Master Thesis
Treball Final de Carrera

PERFORMANCE EVALUATION OF MPEG-4 VIDEO ENCODER ON ADRES

Enric Mumbrú Bassas

ENGINYERIA D'ORGANITZACIÓ INDUSTRIAL

Director: Moisès Serra Serra
Daily supervisor: Eric Delfosse

Acknowledgements

I want to say thanks to David because he convinced me to go to Belgium and make this project. This give me the opportunity to work in one of most important research centers in microelectronics, to meet a new country and a new culture and the most important to meet new people, new friends. Thanks also to my family and friends, who support me in that important decision. Thanks also to my new friends here that I meet in Belgium for all the time spent. Finally thanks to Eric and David, again, because they helped me in my daily work.

Glossary

Acronyms and Abbreviations

ADRES	Architecture for Dynamically Reconfigurable Embedded Systems
CFG	Control-Flow Graph
CGRA	Coarse-Grained Reconfigurable Architecture
CPLD	Complex Programmable Logic Device
DCT	Discrete Cosine Transform
DLP	Data-Level Parallelism
DPCM	Differential Pulse Code Modulation
DRESC	Dynamically Reconfigurable Embedded System Compiler
FPGA	Field Programmable Gate Arrays
FU	Functional Unit
IDCT	Inverse Discrete Cosine Transform
II	Initiation Interval
ILP	Instruction Level Parallelism
IMEC	Interuniversitary MicroElectronics Center
IMPACT	Illinois Microarchitecture Project utilizing Advanced Compiler Technology
IPC	Instructions Per Cycle
IR	Intermediate Representation
MB	MacroBlock
MC	Motion Compensation
ME	Motion Estimation
MPEG	Motion Picture Expert Group
MRRG	Modulo Routing Resource Graph
MV	Motion Vector
PHP	Hypertext Preprocessor
P&R	Placement and Routing
RAM	Random Access Memory
RF	Register File
SA	Simulated Annealing
SAD	Sum of Absolute Differences
SD	Scheduling Density
SIMD	Single Instruction Multiple Data
SSA	Static Single Assignment
TC	Texture Coding
VLIW	Very Long Instruction Word

VOP	Video Object Plane
XML	eXtensible Markup Language

Contents

List of Tables

List of Tables

List of Graphics

Chapter 1
Introduction

Nowadays, a typical embedded system requires high performance to perform tasks such as video encoding/decoding at run-time. It should consume little energy to work hours or even days using a lightweight battery. It should be flexible enough to integrate multiple applications and standards in one single device. It has to be designed and verified in short time-to-market despite substantially increased complexity. The designers are struggling to meet these huge challenges, which call for innovations of both architectures and design methodology.

Coarse-grained reconfigurable architectures (CGRAs) are emerging as potential candidates to meet the above challenges. Many of them were proposed in recent years. These architectures often consist of tens to hundreds of functional units (FUs), which are capable of executing word-level operations instead of bit-level ones found in common Field Programmable Gate Arrays (FPGAs). This coarse granularity greatly reduces delay, area, power and configuration time compared with FPGAs. On the other hand, compared with traditional "coarse-grained" programmable processors, their massive computational resources enable them to achieve high parallelism and efficiency. However, existing CGRAs have yet been widely adopted mainly because of programming difficulty for such complex architecture.

ADRES is a novel CGRA designed by Interuniversity Micro-Electronics Center (IMEC). It tightly couples a very-long instruction word (VLIW) processor and a coarse-grained array by providing two functional views on the same physical resources. It brings advantages such as high performance, low communication overhead and easiness of programming. Finally, ADRES is a template instead of a concrete architecture. With the retargetable compilation support from DRESC (Dynamically Reconfigurable Embedded System Compile), architectural exploration becomes possible to discover better architectures or design domain-specific architectures.

In this thesis, a performance of an MPEG-4 encoder in ADRES is presented. The thesis shows the code evolution to obtain a good implementation for a given architecture. Additionally the main features of ADRES and its compiler (DRESC) are presented.

The thesis is organized as follows: firstly ADRES architecture is presented and compared with other current architectures. Then the principal characteristics of DRESC compiler, the design flowchart and some other necessary background are explained. The necessary requisites for mapping loops properly in the CGA are explained as well. Brief overviews of the MPEG-4 standard and MPEG-4 encoder are given. Finally the different code transformations, code issues and results are presented.

Chapter 2
ADRES an architecture template

2.1 Introduction

Coarse-grained reconfigurable architectures (CGRAs) [1] are emerging technology that has been deeply influenced by some existing architectures, including FPGAs (Field Programmable Gate Arrays) [2] and VLIW (Very Long Instruction Word) processors [3]. Because CGRAs are highly parallel architectures, they are also similar with other parallel computing architectures like vector processors. Moreover, the ADRES architecture combines features of both CGRA and VLIW processors and borrows many techniques from processor compilation. To fully understand ADRES and its compilation techniques, it is necessary know all the related areas. Figure 1 shows the relations between CGRAs and other architectures, as well as the position of the ADRES architecture in relation to these architectures. CGRAs partly originated from fine-grained reconfigurable architectures represented by FPGAs. Basically, at the top-level they look very similar. Both comprise an array of basic units, configurable logic blocks for FPGAs and functional units for CGRAs. Both are connected by reconfigurable routing networks. The functionality of a target application can be implemented by specializing both the basic units and the routing networks. Both are highly parallel architectures that enable exploitation of massive parallelism. The limited routing resources impose great design challenges on both architectures as well.

Much coarse-grained reconfigurable architecture (CGRA) has been developed in recent years. These architectures provide potential vehicles for future embedded system design. However, they still present many challenging issues, especially in how to support an efficient and automated design methodology. To attack these issues, the solution has to come from close interplay between both the architecture and design methodology developments.

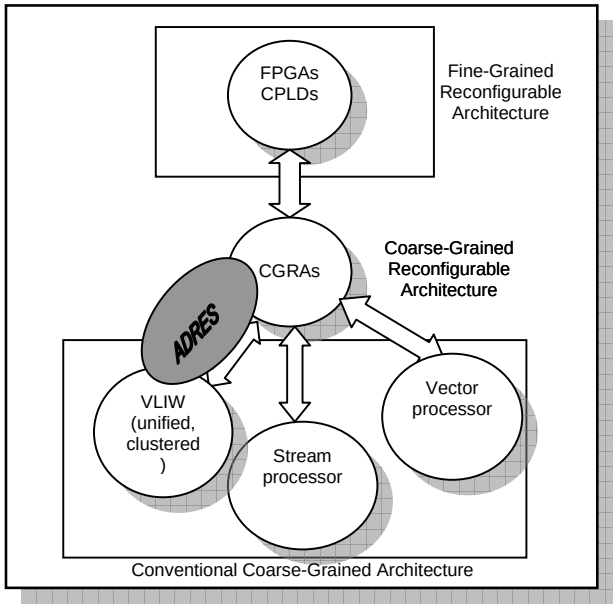


Figure 1 ADRES in relation to other architectures

2.2 Architecture Template Description

describes the system view of the ADRES architecture [4]. It is similar to a processor with an execution core attached to a memory hierarchy. Though architectural details at this moment are not well defined yet, we assume an ADRES array is connected to both data and instruction caches. At the next level, the caches are connected to a unified main memory. Though we assume two levels of memory hierarchy, more levels are possible, depending on application requirements.

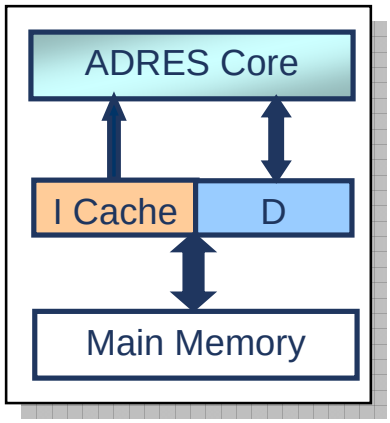


Figure 2: The ADRES system

Inside the ADRES array (Figure 3), we find many basic components, including computational resources, storage resources and routing resources. The computational resources are functional units (FUs), which are capable of executing a set of operations.

The storage resources mainly refer to the register file (RFs) and memory blocks, which can store intermediate data. Currently, only the RFs are supported by the compiler. The routing resources include wires, multiplexers and busses. Basically, computational resources and storage resources are connected by the routing resources in the ADRES array. This is similar to other CGRAs. The ADRES array is a flexible template instead of a concrete instance.

Figure 3 only shows one instance of the ADRES array with a topology resembling the MorphoSys architecture [8].

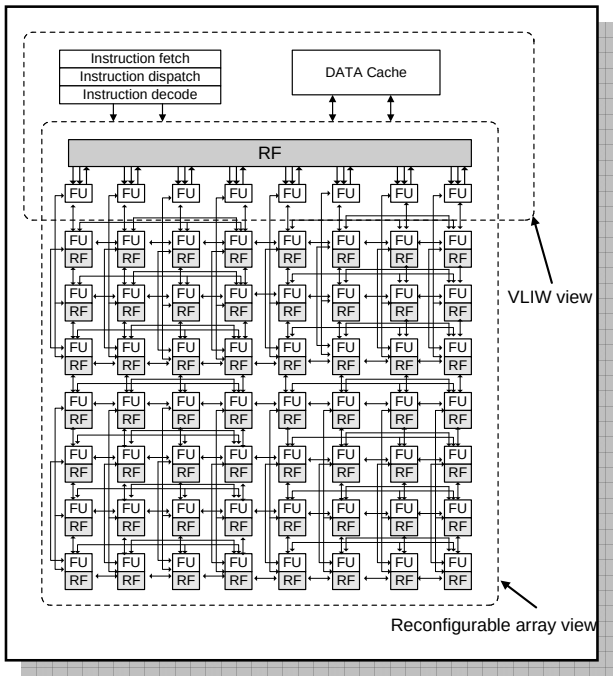


Figure 3 an instance of the ADRES array

Figure **4** shows an example of the detailed datapath. The FU performs coarse grained operations. To remove the control flow inside loops, the FU supports predicated operations. To guarantee timing, the outputs of FUs are required to be buffered by an output register. The results of the FU can be written to the RF, which is usually small and has fewer ports than the shared RF, or routed to other FUs. The multiplexers are used for routing data from different sources. The configuration Random Access Memory (RAM) provides bits to control these components. It stores a number of configuration contexts locally, which can be loaded on a cycle-by-cycle basis. The configurations can also be loaded from the memory hierarchy at the cost of extra delay if the local configuration RAM is not big enough.

Figure **4** shows only one possibility of how the datapath can be constructed. Very different instances are possible. For example, the output ports of a RF can be connected to input ports of several neighboring FUs. The ADRES template has much freedom to build an instance out of these basic components.

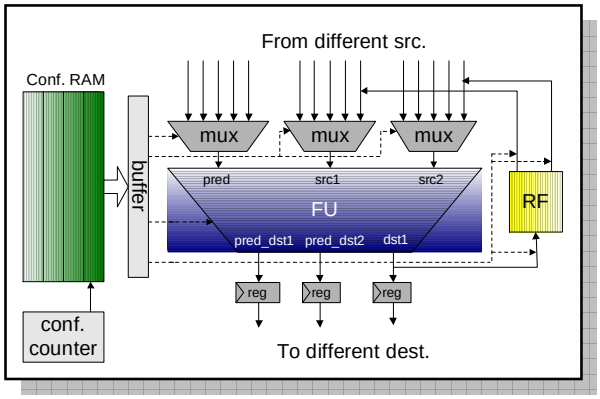


Figure 4 an example of detailed datapath

The most important feature of the ADRES architecture is the tight coupling between a VLIW processor and a coarse-grained reconfigurable array. Since the VLIW processor and CGRAs use similar components like FUs and RFs, a natural thought is to make them share those components though the FUs and RFs in the VLIW are typically more complex and powerful. The whole ADRES architecture has two virtual functional views: a VLIW processor and a reconfigurable array. These two virtual views share some physical resources because their executions will never overlap with each other thanks to the processor/coprocessor execution model. For the VLIW processor, several FUs are allocated and connected together through one multi-port register file. The FUs used in VLIW are generally more powerful. For example, some of them have to support the branch and subroutine call operations. Additionally, only these FUs are connected to the main memory hierarchy at this moment, depending on available ports. The instructions of the VLIW processor are loaded from the main instruction memory hierarchy. This requires typical steps like instruction fetching, dispatching and decoding. For the reconfigurable array part, all the resources, including the RF and FUs of the VLIW processor, form a big 2D array. The array is connected by partial routing resources. Dataflow like kernels are mapped to the array in a pipelined way to exploit high parallelism. The FUs and RFs of the array are simpler than those of the VLIW processor. The communication between these two virtual views is through the shared VLIW register file and memory access. The sharing is in the time dimension so that it does not increase the hardware cost. For example, it does not require more ports in the VLIW RF.

