by-cycle basis. The configuration contexts can also be loaded from the memory hierarchy at the cost of extra delay if the local configuration RAM is not big enough.

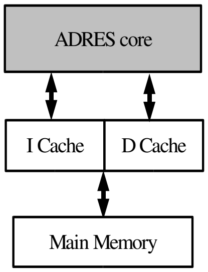


Figure 2. ADRES system

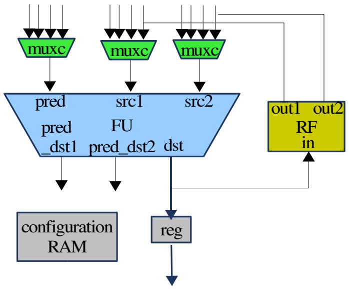


Figure 3. Reconfigurable Cell

The communication between the VLIW and the reconfigurable matrix happens through the shared RF, i.e., the VLIW RF, and the shared access to the memory hierarchy. In fact, ADRES is a template of an architecture instead of a fixed architecture. An XML-based architecture description language is used to define the overall topology, supported operation set, resource allocation, timing and even the internal organization of each RC.

Thanks to its tight integration, ADRES has many advantages. First, a system comprising a VLIW instead of a RISC found in other reconfigurable systems provides a way to accelerate the non-kernel code, which is often a bottleneck in many applications. Second, it greatly reduces both communication overhead and programming complexity through the shared RF and memory access between the VLIW and reconfigurable matrix. Finally, shared resources reduce costs considerably.

3 C-Based Design Flow

The design flow of the ADRES architecture is shown in fig. 4. A design starts from a C-language description of the application. The profiling/partitioning step identifies the candidate loops for mapping on the reconfigurable matrix based on the execution time and possible speed-up. Source-level transformations try to rewrite the kernel in order to make it pipelineable and to maximize the performance. Next, we use IMPACT, a compiler framework mainly for VLIW [2], to parse the C code and do some analysis and optimization. IMPACT emits an intermediate representation, called *Lcode*, which is used as the input for scheduling. On the right side, the target architecture is described in an XML-based language. Then the parser and abstraction steps transform the architecture to an internal graph representation [6]. Taking the program and architecture representations as input, the modulo scheduling algorithm is applied to achieve high parallelism for the kernels, whereas traditional ILP scheduling techniques are applied to gain moderate parallelism for the non-kernel code. The communication between these two parts is automatically identified and handled by our tools. Next, the tools generate scheduled code for both the reconfigurable matrix and the VLIW, which can be simulated by a co-simulator. If not all the kernels can be stored locally on the configuration RAM, we need to apply a kernel scheduling step, which is still ongoing research, to reduce the configuration overhead.

While many key steps are fully automated, some steps still need input from the designer to achieve best performance, particularly the partitioning and source-level transformation, where the most design efforts are spent on. The profiling is done on our co-simulator. The partitioning decision has to be made in the early phase. One reason is that the optimization requirements for the reconfigurable matrix and VLIW are different. For example, in order to have high performance, the VLIW compiler might selectively construct hyperblocks, whereas the optimization toward reconfigurable matrix requires aggressive hyperblock construction [6, 4]. We use two optimization settings for the IMPACT frontend, which does much analysis and optimization work. The source-level transformation is crucial to map more loops to the reconfigurable matrix. There are a number of restrictions for loop to be pipelineable. For example, only innermost loop can be pipelined and jumping out of the loop should occur at the end of a loop body. A "natural" source code might not have enough pipelineable loops. Several techniques can be used in constructing pipelineable loops such as function inlining and loop unrolling. The transformed code can be directly verified on the host computer or the VLIW without invoking the time-consuming modulo scheduling algorithm.