2.2.1 Execution and Configuration Model

There are two execution modes, VLIW mode and array mode, for the ADRES architecture. These two modes work mutually exclusive and take advantage of the tight coupling of the architecture. The VLIW mode executes the code that can not be

pipelined but mapped effectively in an ILP way, while the array mode executes kernels pipelined on the entire array. The control is transferred between these two modes by detecting entry and exit conditions of pipelined loops.
In the VLIW mode, the configuration is performed as in all other VLIW processors; in each cycle, an instruction is fetched and executed in each cycle from the instruction memory hierarchy. In the array mode, the configuration contexts are fetched from the on-chip configuration memory. Each kernel may use one or more consecutive contexts.

2.2.2 Functional Units

An FU can perform a set of operations. In ADRES, only fixed-point operations are supported because they are considered sufficient for typical telecommunication and multimedia applications. The instruction set used in ADRES is constrained by the compiler front-end, i.e., the IMPACT framework. All FUs are fully pipelined so that one instruction can be issued at each cycle even when the latency of that instruction is bigger than one cycle. Different implementations may lead to different latency, which can be specified in the architecture description and is supported by the compiler.
Unlike with most other CGRAs, predicated execution is introduced in the FUs in order to remove control-flow and do other transformations. Figure 5 shows the general picture of an FU. Basically, it has three source operands: pred, src1 and src2. pred is a 1-bit signal. If it is 1, the operation is executed; otherwise, the operation is nullified. src1 and src2 are normal data source operands.

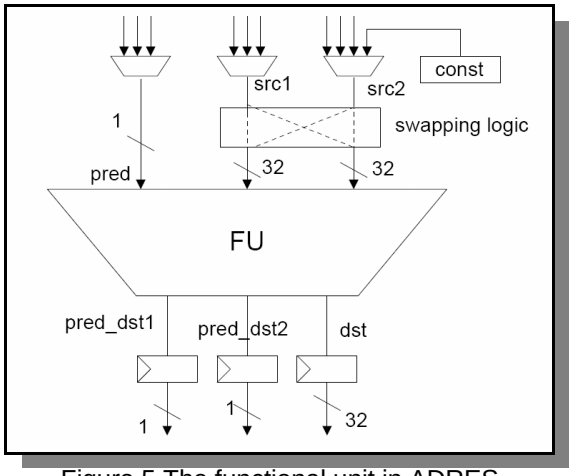


Figure 5 The functional unit in ADRES

Some operations may only use one of them. The operation set comprises several groups: arithmetic, logic, shift memory, comparison and operations that generate predicates.

2.2.3 Register Files

The register files (RFs) are used to store temporary data. There are two types of RFs: predicate and data RFs. The predicate RFs are 1-bit to store the predicate signal and the data RFs have the same data width as FUs. The modulo scheduling used for pipelining kernels imposes special requirements on the register file.
The modulo scheduling used for pipelining kernels imposes special requirements on the register le. In the pipelined loops, different iterations are overlapped. Therefore, the life-time of the same variable may overlap over different iterations. To accommodate this situation, each of the simultaneously live instances needs its own register. Furthermore, the name of the used register has to be clearly identified, either in software or in hardware.

2.2.4 Routing Networks

The routing networks consist of a data network and a predicate network. The data network routes the normal data among FUs and RFs, while the predicate network directs 1-bit predicate signals. These two networks do not necessarily have the same topology and can not overlap because of different data widths. Apart from its main purpose of handling control-flow, the predicate signal together with its routing network also serves for other purposes: eliminating prologue and epilogue; controlling the WE (write enable port) of the VLIW register file.

2.3 XML-Based Architecture Description Flow

To describe an architecture instance within the vast space of the ADRES template, it uses an architecture flow based on both Extensible Markup Language (XML) and Hypertext Preprocessor (PHP) languages (Figure 6).

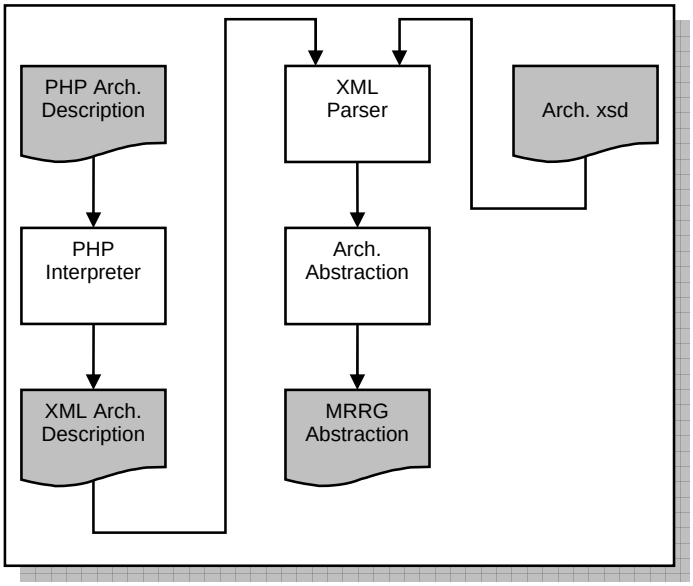


Figure 6 The architecture description flow of the ADRES template

XML is a simple, very flexible text format. It is designed to describe and deliver structured documents over the Internet. Unlike other markup languages such as HyperText Markup Language (HTML), its tags and semantics are not predefined. Therefore, XML is a kind of meta-language. A new language can be easily derived from XML by defining tags and structural relationships between them. Since XML is widely used in Internet context, several implementations of XML parsers are readily available in the form of open-source libraries. These libraries can be linked into the target applications and provide built-in parsing capability.

The overall architecture description comprises four sections: resource, connection, behavior and component. The resource section allocates a number of resources of different types.

The resources include FUs, RFs and TRNs (transitory nodes). For FU, the names of input and output ports, data width and supported operation groups can be specified. For FU, the names of input and output ports, data width and supported operation groups can be specified. The operation groups themselves are defined in the behavior section. RFs are specified in a similar way.

The connection section defines the topology of an ADRES instance. The behavior section defines some other architectural properties. For example, it specifies which RF is used as the one of the VLIW processor, how operations are grouped and the latency of each operation group. The area models of other components like FUs and RFs are integrated into the resource section for implementation convenience. Finally, the component section currently specifies area models of multiplexers so that the DRESC

framework can quickly estimate the area required for the interconnection for a given ADRES instance.

2.4 ADRES compiler: DRESC

2.4.1 The structure of DRESC compiler

Figure 7 shows the overall structure of DRESC compiler [5]. DRESC is supported on IMPACT compiler framework [6] as a front-end to parse C source code, do some optimization and analysis, construct the required hyperblock [[11], and emit the intermediate representation (IR), which is called *lcode*. Moreover, IMPACT is also used as a library in DRESC implementation to parse *lcode*, on the basis of which DRESC's own internal representation is constructed.

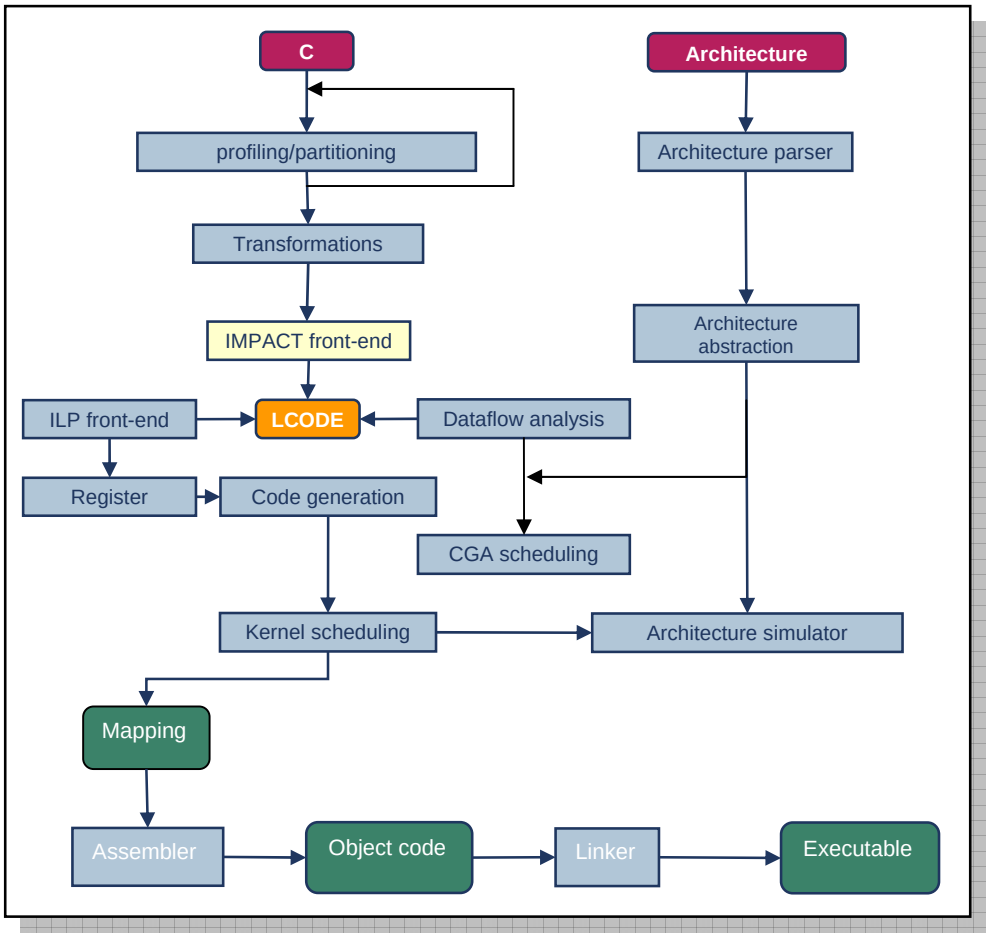


Figure 7 structure of DRESC compiler

Taking *Icode* as input, various analysis passes are executed to obtain necessary information for later transformation and scheduling, for instance, pipelinable loops are identified and predicate-sensitive dataflow analysis is performed to construct a data dependency graph (DDG). Next, a number of program transformations are performed to build a scheduling-ready pure dataflow used by the scheduling phase. Since the target reconfigurable architectures are different from traditional processors, some new techniques are developed, while others are mostly borrowed from VLIW compilation domain. In the right-hand side of Figure 7, the architecture description and abstraction path is shown. An architecture parser translates the description to an internal architecture description format. From this internal format, an architecture abstraction step generates a modulo routing resource graph (MRRG) which is used by the modulo scheduling algorithm. The modulo scheduling algorithm plays a central role in the DRESC compiler because the major strength of coarse-grained reconfigurable architectures is in loop-level parallelism. At this point, both program and architecture are represented in the forms of graphs. The task of modulo scheduling is to map the program graph to the architecture graph and try to achieve optimal performance while respecting all dependencies. After that, the scheduled code is fed to a simulator.

2.4.2 Program Analysis and Transformation

The array part of the ADRES architecture relies on loop pipelining to achieve high parallelism. Consequently, the techniques to extract and prepare properly loops are essential to the whole DRESC compiler framework. The flow of these steps is shown in Figure 8.

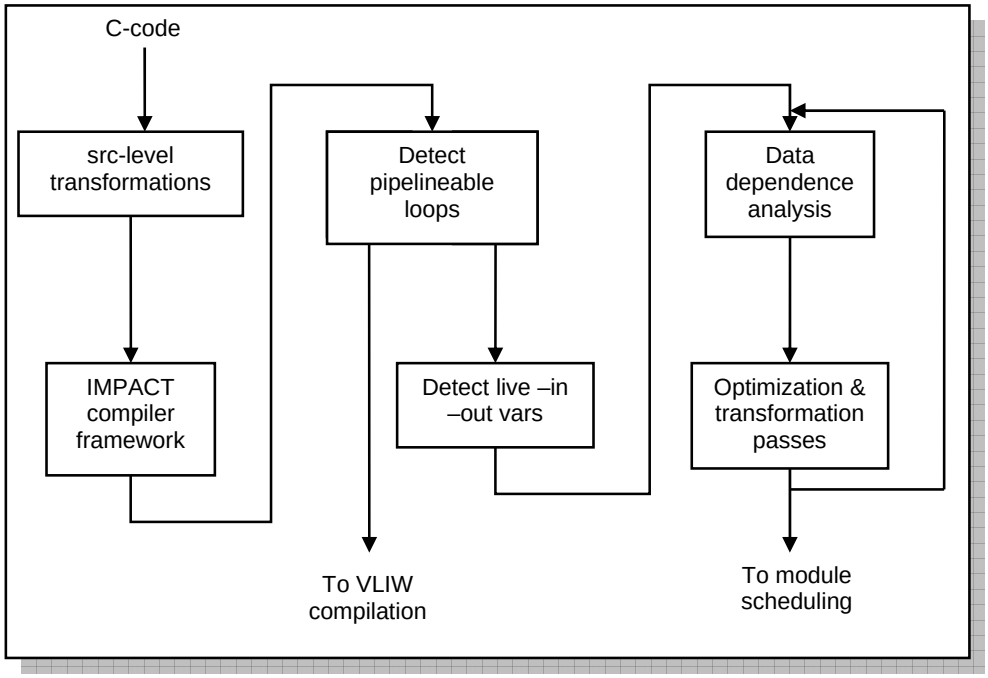


Figure 8 The flow of transformation, analysis and optimization steps

First, source-level transformations are applied to the application, currently based on manual C-to-C rewriting. This step tries to prepare pipelineable loops since the original source code may not be written in a pipelining-friendly way. Next the IMPACT compiler framework parses the C code, does its own analysis and optimization steps, which are designed for VLIW compilation, and emits an assembly-based intermediate representation, Lcode. Taking Lcode as input, the data-flow analysis implemented in DRESC includes detection of pipelineable loops, detection of live-in and live-out variables, and data dependence analysis. These steps generate data dependence graphs (DDG) representing the loop bodies and relevant information. However, they still cannot be directly scheduled on the ADRES architecture. For example, to reduce configuration overhead the prologue and epilogue should be properly handled in order to obtain the kernel-only code. Hence, some new optimization and transformation techniques are developed. Each transformation or optimization pass is followed repeatedly by the data dependence analysis step to update the DDG. Finally, the analyzed and optimized DDGs are fed to the modulo scheduler to exploit parallelism. The principal steps are:

- Identifying Pipelinable Loops.
- Data Dependence Graph Construction.
- Normalized Static Single Assignment Form (SSA).
- Live-in and Live-out Analysis
- Removing Explicit Prologue and Epilogue.

- Calculation of Minimum Initiation Interval (MII).
- Operation Ordering.

2.4.3 Modulo scheduling

Modulo scheduling is one of many software pipelining techniques [7]. Its objective is to engineer a schedule for one iteration of the loop such that this same schedule is repeated at regular intervals with respect to intra- or inter-iteration dependency and resource constraints. This interval is termed the *initiation interval* (II), essentially reflecting the performance of the scheduled loop. Various effective heuristics have been developed to attack this problem for both unified and clustered VLIW. However, they can't be applied to the case of a coarse-grained reconfigurable matrix because the nature of the problem becomes more difficult.

For unified VLIW, scheduling means to decide *when* to place operation. For clustered VLIW, we also have to decide *where* to assign the operation, this is a placement problem. For coarse-grained reconfigurable architecture, there is one additional task: determining *how* to connect placed operations. This is essentially a routing problem. If we view time as one dimension of P&R space, the scheduling can be defined as a P&R problem in 3D space, where routing resources are asymmetric and modulo constraints are applied.

This scheduling problem is more complex, especially if the nature of P&R space and scarce routing resources are considered. In FPGA's P&R algorithms, it is easy to run the placement algorithm first by minimizing a good cost function that measures the quality of placement. After the minimal cost is reached, the scheduling algorithm connects placed nodes. The coupling between these two sub-problems is very loose. In ADRES, it is difficult to separate placement and routing as two independent problems. It is almost impossible to find a placement algorithm and cost function which cans force the routability during the routing phase. The solution applied in ADRES is to solve these two sub-problems in one framework.

The algorithm is described in Figure 9. Like other modulo scheduling algorithms, the outermost loop tries successively larger II, starting with an initial value equal to MII, until the loop has been scheduled. For each II, it first generates an initial schedule which respects dependency constraints, but may overuse resources. For example, more than one operation may be scheduled on one FU at the same cycle.

In the inner loop, the algorithm iteratively reduces resource overuse and tries to come up with a legal schedule. At every iteration, an operation is ripped up from the existing schedule, and is placed randomly. The connected nets are rerouted accordingly. Then a cost function is computed to evaluate the new placement and routing. A *simulated annealing* strategy is used to decide whether we accept the new placement or not. If the

new cost is smaller than the old one, the new P&R of this operation will be accepted. Even if the new cost is bigger, there is still a chance to accept the move, depending on *"temperature".* This method helps to escape from local minimum.

The temperature is gradually decreased from a high value. So the operation is increasingly difficult to move. The cost function is constructed by taking account into overused resources. The penalty associated with them is increased every iteration. In this way, placer and router would try to find alternatives to avoid congestion. This idea is borrowed from the *Pathfinder* algorithm. In the end, if the algorithm runs out of time budget without finding a valid schedule, it starts with the next II. This algorithm is time-consuming. It takes minutes to schedule a loop of medium size.

```
II := MII;

while not scheduled do
   InitMrrg(II);
    InitTemperature();
    InitPlaceAndRoute();

  while not scheduled do
     for each op in sorted operation list
        RipUpOp();

        for i := 1 to random_pos_to_try do
           pos := GenRandomPos();
           success := PlaceAndRouteOp(pos);

           if success then
             new_cost := ComputeCost(op);
             accepted := EvaluateNewPos();
             if accepted then
               break;
             else
               continue;
             endif
           endfor

           if not accepted then
             RestoreOp();
           else
             CommitOp();

            if get a valid schedule then
              return scheduled;
         endfor

         if run out of time budget then
           break;

         UpdateOverusePenalty();
         UpdateTemperature();

  endwhile
  II++;
endwhile
```

Figure 9 The modulo scheduling algorithm core

Chapter 3
Source-Level Transformations

Since applications written in C language are often intended for software implementation, their loops may not be appropriate for pipelining. Therefore, some source-level transformations techniques are needed to prepare proper loops for mapping on the ADRES array [4]. Generally, there are two types of transformations. One type is to remove constraints for pipelining because a loop is pipelineable only if it meets some strict requirements. For example, it cannot contain function calls inside the loop body and cannot jump out in the middle of the loop body. To meet these requirements, techniques such as function inlining have to be applied to make code pipelineable. The other type of transformations helps to improve performance because a loop may not produce good performance in its original form though it is pipelineable. For example, if there are too few iterations in the loop, loop coalescing can be used to increase total iterations by combining nested loops.

3.1 Constraint-Removing Transformations

A loop is pipelineable only if it meets the following conditions:

- The loop body does not contain control-flow unless if-conversion can be applied.
- The loop body does not contain function calls.
- The loop body does not contain exit points other than the one at the end of the loop body.

Removing Control-flow and Early Exit: The control-flow and the multiple exit points are partly addressed by aggressive hyperblock formation. Nonetheless, the automatic compilation technique sometimes cannot figure out how to deal with control-flow that requires application-specific knowledge. Figure 10 shows an IDCT (inverse discrete cosine transform) example from the MPEG-2 decoder. The original source code is optimized for a traditional processor. There is a piece of shortcut code that tries to identify a special case and jump out the loop early. This piece of code is not pipelineable on the ADRES array. Therefore, it is removed by source-level transformation.

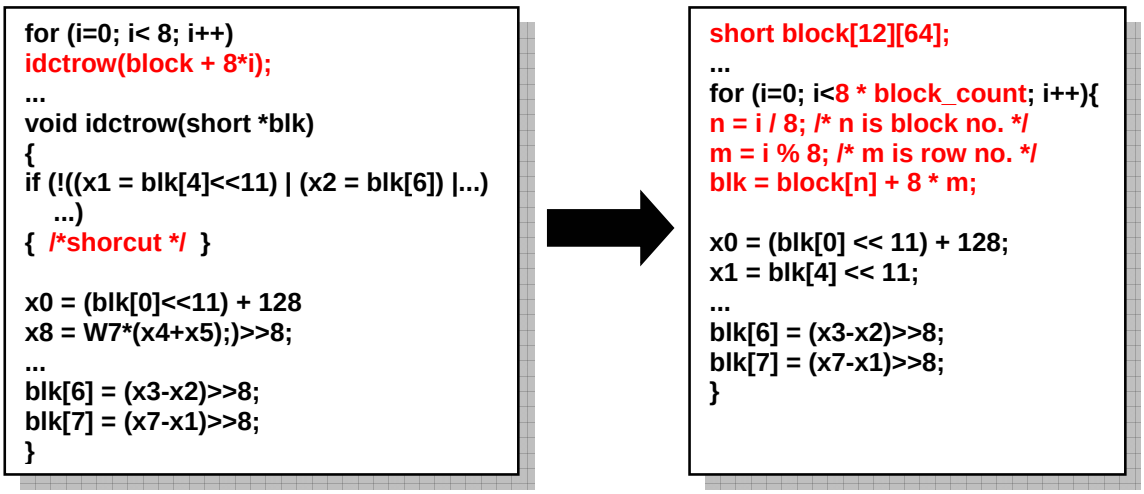


Figure 10 Transformations for an IDCT loop

Function inlining is a widely used optimization technique to reduce the overhead associated with function calls. However, it comes at the expense of increased code size if the inlined function is called in multiple places. Applied to the ADRES architecture, its primary purpose is to make loop pipelining feasible because the function call is not allowed inside a pipelineable loop. In Figure 10, the original loop includes a function idctrow, which performs 1D-IDCT on a row of pixels. After transformation, this function is inlined in the loop body to enable the pipelining.

3.2 Performance-Enhancing Transformations

Loop Coalescing: Currently the DRESC compiler can only pipeline the innermost loop of a nested loop. If the outer loops contribute to a significant portion of total execution time, or the total number of iterations of the innermost loops is too small so that the overhead of prologue and epilogue is dominant, only pipelining the innermost loops won't produce good performance according to Amdahl's law. One technique that helps to solve this problem is loop coalescing. Coalescing combines a loop nest into a single loop, with the original indices computed from the resulting single induction variable. Figure 11 describes an example. This transformation is originally developed for multiprocessor-based parallel computing. It can effectively increase the significance of the innermost loop though at the cost of recomputing the indices. Normally, coalescing two innermost loops should be sufficient to form a significant innermost loop for pipelining while still keeping the overhead low.

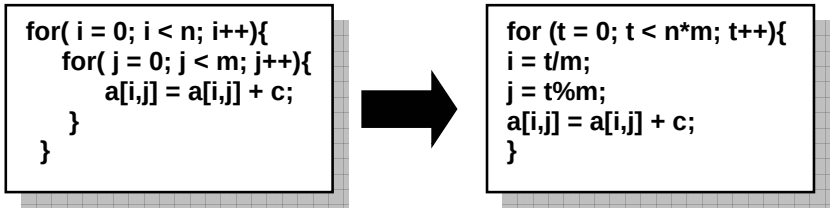


Figure 11 Example of loop coalescing

Loop Unrolling: The ADRES array is much bigger than other processor architectures. When a loop is mapped to the ADRES array, the loop body should be big enough to efficiently utilize the resources. Considering a loop consisting of only 10 operations, when it is mapped to an 8x8 array the utilization is at most 15.6% (10/64) and instruction-per-cycle is at most 10 if the loop is fully pipelined. One solution to this problem is known as loop unrolling. Generally, loop unrolling is the process of expanding a loop so each new iteration contains several copies of a single iteration. It also adjusts loop termination code and eliminates redundant branch instructions. Traditionally, it is an optimization technique that can reduce the total number of instructions executed by a processor and can increase instruction-level parallelism. Applied to the ADRES architecture, it also helps to increase the size of loop bodies so that pipelining is more efficient on a big array. Figure 12 shows one example. The original loop body is very small and cannot efficiently utilize an ADRES array. After unrolling the loop 4 times, the loop body contains about 4 times more static operations (the number of operations in the compiled code) than the original loop body, while the total number of dynamic operations (the number of operations actually executed) is about the same. Both the original loop and the transformed one can be fully pipelined to achieve 1 cycle/per iteration. Therefore the performance is increased 4 times after loop unrolling.

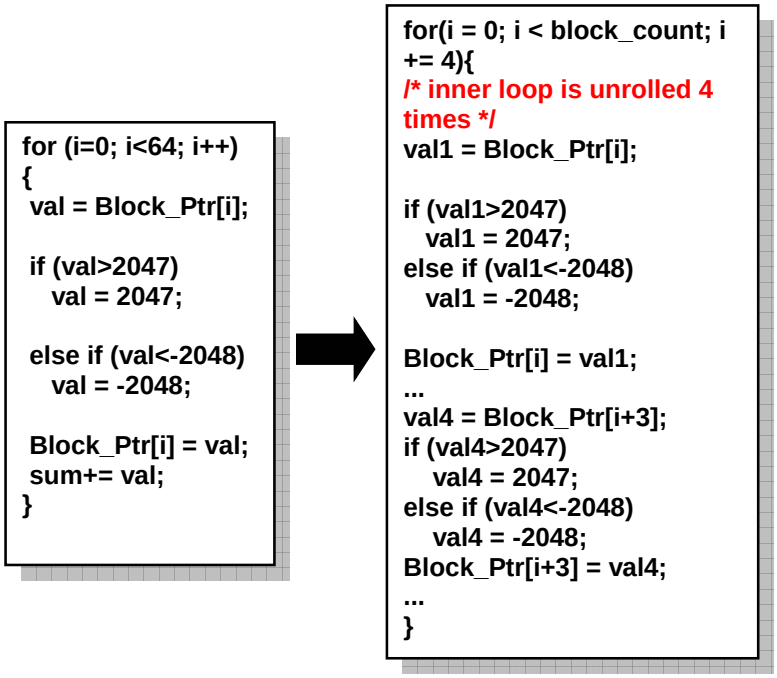


Figure 12 Example of loop unrolling

Loop unrolling involves a trade-off between the loop body size and the total number of iterations. If the loop is unrolled too many times, the total number of operations will increase undesirably while the performance doesn't increase accordingly. A bigger loop body requires more configuration contexts. At the same time, the total number of iterations will shrink so that the overhead of prologue and epilogue becomes prominent.