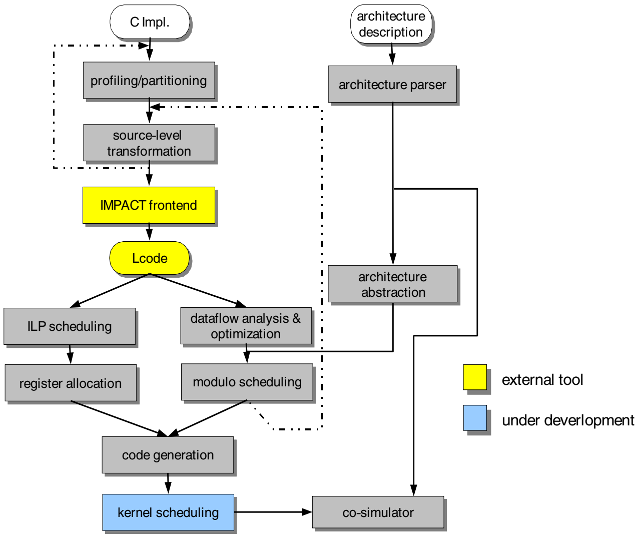


Figure 4. Design flow for ADRES

A centralized setting file controls the whole process of the compilation. Information obtained and decisions made during the design are annotated into the setting file. The setting file contains the control over both the global design environment and each individual loop, which enables us to focus on one loop at a time. This strategy reduces design time significantly (fig. 5).

When we deal with *loop1*, the transformed source code *loop1** is verified first on the VLIW, which is very fast. Next, the modulo scheduling is applied to *loop1**. Since our modulo scheduling algorithm is similar to a placement and routing algorithm, it is also time-consuming, typically from seconds to hours, and sensitive to initialization conditions, e.g., the random seed. Modulo scheduling tries to engineer a schedule with as small as possible II (initiation interval) [9]. A difference of 1 in II results in a big difference in the performance of the pipelined loop. Therefore, it is worth trying different random seeds if a lower II looks possible. After finding a low II, the parameters are annotated into the setting file. If the modulo scheduling can't find a good result, we may go back to the source-level transformation step for more refinement. At the next step, we can focus on *loop2* while mapping *loop1** to the VLIW temporarily. After all the loops are successfully mapped to the reconfigurable matrix, we can integrate all the pipelined loops together using the parameters preserved in the setting file. Using this strategy, we always focus on mapping one loop, and every loop is verified through the co-simulation. This feature is especially useful to parallelize the design effort. Basically, each person can work on one loop and get it verified in the co-simulation environment. Finally, multiple loops can be put together for integration. With the processor/co-processor execution model, no unexpected things will happen.

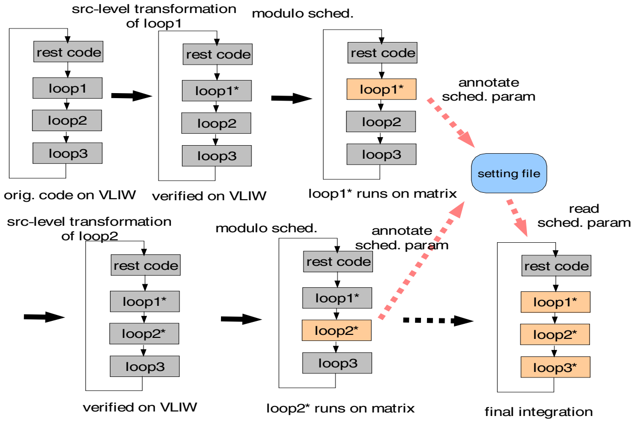


Figure 5. Focus on one loop at a time

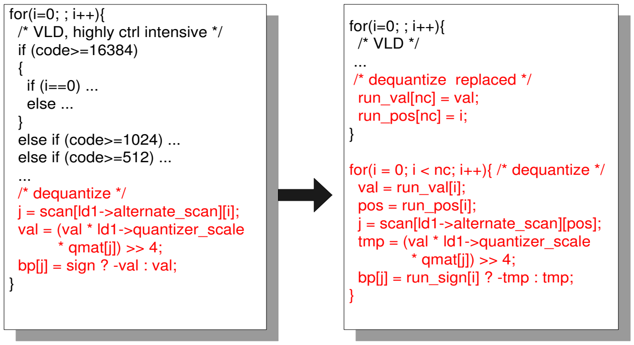
Thanks to the tight integration of the ADRES architecture, communication between the kernels and the non-kernel code can be handled by our compiler automatically with much less communication overhead compared to loosely coupled reconfigurable systems. The compiler only needs to identify live-in and live-out variables of the loop and assign them to the shared RF (VLIW RF). For communication through the memory space, we needn't do anything because the matrix and the VLIW share the memory access, which also eliminates the need for data copying.

4 Mapping an MPEG-2 Decoder Application

In this section, we use an MPEG-2 decoder as a design example to illustrate the C-based design flow for the ADRES architecture. The MPEG-2 decoder is a representative multimedia application. It requires very high computation power. Most execution time is spent on several kernels, which have high inherent parallelism. This means it is a good candidate for reconfigurable architectures. We use a C implementation from the MPEG Software Simulation Group (MSSG) as a starting point. The code is highly optimized for processors and a part of MediaBench [3]. The application comprises 21 files and around 10,000 lines.

4.1 Mapping to the ADRES Architecture

First, we identify 14 loops from the original applications as candidates for pipelining on the reconfigurable matrix by profiling the application. Among these loops, *form_comp_pred1* to *form_comp_pred8* are 8 conditional branches in the motion compensation. *idct1* and *idct2* are vertical and horizontal loops of the fast IDCT (inverse discrete cosine transformation). *add_block1* and *add_block2* are loops of summing blocks for intra and non-intra frames respectively. In addition, to improve the performance as much as possible, we extract two dequantization loops, i.e., *non_intra_dequant* and *intra_dequant*, from the VLD (variable length decoding) loops. The VLD itself is highly control-intensive and has strong cross-iteration dependency (fig. 6). Transforming the source code, we create pipelineable loops for dequantization at the expense of extra operations, but pipelining them on the reconfigurable matrix will pay off. All the 16 loops (tab. 1) account for 84.6% of the total execution time and only 3.3% of the total code size.



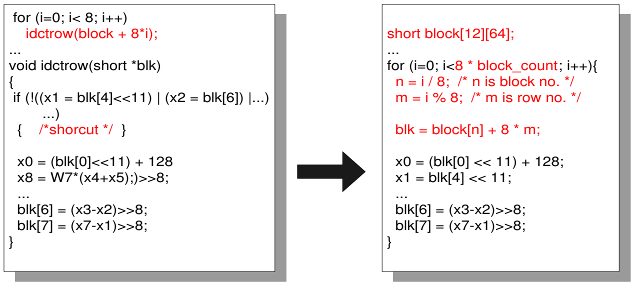
```
for(i=0; ; i++){
  /* VLD, highly ctrl intensive */
  if (code>=16384)
  {
    if (i==0) ...
    else ...
  }
  else if (code>=1024) ...
  else if (code>=512) ...
  ...
  /* dequantize */
  j = scan[ld1->alternate_scan][i];
  val = (val * ld1->quantizer_scale
           * qmat[j]) >> 4;
  bp[j] = sign ? -val : val;
}
```

```
for(i=0; ; i++){
  /* VLD */
 ...
 /* dequantize  replaced */
  run_val[nc] = val;
  run_pos[nc] = i;
}

for(i = 0; i < nc; i++){ /* dequantize */
  val = run_val[i];
  pos = run_pos[i];
  j = scan[ld1->alternate_scan][pos];
  tmp = (val * ld1->quantizer_scale
            * qmat[j]) >> 4;
  bp[j] = run_sign[i] ? -tmp : tmp;
}
```