Increasing Iterations: Loop coalescing is one way to increase the total number of iterations in order to reduce the prologue and epilogue overhead in nested loops. In fact, increasing the total number of iterations can be very application-specific. If the designer understands the application better, more opportunities may be discovered to increase total number of iterations of a loop. Figure 10 also shows one example of this transformation applied to IDCT. The original loop is based on a basic block (8x8) consisting of only 8 iterations. After transformation, the IDCT loop is performed on a macroblock, which usually contains a number of basic blocks (shown as block count). Hence, the total number of iterations increase to 8 x block count so that the prologue and epilogue overhead is greatly reduced.

Tree Height Reduction: in a pipelined loop, the schedule length determines the total number of pipeline stages. A higher number of pipeline stages has a negative impact on performance due to the increased prologue and epilogue overhead. Sometimes the schedule length can be reduced by a technique known as tree height reduction. Figure 13 describes a simple example. Before and after the transformation the tree heights are

7 and 3 cycles respectively. Though some algorithm was developed to automatically perform this transformation, it can be easily done at source-level by the developer.

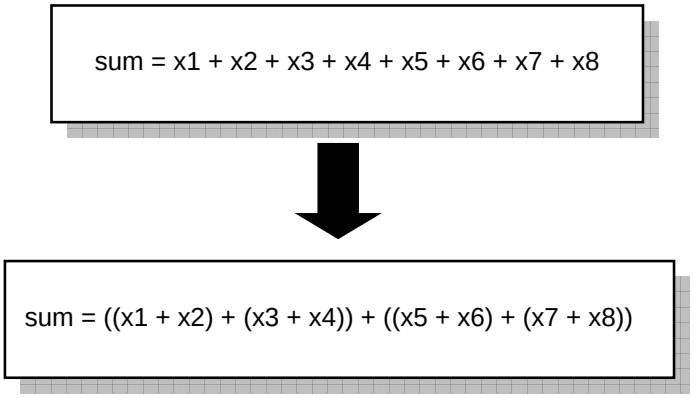


Figure 13 Example of tree height reduction

Reducing Memory Access: though the ADRES array has abundant computing resources, it still has resource bottlenecks, especially related to the memory bandwidth. Pipelining usually demands very high memory bandwidth. Consider for example an iteration that contains 10 memory accesses (both reads and writes) and an ADRES instance that has 2 memory ports. In this case, each iteration requires at least 5 cycles to meet the memory bandwidth constraint. For many kernels the memory access bandwidth can be reduced by replacing array variables with scalar ones. In normal processors, the scalar variables are stored in the register file. If there are too many scalar variables, they have to be spilled to the memory in which case the memory access is not eliminated. In the ADRES architecture, the scalar variables are stored in register files distributed throughout the array. These RFs are cheap to use and don't introduce a resource bottleneck for memory access. Normally after transform the code, the memory accesses decrease at the expense of an increased amount of operations. With abundant FUs available in the ADRES array, it is worth to trade memory accesses for more operations if the memory bandwidth is a bottleneck.

3.3 Guidelines for Source-Level Transformations

The required source-level transformations are very diverse and sometimes application-specific. In many cases, several transformations are performed together on one loop. So in this section are given some examples to illustrate the transformations instead of defining formal algorithms for the transformations. While some transformations such as function inlining and tree height reduction are good candidates for future automation, it is very difficult to automate other transformations, especially the application-specific ones.

Currently they are performed manually by the application designer. Nonetheless, it doesn't require a lot of effort to modify a number of loops at source-level for typical applications.

- Remove obstacles of pipelining for the candidate loop.
- Make the candidate loop significant in terms of execution time.
- Transform the candidate loop to have sufficient iterations.
- Balance resource utilization and total number of configuration contexts.
- Reduce memory access if the memory bandwidth is a bottleneck.
- Reduce total number of pipeline stages to minimize prologue/epilogue overhead.

Chapter 4
MPEG-4

As it has already been said, in order to evaluate the capabilities of ADRES an MPEG-4 encoder has been chosen. This chapter tries to give a general overview of the MPEG-4 natural video coding. Some functionalities are described briefly as well as some basic principles of the standard. Additionally the importance of each part in cycles terms is showed. Further information of the specific parts of the MPEG-4 encoder will be explained more accurately in following chapters.

4.1 MPEG-4 standard

ISO/IEC Standard 14496 Part 2 (MPEG-4 Visual [[9,[10]) improves on the popular MPEG-2 standard both in terms of compression efficiency (better compression for the same visual quality) and flexibility (enabling a much wider range of applications). It achieves this in two main ways, by making use of more advanced compression algorithms and by providing an extensive set of 'tools' for coding and manipulating digital media. MPEG-4 Visual consists of a 'core' video encoder/decoder model together with a number of additional coding tools. The core model is based on the well-known hybrid DPCM/DCT (Differential Pulse Code Modulation/Discrete Cosine Transform) coding model (see description section) and the basic function of the core is extended by tools supporting (among other things) enhanced compression efficiency, reliable transmission, coding of separate shapes or 'objects' in a visual scene, mesh-based compression and animation of face or body models.
MPEG-4 Visual attempts to satisfy the requirements of a wide range of visual communication applications through a toolkit-based approach to coding of visual information. Some of the key features that distinguish MPEG-4 Visual from previous visual coding standards include:

- Efficient compression of progressive and interlaced 'natural' video sequences (compression of sequences of rectangular video frames). The core compression tools are based on the ITU-T H.263 standard and can out-perform MPEG-1 and MPEG-2 video compression. Optional additional tools further improve compression efficiency.
- Coding of video objects (irregular-shaped regions of a video scene). This is a new concept for standard-based video coding and enables (for example) independent coding of foreground and background objects in a video scene.

- Support for effective transmission over practical networks. Error resilience tools help a decoder to recover from transmission errors and maintain a successful video connection in an error-prone network environment and scalable coding tools can help to support flexible transmission at a range of coded bitrates.
- Coding of still 'texture' (image data). This means, for example, that still images can be coded and transmitted within the same framework as moving video sequences. Texture coding tools may also be useful in conjunction with animation-based rendering.
- Coding of animated visual objects such as 2D and 3D polygonal meshes, animated faces and animated human bodies.
- Coding for specialist applications such as 'studio' quality video. In this type of application, visual quality is perhaps more important than high compression.

MPEG-4 Visual provides its coding functions through a combination of *tools, objects* and *profiles*. A *tool* is a subset of coding functions to support a specific feature (for example, basic video coding, interlaced video, coding object shapes, etc.). An *object* is a video element (e.g. a sequence of rectangular frames, a sequence of arbitrary-shaped regions, a still image) that is coded using one or more tools. For example, a simple video object is coded using a limited subset of tools for rectangular video frame sequences, a core video object is coded using tools for arbitrarily-shaped objects and so on. A *profile* is a set of object types that a CODEC is expected to be capable of handling.

One of the key contributions of MPEG-4Visual is a move away from the 'traditional' view of a video sequence as being merely a collection of rectangular frames of video. Instead, MPEG-4 Visual treats a video sequence as a collection of one or more *video objects*. MPEG-4 Visual defines a video object as a flexible 'entity that a user is allowed to access (seek, browse) and manipulate (cut and paste)'. A video object (VO) is an area of the video scene that may occupy an arbitrarily-shaped region and may exist for an arbitrary length of time. An instance of a VO at a particular point in time is a *video object plane* (VOP).

Notwithstanding the potential flexibility offered by object-based coding, the most popular application of MPEG-4 Visual is to encode complete frames of video. The tools required to handle rectangular VOPs (typically complete video frames) are grouped together in the so-called *simple* profiles. The basic tools are similar to those adopted by previous video coding standards, DCT-based coding of macroblocks with motion compensated prediction. The Simple profile is based around the well-known hybrid DPCM/DCT model (it will be described in the description section) with some additional tools to improve coding efficiency and transmission efficiency. Because of the widespread popularity of Simple profile, enhanced profiles for rectangular VOPs have been developed. The input to an MPEG-4 Visual encoder and the output of a decoder is a video sequence in 4:2:0, 4:2:2 or 4:4:4 progressive or interlaced formats.

4.2 MPEG-4 encoder description

The major video coding standards released since the early 1990s have been based on the same generic design (or model) of a video CODEC that incorporates a motion estimation and compensation front end (sometimes described as DPCM), a transform stage and an entropy encoder. The model is often described as a hybrid DPCM/DCT CODEC. Any CODEC that is compatible with H.261, H.263, MPEG-1, MPEG-2, MPEG-4 Visual and H.264 has to implement a similar set of basic coding and decoding functions (although there are many differences of detail between the standards and between implementations). Figure 14 depicts a generic DPCM/DCT hybrid encoder. In the encoder, video frame *n* (F*n*) is processed to produce a coded (compressed) bitstream.

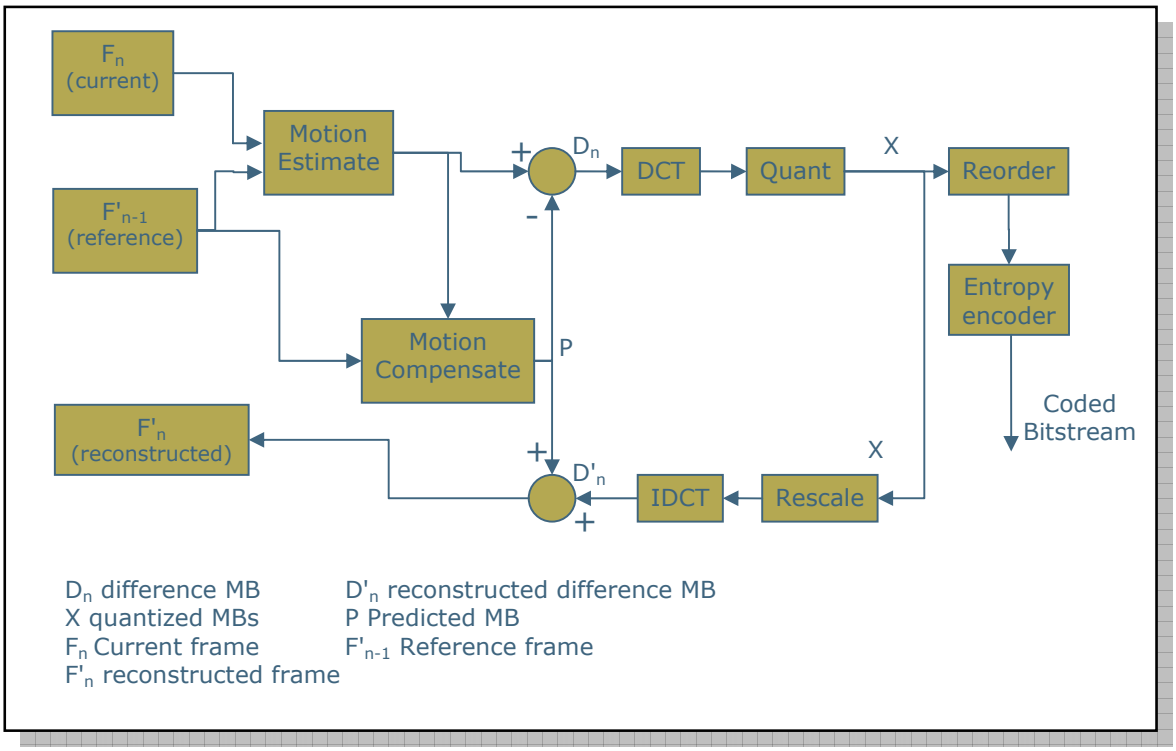


Figure 14 Encoder data flow

There are two main data flow paths in the encoder, left to right (encoding) and right to left (reconstruction). The encoding flow is as follows:

1. An input video frame Fn is presented for encoding and is processed in units of a macroblock (corresponding to a 16 × 16 luma region and associated chroma samples).
2. Fn is compared with a reference frame, for example the previous encoded frame (F'$_{n-1}$). A motion estimation function finds a 16 × 16 region in F'$_{n-1}$ (or a sub-sample interpolated version of F'$_{n-1}$) that 'matches' the current macroblock in Fn

(i.e. is similar according to some matching criteria). The offset between the current macroblock position and the chosen reference region is a motion vector (MV).

3. Based on the chosen MV, a motion compensated prediction P is generated (the 16 × 16 region selected by the motion estimator).
4. P is subtracted from the current macroblock to produce a residual or difference macroblock D.
5. D is transformed using the DCT. Sometimes, D is split into 8 × 8 or 4 × 4 sub-blocks and each sub-block is transformed separately.
6. Each sub-block is quantized (X).
7. The DCT coefficients of each sub-block are reordered and run-level coded.
8. Finally, the coefficients, motion vector and associated header information for each macroblock are entropy encoded to produce the compressed bitstream.

The reconstruction data flow is as follows:

1. Each quantized macroblock X is rescaled and inverse transformed to produce a decoded residual D'. Note that the nonreversible quantization process means that D' is not identical to D (i.e. distortion has been introduced).
2. The motion compensated prediction P is added to the residual D' to produce a reconstructed macroblock and the reconstructed macroblocks are saved to produce reconstructed frame F'_n. After encoding a complete frame, the reconstructed frame F'_n may be used as a reference frame for the next encoded frame F'_{n+1}

Notice that not all the frames follows this process because sometimes it is necessary to send to the decoder all the information frame and not only de difference with the reference frame. Frames that haven not reference frames are called INTRA frames and the others are called INTER frames.

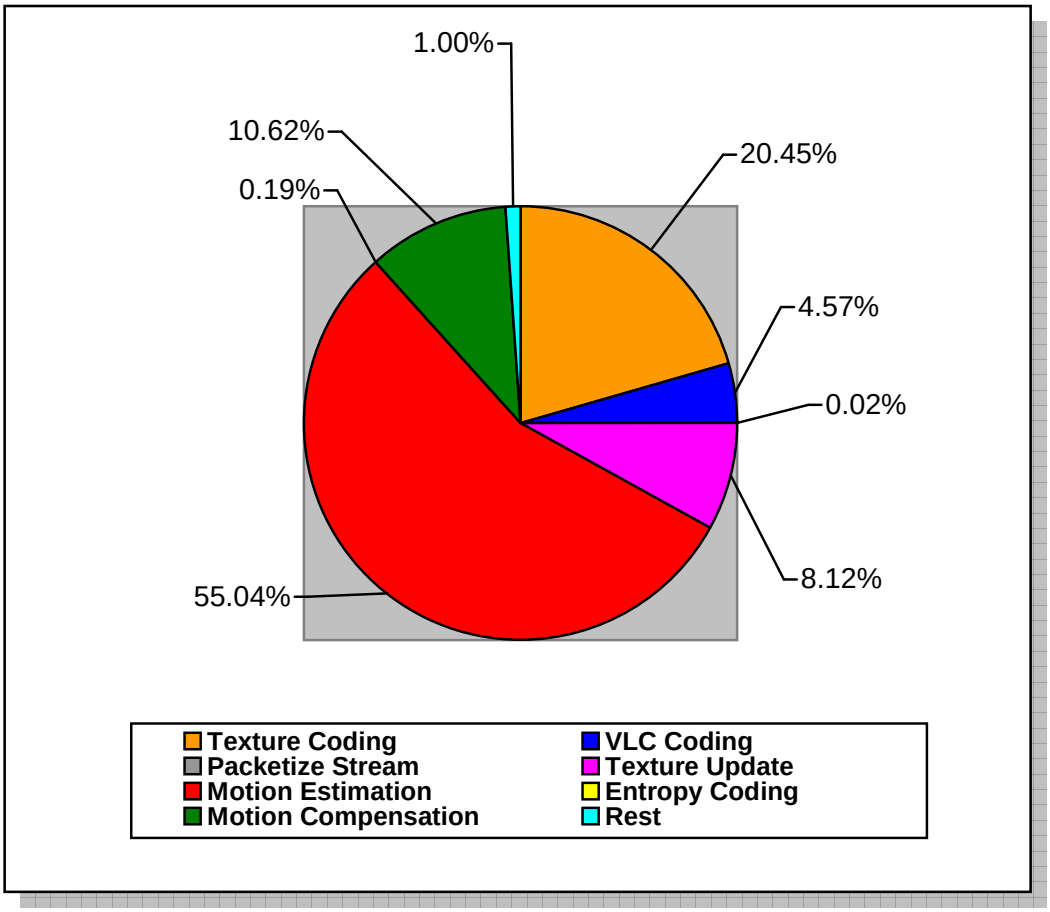
4.3 Code mapped

Before start changing the code it is necessary to analyze the code in order to know where it is necessary to concentrate efforts.

Graphic 1 shows us which parts of an MPEG-4 encoder has more specific weight in terms of cycles. Main parts are Motion Estimation block, where a 55,04% of total cycles are spent, and Texture Coding block with 20,45%. Therefore the main optimizing effort is spent in these 2 blocks.

Additionally Motion Compensation Block and Texture Update Block have rather importance, 10,62% and 8,12% respectively, and for this reason are also mapped.

Hence, code cycles spent in these four parts together are more than 94% of the total amount of cycles in VLIW mode.



Graphic 1 Cycles percentages spent in different parts of MPEG-4 encoder (VLIW mode, initial code)

Chapter 5
Texture Coding

5.1 Introduction

The first step of the TC function consists of choosing the block type:

- Blocktype 0: no computation is made.
- Blocktype 1: only 8 coefficients are computed.
- Blocktype 2: all 64 coefficients are calculated.

By distinguish 3 kind of blocks is possible to avoid calculating unnecessary coefficients. Due to the Discrete Cosine Transform (DCT) a great number of coefficients are 0 or near to 0. Finally, these coefficients can be discarded in the Quantization step.

The selection of the blocktype is carried out calculating with the Sum of Absolute Differences (SAD). The SAD finds the similarity between two macroblocks (MB). A greater similarity between the two matrices results in a smaller SAD value. Depending on his level the block types are chosen. In order to do this 2 thresholds are settled: if the SAD block is minor than the first threshold then the block is type 0 and the next steps are skipped; if the SAD block is between the 2 thresholds then the block is type 1 and only 8 coefficients will be computed; and if the second threshold is surpassed then the next steps will be executed for all 64 coefficients. Therefore, all the blocks in mode INTRA will be type 2.

Once the block type decision has been carried out, the next steps depend on the block type. As the number of iterations is greater when a block is type 2, we focus the optimization in this part of the code.

Figure 15 shows in general terms how Texture Coding function behaves.

As we compute all coefficients when the block is type 2, because the major number of cycles is spent here, but in order to decrease as much as possible the cycles of Texture Coding, the functions in mode 1 are also mapped. These are the functions mapped in CGA:

- _DRESC_BlockDirDCTQuantH263
- _DRESC_BlockDirDequantH263IDCT
- _DRESC_DCTrows
- _DRESC_DCTcols
- _DRESC_Q_invQ
- _DRESC_IDCTcols
- _DRESC_IDCTrows

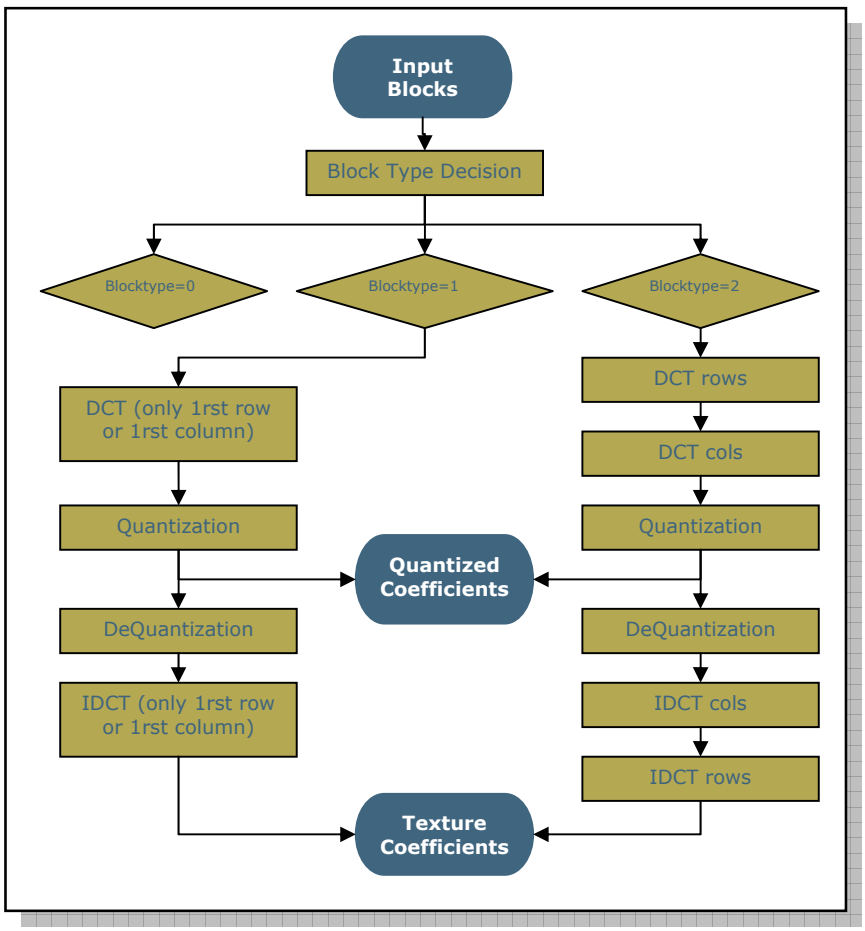


Figure 15 Texture Coding flowchart

To optimize the code 3 main steps are applied. In the first step, the code is modified in order to map into CGA (remove functions calls inside the loops, merging loops…). Once the different functions (DCT_Rows, DCT_Cols…) could be mapped, the code is adapted to obtain better results (decrease the number of cycles). Finally the last transformation step is to pass from block level to Macro Block level; the reason is to increase the amount of iterations in each loop. At the end of this process the optimization of kernels for blocks type 1 is performed.

5.2 Block Level

The original code was written in a way that the functions are inside the loop. Therefore, first the code was changed to split the loop in different loops for the different functions, as well to create de quantization-dequantization function (called Q_invQ).

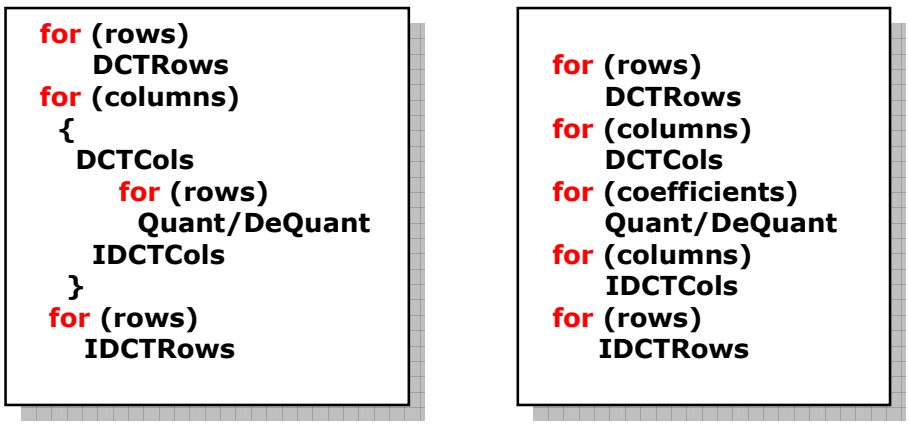


Figure 16 Initial and final schematic block level code

The first 2 functions were basically transformed to enable the mapping in the CGA. Therefore, the functions DCTrows and DCTcols, where a DCT is carried out first for the rows in the block and then for the columns, remained essentially the same as in the original code only changing the pertinent code to avoid function calls inside the loops.
These transformations results in the following scheduling characteristics

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
DCT_Rows	107	78,47%	9	4	12,55	34
DCT_Cols	98	76,56%	8	5	12,25	33

Table 1 Scheduling results for DCT

5.2.1 Quantization/DeQuantization function

This function has a higher complexity and many code transformations are applied. As the quantization is different depending on mode of the Macro Block (INTER or INTRA), there were 3 different loops in the original code: one for the INTRA blocks, one for the INTER blocks and the other to reconstruct coefficients.
In INTRA mode, the first coefficient has a special treatment, because it is the DC coefficient. It is quantized as follows:

$$QF[0][0] = \left. in[0][0] + \frac{dc\_scaler}{2} \right/ dc\_scaler$$

Whereas the other coefficients have this other quantization:

$$\left|QF[i][j]\right| = \left|in[i][j]\right| \Big/ 2 \cdot Qp$$

In INTER mode all the coefficients inside the block are quantized in the same way:

$$\left|QF[i][j]\right| = \left|in[i][j] - \frac{Qp}{2}\right| \Big/ 2 \cdot Qp$$

The formula to reconstruct the coefficients is as follows:

$$\left|F[i][j]\right| = \begin{cases} 0 & if \quad QF[i][j]=0 \\ (2\cdot\left|QF[i][j]\right|+1)\cdot Qp & if \quad QF[i][j]\neq 0 \quad Qp \quad is \quad odd \\ (2\cdot\left|QF[i][j]\right|+1)\cdot Qp-1 & if \quad QF[i][j]\neq 0 \quad Qp \quad is \quad even \end{cases}$$

First, a function for the quantization-dequantization was created, (_DRESC_Q_invQ). Once the function was created, there were 4 loops. The Figure 17 shows the loop disposition.

```
for (rows)
    for (columns)
        INTRA_Quantization
    for (columns)
        INTER_Quantization
    for (columns)
        Dequantization
```

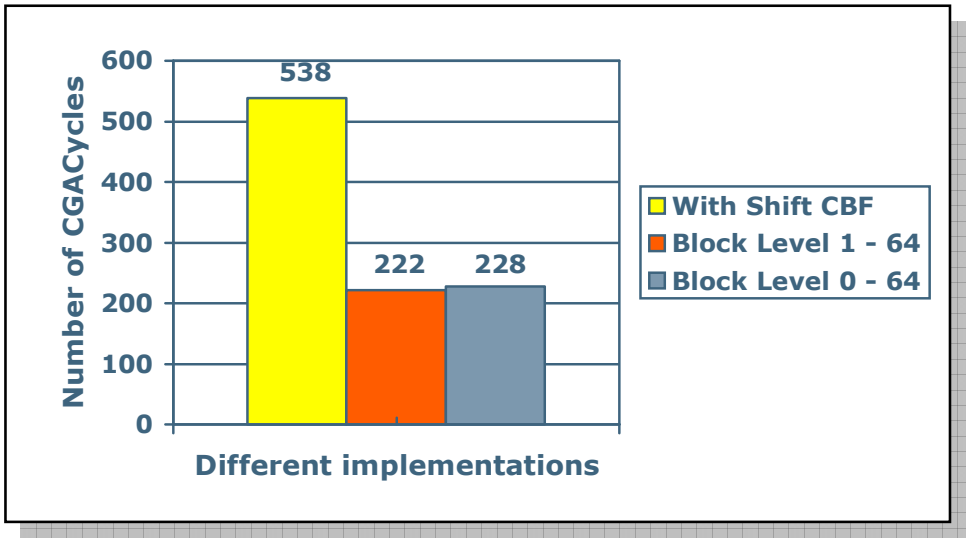
Figure 17 DRESC_Q_invQ original schematic disposition

Therefore, to able the mapping in CGA, the technique of loop coalescing was applied. Unfortunately the function still had 3 loops, because of the different quantization between modes, and also the last loop to reconstruct the coefficients.