Figure 6. Extract intra_dequant loop

Only a few of these loops can be immediately mapped to the reconfigurable matrix. For the others we have to perform source-level transformation. For example, in the original IDCT kernel (fig. 7), the inner loop body is in the form of a function, which is fine for a processor, but unpipelineable for a reconfigurable matrix. There is some shortcut computation, which reduces execution time on the processor but is highly irregular. Moreover, the *idct* loop is only applied to an 8x8 block at once, which means only 8 iterations. This is very costly in terms of prologue and epilogue in a pipelined loop. To map the *idct* to the reconfigurable matrix, we have to transform the code considerably. First, the loop body function *idctrow* is inlined. Second, all the shortcut computation is removed to make the loop more regular. Finally, the idct is applied to a macroblock, which usually has 6 8x8 blocks in MPEG-2 video. So the total number of iterations of each *idct* is increased to 48, greatly reducing the pipelining overhead. After all these transformations, we end up with two pipelineable loops in *idct* with high performance potential for pipelining. For other loops, more transformations such as loop unrolling, loop flattening, and variable replicating are applied.



```
 for (i=0; i< 8; i++)
   idctrow(block + 8*i);
...
void idctrow(short *blk)
{
 if (!((x1 = blk[4]<<11) | (x2 = blk[6]) |...)
        ...)
  {    /*shorcut */  }

  x0 = (blk[0]<<11) + 128
  x8 = W7*(x4+x5);)>>8;
  ...
  blk[6] = (x3-x2)>>8;
  blk[7] = (x7-x1)>>8;
}
```

```
short block[12][64];
...
for (i=0; i<8 * block_count; i++){
  n = i / 8;  /* n is block no. */
  m = i % 8;  /* m is row no. */

  blk = block[n] + 8 * m;

  x0 = (blk[0] << 11) + 128;
  x1 = blk[4] << 11;
  ...
  blk[6] = (x3-x2)>>8;
  blk[7] = (x7-x1)>>8;
}
```

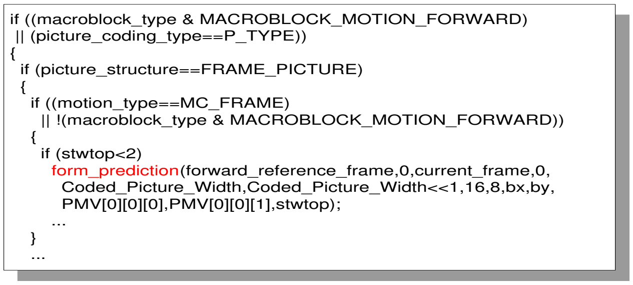
Figure 7. Transformation for idct1 loop

The above example shows that the required transformations are very diverse. It requires experience from the designer to figure out the appropriate ones. Therefore, it is unlikely to be automated in the near future. Fortunately, since we only need to focus on a few loops instead of the entire application, design efforts are limited to C-to-C rewriting for these loops while bearing the target matrix architecture in mind. In this particular case, it took the author 3 days to rewrite the C code to reflect all the required transformations.

We annotate the partitioning decision of these 16 loops into the setting file. Next, IMPACT is invoked to parse the source code, do analysis and optimization, where two sets of optimization parameters are applied toward the VLIW and the matrix respectively according to the information recorded in the setting file. Afterward, our compiler performs modulo scheduling and ILP scheduling. Although there are 16 loops to be scheduled and scheduling each loop take from seconds to half an hour (tab. 1). We only need to schedule one loop at a time, whereas the non-kernel code (including other loops) can be scheduled on the VLIW. In this way, we can rapidly explore different design choices for one loop without invoking time-consuming modulo scheduling for other loops.

After all the loops are successfully mapped, we obtained the configuration contexts required for each loop. The characteristics of the MPEG-2 decoder can help to reduce the configuration RAM requirements. In MPEG-2 video, there are three types of frames, I-, P- and B-Frame. Different frames call different kernel sequences. I-Frame uses only 6 loops, while P- and B-Frame call 10 and 14 loops respectively. A frame lasts normally more than 10ms, while loading configurations takes only μs. Hence, the kernels needed by a frame can be loaded before the frame starts without incurring much overhead. In MPEG-2, the maximal number of contexts required is constrained by B-Frame, which uses 29 contexts. In case the configuration RAM is not big enough, we should apply kernel scheduling techniques, still ongoing research, to minimize reconfiguration overhead.