To merge the 2 modes a parameter (called QuantA) was created. It has 2 different values depending on the mode: 0 for mode INTRA and *–Qp/2* for mode INTER. In addition, another parameter to select the sign of the quantized coefficients is created. At this point, to merge the loops 2 possibilities were tested:

1. Calculate the 64 coefficients and in INTRA mode recalculate the DC coefficient afterwards.
2. Calculate the first coefficient depending on the mode and choose the value of QuantA, and then calculate the 63 remaining coefficients.

Finally it's easy to put the code to reconstruct the coefficients at the end of the loop.



Graphic 2 Q_invQ implementation results

Graphic 2 shows the different results. First results are so worst because of a high dependency with the variable CBF. CBF indicates in which columns must be applied the IDCT, by setting to one the correspondent bit per column. As the code was changed this shift operation was removed and the number of cycles decreased dramatically.

The graph also shows that the first option (to calculate all the coefficients and then recalculate DC if block is mode INTRA) was worse than the second. Apparently this is logical because in this option the first coefficient is evaluated twice.

There is the possibility to unroll the loop in the first option, to try to decrease the CGA Cycles, unfortunately this option was tried but the CGA couldn't map, because of the architecture resources.

The Table 2 shows the scheduling results for the 2 options tested.

Kernel Option	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
1st Option	228	85,42%	3	11	13,67	36
2n Option	222	77,08%	3	11	12,33	29

Table 2 Scheduling results for different Q_invQ options

5.2.2 IDCT_cols function

This function already existed in the original code but there was a function call inside. So the first step is to put the main loop inside the function and change the function call by pasting the function code. Then some if statements that only adds overhead were removed.

At this point 2 options were tested:

1. Maintaining if CBF conditions.
2. Removing if CBF conditions.

CBF was used to know if the IDCT has to be applied in the block or not, on one hand the amount of iterations was reduced, but on the other hand this added more dependencies that means more cycles.

With the CBF condition it was possible to skip a block and only the IDCT is applied if the MB is mode INTRA.

The number of CGA cycles is the same for the 2 options because if statements are outside the loop, but the total cycles are different in the options. Whereas for with if conditions the number of cycles are 262.806 that means an average of 92,43 cycles/block without this conditions the total number of cycles is 292.562 an the average is 104 cycles/block.

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
IDCT_cols	88	79,46%	7	5	12,71	31

Table 3 IDCT_cols scheduling results

5.2.3 IDCT_rows function

Like in the other functions there was a function call inside the loop that was removed and replaced for code. Then 2 options were tested as there was also the clipping inside the same loop, it was tested the possibility to split the loop in 2 loops (one for the IDCT and the other for the clipping) but finally the results were worst if the loop was split.

shows the scheduling characteristics for these 2 options:

Kernel Option	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
1IDCT+CLIP	122	82,39%	11	3	13,18	33
IDCT	99	72,92%	9	3	11,66	27
CLIP	68	69,79%	6	4	11,16	20

Total CLIP + IDCT	167

Table 4 IDCT_rows scheduling results

5.3 MacroBlock Level

The idea of going to MacroBlock level is to increase the amount of iterations to reduce the "length effect" that means a decrease in the number of cycles.
In the original code, the TextureCoding function was running inside a loop for each block. Therefore the function was called 6 times in a MB and each time the function went over rows, columns... The transformation consisted in coalescing the main loop with the smaller function loops (Figure 18).
However the different block types were an issue. The solution consisted in create 3 arrays (for each type of block) where the number of block in the MB was stored in order to compute the correct number of coefficients for each block. Then the coalescing technique was applied.

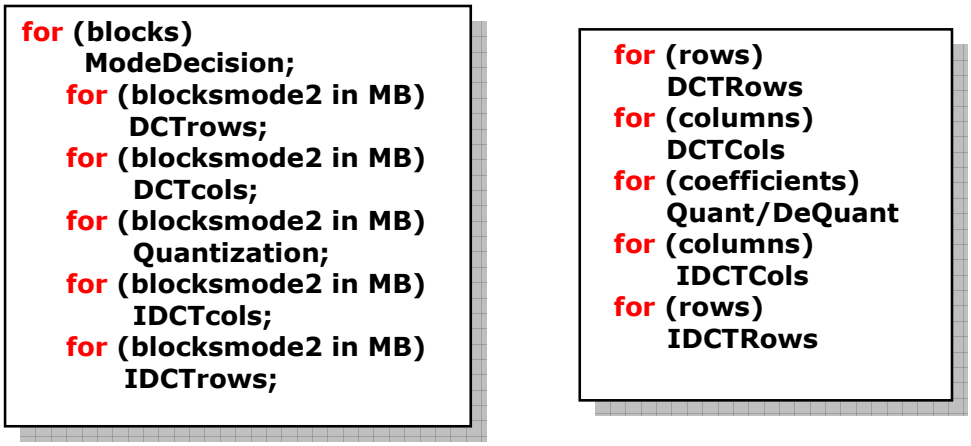


Figure 18 Schematic code in block and macroblock level

Another problem was that in MB level there were more reads at memory than in the block level because it is necessary to store the data in buffers between the different functions. Initially the data was written in the correct position in the MB and then it was changed by writing in consecutive form, and only wrote correctly the TC outputs (Quantized coefficients and reconstructed coefficients), but not the intermediate buffers employed. In this way the total number of reads at memory decreases.

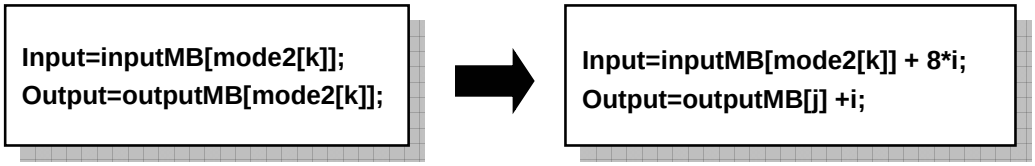


Figure 19 Changes in memory accesses

Where mode2[k] is the array which indicates block position in MB and i and j are used to access the different coefficients. What this means is that firstly the outputs were in the correct way but then it was changed to write consecutively.

Another way was tried but with worst results. To carry out the DCT and IDCT (over rows and columns), it is necessary to compute 8 coefficients in each iteration, that means 8 reads to the memory. To avoid this, instead of reading 8 times 16 bits (short type), the memory was read 4 times as integer type (32 bits) and then make shifts to store the correct value in the scalar variables. In this way the code had 10 cycles more per block.

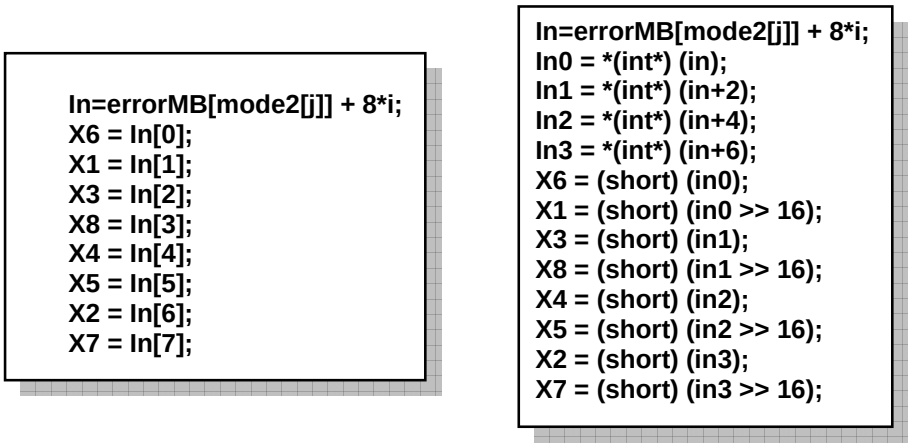


Figure 20 Different options to read from the memory: short type (left) or integer type (right)

Where In is a pointer, X1...X7 are short scalars and In0...In3 are integer scalars.

There was another issue with CBF condition. A buffer was needed to store the value of CBF for each block; as a result of this the number of reads at memory was high.

DCT_Rows tested	*NrCycles average*[1]	*SD*	*II*	*PS*	*IPC*	*Length*
Changing buffer	82,79	76,56%	8	7	12,25	43
Normal read/write	101,36	82,5%	10	5	13,2	49

Table 5 Scheduling results for different options to read at memory

[1] Cycles average depends on the number of blocks type 2 in a MB. All statistics are taken with a 300 frames qcif foreman sequence.

There are 2 things which are necessary to indicate. Firstly the number of cycles is an average because sometimes the number of iterations in the function is different. This is due to the different number of blocks in mode 2 in the MB. The second thing is that the II is reduced from 10 to 8, and in the block level it was 9. Therefore, this is an important point to be considered in future. In DCT_cols function happens the same. The reduction of load/store at memory allows to decrease the II and of course the number of cycles.

	NrCycles average	***SD***	***II***	***PS***	***IPC***	***Length***
DCT_cols	79,80	73,44%	8	5	11.75	36

Table 6 Scheduling results for DCT_Cols in MB level

In Quantization/DeQuantization function, the problematic with accessing to the memory was more visible because of the number of iterations is higher, 64*Nrblocks instead of 8*Nrblocks, in the other functions.

The clipping in this function and in IDCT_Rows functions, are different from the available intrinsic clipping functions. Therefore a new function was created. The reason is that the previously created intrinsic functions make the clipping as follows:

$$-MaxVal \leq Value \leq Maxval$$

$$0 \leq Value \leq Maxval$$

Whereas in the code has to be:

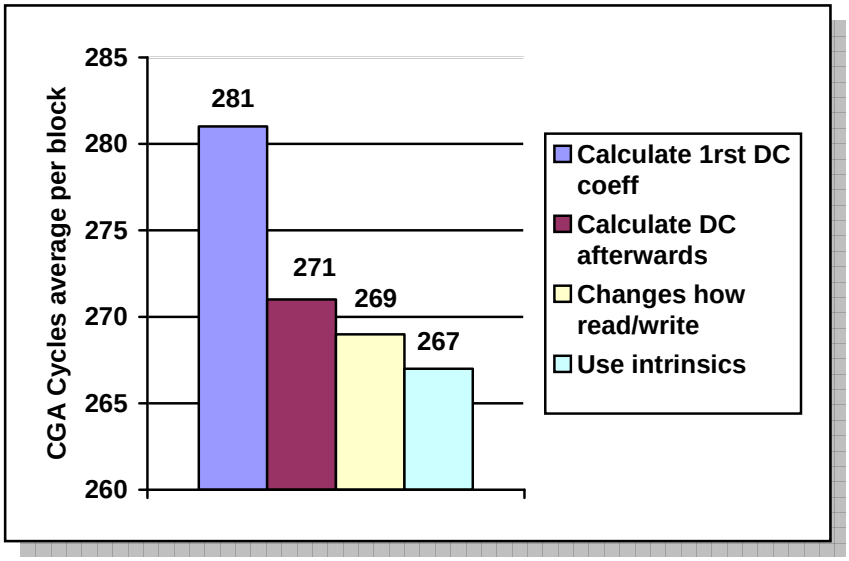
$$-MaxVal - 1 \leq Value \leq Maxval$$

$$1 \leq Value \leq Maxval$$

Once the function was mapped in MB level some transformations were made. Firstly the order of calculations is reverted to 1rst option, that means to calculate the 64 coefficients for all blocks and then, if it was necessary, the DC coefficients for INTRA mode blocks. Therefore there are 2 loops in the function. It wasn't necessary another loop to know the value of parameter QuantA because all blocks in the same MB are in the same mode.