In the application, we observed how the tight integration of ADRES plays an important role in reducing programming complexity and communication overhead. Taking *form_predictions* as an example (fig. 8), it has a very complex control structure to compute parameters for each *form_prediction* function call, which contains all 8 *form_comp_pred* loops. The code is very control-intensive and not a loop, therefore, it can't be mapped to the reconfigurable matrix. There is quite a lot of communication traffic between it and the *form_comp_pred* loops. ADRES provides an ideal solution for this kind of situation. The above code is mapped to the VLIW, while the *form_comp_pred* loops are mapped to the matrix. The scalar variables passed between them are identified and assigned to the shared RF by the compiler automatically. Moreover, thanks to the shared memory access, only the memory address, which itself is a scalar variable, is passed to the pipelined loops instead of copying the memory block. The VLIW and the reconfigurable matrix work together cooperatively.



```
if ((macroblock_type & MACROBLOCK_MOTION_FORWARD)
 || (picture_coding_type==P_TYPE))
{
  if (picture_structure==FRAME_PICTURE)
  {
    if ((motion_type==MC_FRAME)
      || !(macroblock_type & MACROBLOCK_MOTION_FORWARD))
    {
      if (stwtop<2)
        form_prediction(forward_reference_frame,0,current_frame,0,
          Coded_Picture_Width,Coded_Picture_Width<<1,16,8,bx,by,
          PMV[0][0][0],PMV[0][0][1],stwtop);
        ...
    }
    ...
```

Figure 8. A piece of form_predictions

4.2 Mapping Results

To do the experiment, an architecture resembling the topology of MorphoSys [11] is instantiated from the ADRES template. In this configuration, a total of 64 FUs are divided into four tiles, each of which consists of 4x4 FUs. Each FU is not only connected to the 4 nearest neighbor FUs, but also to all FUs within the same row or column in this tile. In addition, there are row buses and column buses across the matrix. The first row of FUs are also used by the VLIW processor and connected to a multi-port register file. Only the FUs in the first row are capable of executing memory operations, i.e., load/store operations.

The entire design took less than one person-week to finish starting from the software implementation. The main efforts are spent on identifying pipelineable loops and source-level transformations. The scheduling results for kernels are listed in tab. 1, while the simulation results for the entire application are listed in tab. 2, which are obtained from an in-house co-simulator. The second column of tab. 1 is the number of operations of the loopbody. Initiation interval (II) means a new iteration can start at every II cycle. Instruction-per-cycle (IPC) reflects the parallelism. Stages refer to total pipeline stages which have an impact on prologue/epilogue overhead. Scheduling time is the CPU time to compute the schedule on a Pentium 4 1.7GHz/Linux PC.

kernel	no. of ops	II	IPC	stages	sched. time (secs)
clear_block	8	1	8	3	0.05
form_comp_pred1	41	2	20.5	6	81
form_comp_pred2	13	1	13	6	4.6
form_comp_pred3	57	2	28.5	10	586
form_comp_pred4	33	2	16.5	5	30
form_comp_pred5	54	2	27	8	245
form_comp_pred6	30	2	15	5	42
form_comp_pred7	67	3	22.3	6	167
form_comp_pred8	43	2	21.5	6	132
saturate	78	3	26	10	1720
idct1	83	3	27.7	7	363
idct2	132	4	33	7	459
add_block1	48	2	24	7	73
add_block2	44	2	22	4	27
non_intra_dequant	20	1	20	12	53
intra_dequant	18	1	18	12	18

Table 1. Scheduling results for kernels

4.3 Comparison with VLIW Architecture

Since we view coarse-grained reconfigurable architecture as a promising alternative competing with other established programmable architectures, we compared our results to those on the VLIW, which is widely used in DSP and multimedia applications and has mature compiler support. It should be noted that commercial VLIWs, e.g., TI's 64x series, have many specific instructions such as SIMD to improve performance. But that normally requires handcrafted assembly code, and we can add similar instructions into ADRES too. To be fair, we compare ADRES to VLIW without using assembly code. IMPACT framework is used again here as both compiler and simulator to obtain results, where aggressive optimizations are enabled. The tested VLIW has the same configuration as the first row of the tested ADRES architecture. The results are shown in tab. 2. The test video stream is *mobl_015.m2v* with 352x240x450