As it was necessary to write the quantized coefficients in the correct block, the pass of *mode2* array, which indicates the position, was a ballast to decrease the cycles. In addition, the CBF parameter is transformed to an array, which means more accesses to the memory and more cycles.

Finally with the use of intrinsics the number of cycles decreased but not enough to reach the block level results.



Graphic 3 Q_invQ Implementations in MB Level

Quantization using intrinsics	*NrCycles average*	*SD*	*II*	*PS*	*IPC*	*Length*
64 loop	265,79	75%	4	8	12	30
DC coeff loop	1,26	65,62%	2	12	10,5	23
Total	267,06					

Table 7 Q_invQ scheduling results in Mb level

In the IDCT_Cols function the effect of changing the way to access at memory was so important and the number of iterations decreased dramatically. To carry out these accesses, the if CBF conditions were removed. This couldn't be possible in the IDCT_Rows function.

IDCT_Cols	*NrCycles average*	*SD*	*II*	*PS*	*IPC*	*Length*
With CBF conditions	100	64,77%	11	4	10,36	
Changing reads/writes	65,08	86,46%	6	7	13,83	39

Table 8 Different scheduling results for IDCT_Cols in MB level

In IDCT_Rows function it wasn't possible to avoid the CBF condition outside the function. In order to know which blocks have to be computed it was created another loop before the function. In this loop was stored the position of the block in a new array if the CBF of

this block was 1. This is the reason because it wasn't possible to decrease so much the number of cycles, as the input needs to know the position of block and the output must be correct.

	NrCycles average	*SD*	*II*	*PS*	*IPC*	*Length*
IDCT_Rows	86,86	84,38%	8	6	13,5	45

Table 9 IDCT_Rows scheduling results obtained in MB level

5.4 Block Type 1

Although in this mode there are only 2 functions to map in the CGA (BlockDirDCT-QuantH263 and BlockDirDequantH263IDCT), inside of this functions there were several loops. These 2 functions make all the computation for 8 coefficients. The first function choose in which direction has to make the computation (compares adds of rows versus columns and the highest is chosen), and then makes DCT for this 8 coefficients and the quantization. The second function makes inverse process. Firstly makes the dequantization, then the IDCT and finally stores the results.

To be able the mapping in CGA some transformations in the code were made. Firstly splitting the main loop and pasting the function code inside. Then, the coalescing of loops and some variable transformations (in order to maintain the code functionality and to avoid some unnecessary accesses at memory) were made. Finally the clipping intrinsic function was added.

BlockDirDCTQuantH263	*NrCycles average*	*SD*	*II*	*PS*	*IPC*	*Length*
Adding Rows & Cols	95,64	83,04%	7	5	13,28	32
Tmp_DC	78,29	82,81%	4	6	13,25	23
Choosing direction	48,51	59,34%	4	8	9,5	30
DCT	53,32	71,63%	13	4	11,46	43
Quantization	45,26	57,81%	4	7	9,25	27

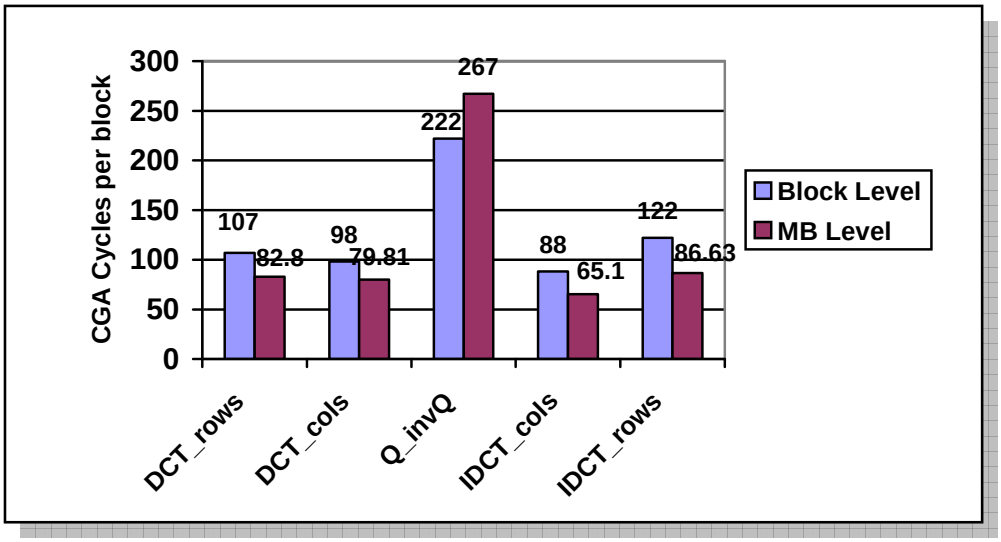
Table 10 BlockDirDCTQuantH263 scheduling results

BlockDirDequantH263IDCT	*NrCycles average*	*SD*	*II*	*PS*	*IPC*	*Length*
Dequantization	62,93	75%	3	12	12	36
IDCT	21,26	78,13%	2	13	12,5	27
Write coefficients	73,45	83,33%	6	6	13,33	36

Table 11 BlockDirDequantH263IDCT scheduling results

5.5 Comparison between block and MB levels

As the graphic shows in all functions, except in Quantization, the results were better because of the major number of iterations.



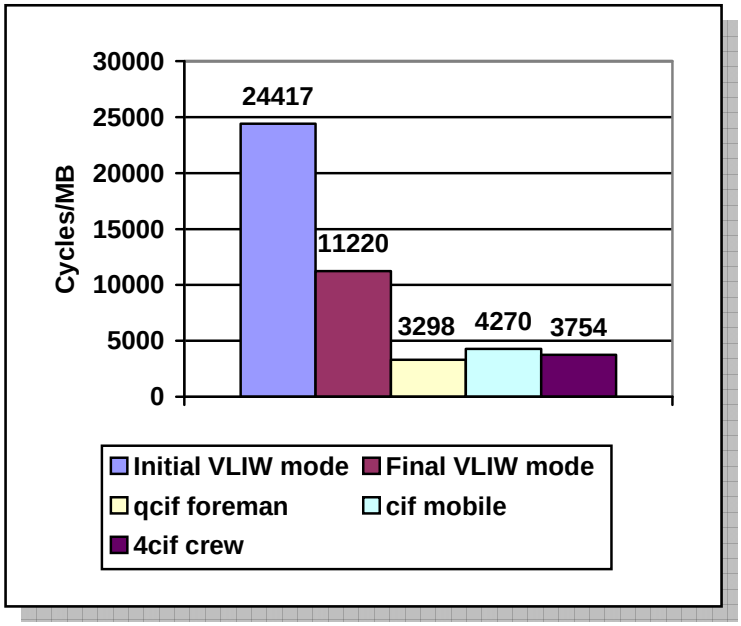
Graphic 4 CGA Cycles comparison between block and MB levels

	Block Level	*Final Implementation*
Cycles / frame	401.019	326.590
Cycles / block average	675,11	549,81
Kernel Cycles / block average blocktype = 2	587,41	544,58
Kernel Cycles / block (worst case) blocktype = 2	637	648
Kernel Cycles / block average	587,41	425,33

Table 12 Block and MB level statistics (Q_invQ in block level)

The graphic show us that results are better when the code is implemented in a MB level, in spite of the Q_invQ function, where the CGA cycles in MB level couldn't be reach the block level because of the major number of accesses at memory that is translated in a major II. For this reason in the final implementation in Q_invQ function is only mapped the inner loop (64 iterations).

Table 12 shows a comparison between the block level implementation and the final implementation. The final code decreases in 19% the total number of cycles, therefore it is worth to increase the number of iterations in the loops in spite of some operations that is necessary to add. Next graph (Graphic 5) shows a comparison between the initial code in VLIW mode, the final code in VLIW mode, and the final code mapped in CGA for 3 different sequences[2]. As it was expected cycles decrease dramatically by mapping into CGA (70%).



Graphic 5 CGA mode and VLIW mode results

Here follows a simple calculation of the max number of cycles required to be able to have HDTV Texture Coding operating real time on a single ADRES:

- HDTV resolution = 1.280 * 720
- HDTV framerate = 30 fps
- MPEG-4 input = YUV 4:2:0
- MacroBlock size = 16*16
- Blocks per Macroblock (YUV) = 6
- ADRES clock = 300 MHz = 300 M cycles/s

[2] Results obtained with 4cif crew sequence are wrong, see Conclusions chapter to see details.

- Pixels per frame = 1.280*720 = 921.600
- MBs per frame = 921.600 / 256 = 3.600
- Blocks per frame = 3.600 * 6 = 21.600
- Blocks per second = 21.600 * 30 = 648.000
- Cycles per block = 300.000.000/648.000 = 463

The necessary clock-speed required for a single ADRES given the current number of cycles:

- Current number of cycles per block = **549,81**
- Current number Texture Coding cycles per second = 549,81 * 648.000 =
  = 356.276.880 = **356,27 MHz**

Therefore to implement the texture coding block taking into account that is necessary to implement the others blocks the clock-speed necessary will be 356,27 MHz.

Chapter 6
Motion Estimation

Due to the large amount of cycles that are involving this block of the MPEG-4 encoder (55% of total encoder cycles), it is important to optimize properly this part. Furthermore Motion Estimation (ME) is one of the most power-consuming components of any predictive video codec.
The goal of ME is to find a 16 × 16-sample region (a MB) in a reference frame that closely matches the current macroblock. The reference frame is a previously encoded frame from the sequence and may be before or after the current frame in display order. An area in the reference frame centered on the current macroblock position (the search area) is searched and the 16 × 16 region within the search area that minimizes a matching criterion is chosen as the 'best match'. The information of the 'best match' called Motion Vector (MV) is then transmitted to the Motion Compensation (MC) block as 2 components: horizontal and vertical. A MV can be expressed in integer or half-pixel accuracy.
The chapter is organized as follows: first a brief introduction of the ME applied algorithm is given. Then the next sections contain the different transformations for each part of the block and the statistics of each loop. At the end some general statistics for al the ME block are presented.

6.1 Motion Estimation description process

A large number of ME algorithms have already exploited the statistical properties of MVs distribution to achieve very good performances in terms of computational complexity reduction. The number of searched locations has been sharply reduced in comparison with the Full Search algorithms, while preserving the performance [12]. The algorithm follows the basic rules of these others algorithms:

- The algorithm should exploit the spatial correlation of MVs within a frame. Often, this is done through the prediction of the search starting point. It could also be used to adapt the search parameters, e.g., the size of the search pattern.
- To exploit the MVs' center-biased distribution, the checking points should be chosen on a pattern that is compact around the initial and central position.

- For faster convergence toward an optimum, the checking points should be chosen adaptively, in the direction of an improvement of the matching criterion gradient descent algorithms).
- Finally, the search should stop as soon as possible, i.e., once the matching criterion is good enough (typically, below some threshold). Making it unnecessary to investigate all the positions of a pre-defined pattern.

The algorithm only operates on Y-inter MBs, as in intra MBs the MC is not needed. Instead, if the MB is coded as mode intra (see Texture Coding chapter) only the calculation of the Sum of Absolute Differences (SAD), which in this case is the sum of the pixel values as there is not reference, is performed.

The algorithm starts to operate calculating the SAD from the current MB, which is predicted from the neighboring MB MVs. Then check if this SAD, and make the same with other SADs if it is the case, is below a certain threshold. If it is not the case, it continues calculating and checking the neighboring SADs.

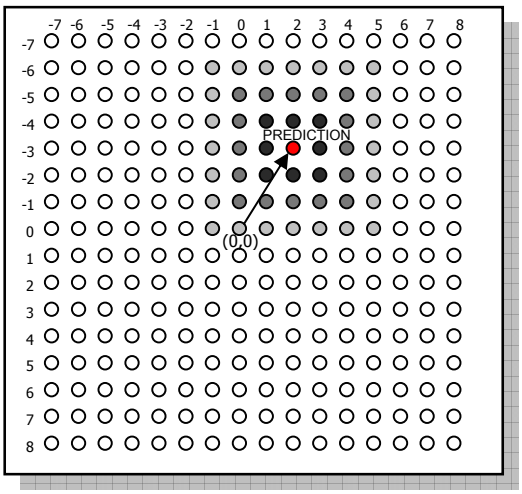


Figure 21 Different search zones. Darkest are search first

If there is not SAD that reaches the minimum threshold, the minimum SAD calculated previously is chosen as the new center. When the minimum SAD is improved on a square that position is used to predict the optimal position (center) in the next square. Actually, the predicted position is the one pointed by the vector originated in the center of the search and passing through the last optimal position. The predicted position leads to the search strategy depicted on Figure 22 Checking points process. Only the neighbors of the predicted position are investigated before going to the next square. The search is stopped as soon as there is not further improvement or the threshold is reached.