	VLIW(IMPACT)	ADRES
total ops	2.92×10^9	5.31×10^9
total cycles	1.28×10^9	4.20×10^8
frames/sec	35.2	107.1
speed-up/kernels		4.84
speed-up/overall		3.05
IPC(excl. kernels)		2.71

Table 2. Comparison with VLIW architecture

frames, and frame rate is obtained by assuming both architectures run at 100MHz. Overall ADRES executes more operations because of the optimization techniques mentioned in section 4.1. However, ADRES manages to speed-up kernels by almost 5 times and entire application by 3 times over the reference VLIW. The IPC excluding the kernels is 2.71, which means we are able to discover some ILP for the non-kernel code.

5 Related Work

Many coarse-grained reconfigurable architectures have been proposed in the past, but design methodology and tools lag behind. MorphoSys [11] uses an assembly-like language to manually map kernels, largely because of its SIMD programming model. Moreover, due to its loosely coupled nature, the designer has to identify and translate data transfers between the RISC and the RC array to communication primitives, involving much data copying. PipeRench [10] features a very clever architecture for pipelining. A methodology is developed to pipeline loops automatically. However, the reconfigurable part communicates with the other part through two FIFOs, which means it suffers from the same problems as other loosely-coupled reconfigurable systems. RaPiD [1] supports a C-like language to program kernels. RaPiD-C is specialized for pipelining, which requires considerable designer's knowledge about the architecture and is not easy to integrate with ANSI C for complete application design. PACT [8] uses the NML language, essentially an assembly language, to model kernels. Automatic placement and routing tools are able to map the kernel to the PACT XPP. Recently, PACT started to build a loosely-coupled system, including an ARM7 processor and using the AMBA bus as communication channel. The co-design flow for a complete application is not established yet.

6 Conclusion and Future Work

Coarse-grained reconfigurable architectures have advantages over traditional FPGAs in terms of delay, area and power. One big issue facing them is how to map not only computation-intensive kernels but also an entire application. To compete with other programmable architectures like the VLIW, which has mature compiler support, the design tools and methodology should deliver both high performance and software-like design experience in order to be widely accepted in applications. We approach this problem by a combination of a compiler-friendly architecture, a modulo scheduling algorithm for mapping loops, and a C-based design flow. Here we present a case study of an MPEG-2 decoder to illustrate the design flow. The results show we can achieve satisfying results for the entire application in less than one person-week. Major efforts are spent on profiling/partitioning and source-level transformation.

The source-level transformations very much depend on the experience of the designer. We would like to formulate it in a way that an average software programmer can easily use these techniques. Other ongoing work is about kernel scheduling. We are developing techniques to minimize reconfiguration overhead in case the local configuration RAM is not big enough to contain all the kernels.

References

[1] C. Ebeling, D. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *Proc. of International Workshop on Field Programmable Logic and Applications*, 1996.

[2] The IMPACT group. http://www.crhc.uiuc.edu/impact.

[3] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[4] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, 1992.

[5] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field-Programmable Logic and Applications*, 2003.

[6] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. In *Proc. Design, Automation and Test in Europe (DATE)*, 2003.

[7] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. In *International Symposium on Field Programmable Gate Arrays (FPGA)*, 1998.

[8] *PACT XPP Technologies*, 2003. http://www.pactcorp.com.

[9] B. R. Rau. Iterative modulo scheduling. Technical report, Hewlett-Packard Lab: HPL-94-115, 1995.

[10] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *Proc. of IEEE Custom Integrated Circuits Conference*, 2002.

[11] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers*, 49(5):465–481, May 2000.

[12] T. Wolf and M. Franklin. CommBench - a telecommunication benchmark for network processors. In *IEEE intl. Symp. on Performance Analysis of Systems and Software*, 2000.