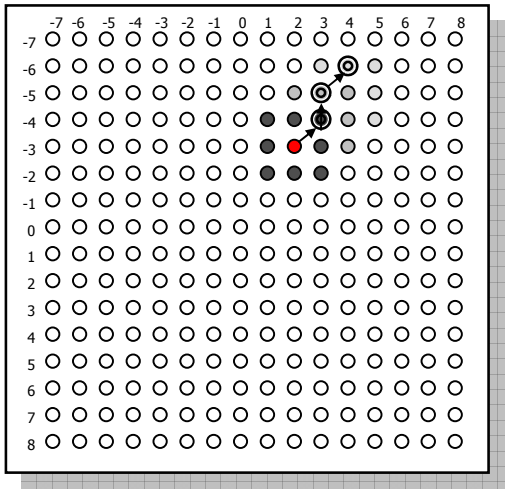


Figure 22 Checking points process.

When this process is finished de mode decision is performed: if the SAD of the best MB is below the mean absolute value difference of the current MB, it will be encoded as an inter MB, otherwise it will be coded as an intra MB.

Finally if the SAD found is bigger than another threshold, it is necessary to perform the Half Pixel ME, which basically consists on:

- Calculate the interpolate pixel values (8 values per current pixel).
- Calculate the 8 SADs of these half pixels MBs.
- Choose the best option between this 8 options and the current pixel (see optimizations in half pixel).

6.2 Code transformations

In ME the use of intrinsics become to be more relevant. Intriniscs allow taking advantage of working with integer types (4 bytes) instead of short types (1 byte) or unsigned char types (1 byte), with the correspondence gain of cycles. These are the intrinsics implemented in the ME:

- Innersum4: calculates the sum of the 4 bytes in a 32-bit integer.
- Subabs4: calculates the absolute difference of each corresponding byte from two 32-bit integers.
- Pack2: stores two 16-bit values in a 32-bit integer
- Spacku4: saturates the Low Significant Byte (LSB) and Most Significant Byte (MSB) parts of two 32-bit integers into 8-bit unsigned integers, and packs the results into a 32-bit integer.
- Avgu4: calculates the average of each corresponding byte from 32-bit integers.

6.2.1 Calculate INTER – MB SAD

As the name of the function says, this function calculates the SAD of an INTER MB. The process is easy. Consists on making the subtraction from the pixels of 2 MBs, one pixel of the current frame and the other of the reference frame, and adding the absolute value of this difference.

The transformations (Figure 23) consist on coalescing the loops to go over columns and over rows, and changing the normal 8-bit operations for the 32-bit operations with the intrinsics: Subabs4 to perform the absolute subtraction of 4 bit and Innersum4 to add these absolute differences in one integer.

After this initial transformation, the loop is unrolled to decrease cycles, as the initial Scheduling Density (SD) was not to much high (71,88%).

In order to read from the frames (memories), 8 integers pointers are created (because the loop is unrolled). It is also possible to use integers instead of pointers but there is not cycling difference between the 2 options. Another possibility could be to create just 2 pointers (one per frame) and increment the pointers. This option was already tested and as a result the number of cycles was increased (4 cycles only). Nevertheless with this option the result are better in the INTRA-MB SAD.

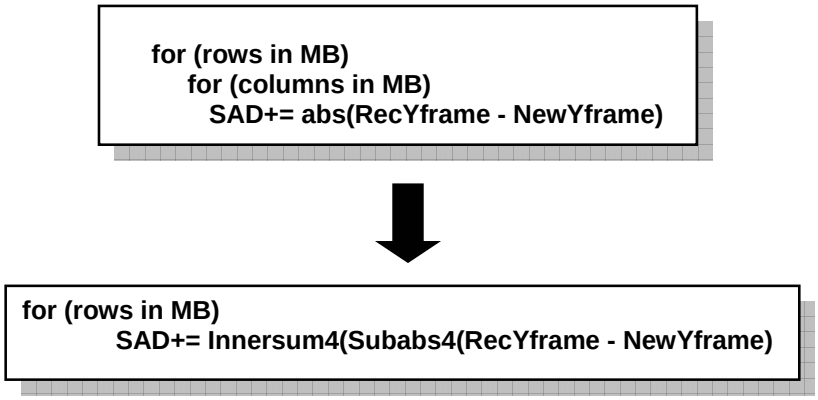


Figure 23 Code transformations in INTER-MB SAD

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
INTER-MB SAD	70	75,00%	3	7	12	21

Table 13 INTER-MB SAD schedule results

6.2.2 Calculate INTRA – MB SAD

The difference between an intra-MB SAD and an inter-MB SAD is that in intra-MB there is not reference frame. This means that is not necessary to subtract the pixels as the reference is 0. The Figure 24 and Table 14 show the transformations made and the results obtained.

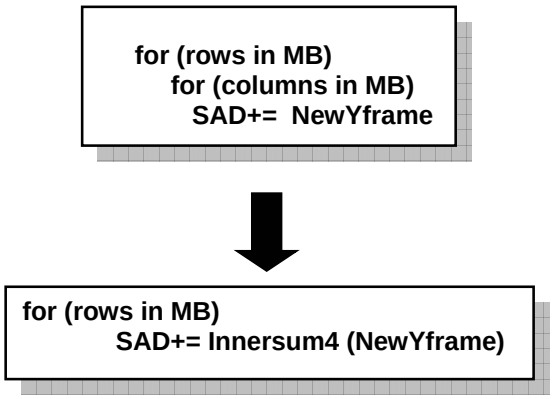


Figure 24 Code transformations in INTRA-MB SAD

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
INTRA-MB SAD	54	75,00%	2	11	12	21

Table 14 INTRA-MB SAD schedule results

6.2.3 Choose Mode

This function is used to choose the mode (inter or intra) that the MB will be codified. The selection of the mode is performed by comparing the SAD of the MB with the absolute difference between the mean of the MB and each pixel. If the SAD is below this difference the MB will be encoded as inter MB.

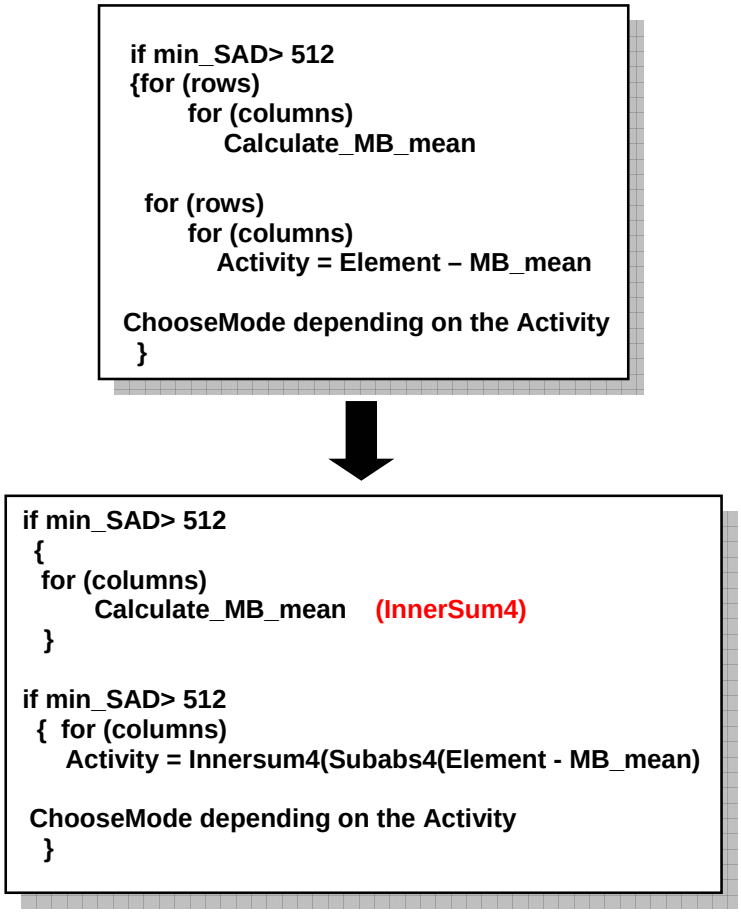


Figure 25 Code transformations in ChooseMode function

Notice that as shows the calculations are only performed if the MB SAD reaches a threshold (512). This if condition is changed by 2 if conditions (one per each loop). The if condition produces an influence between the loops. This influence consists on if both loops are unrolled it is not possible that the loops reach the lower number of cycles obtained when only one is unrolled.

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
Calculate_Mean	51	68,75%	2	9	11	18
Difference	54	76,56%	2	13	13	22

Table 15 ChooseMode schedule results

6.2.4 Full – Pixel Motion Estimation

In the Full-Pixel Motion Estimation the original algorithm is changed in order to increase the number of iteration in the first "spiral" loop. In the first "spiral" the calculation of the different SADs is performed until the threshold is reached, which is not so common. This condition is eliminated to unroll the loop and calculate all 9 SADs together. In addition to make the loop more regular (without if conditions) the original loop is divided. Therefore there are 2 loops: the first to calculate the 9 SADs and second, with several if statements to set and store some parameters for next searches. The Figure 1 depicts schematically the transformations in the first spiral. This second loop can not be mapped in the CGA because it's really irregular.

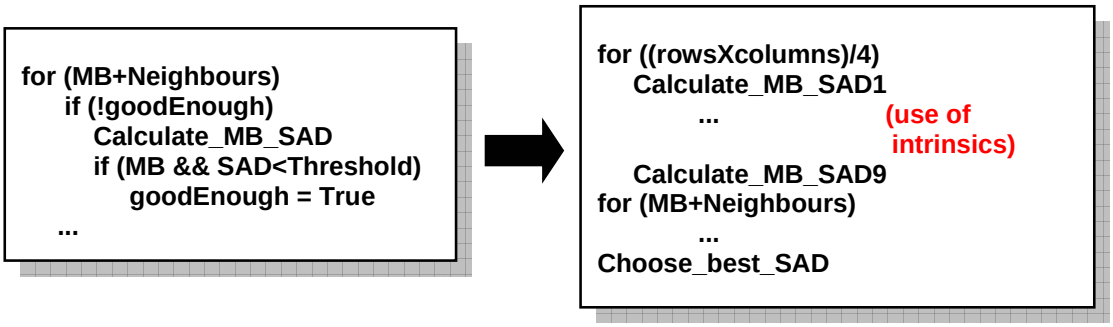


Figure 26 Code transformations in Full-Pel ME

Applying this changes the number of cycles spending in the first spiral pass from 1167 to 603 cycles (almost decrease a 50%).

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
First Spiral	421	67,71%	6	5	10,83	27

Table 16 Scheduling results for Full-Pel ME

6.2.5 Half – Pixel Motion Estimation

As already said in the ME description process section, after the Full-Pixel ME is performed, if the mode MB is inter, the Half-Pixel ME is performed. The goal of Half-Pixel ME is to find a better match for the current MB by calculating the interpolated samples.

The process consists on calculate the 8 SADs from the 8 neighbors of the best match found in the Full-Pixel ME, and check if one of these is better than the current pixel. In order to calculate the SADs it is necessary to interpolate all the values from the MB. Figure 27 depicts the different kinds of interpolated values.

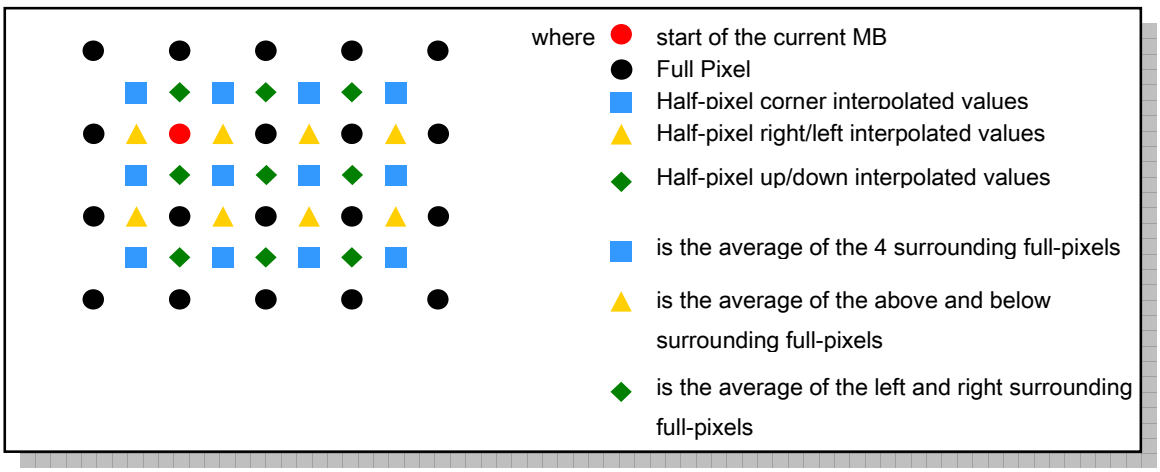


Figure 27 Half-pixel types

In the initial code the calculation of both interpolation values and SAD of MB were made in the same, which also was inside another loop (to go over the 8 half-pixel neighbors). In the final implementation 2 different loops are used. In the first loop the interpolated values are calculated (using intrinsics, details in appendix A) and stored. In the second loop all 8 SADs are calculated (also with intrinsics). The reason is to take advantage of the intrinsics (decrease the total number of iterations), avoid recalculating the interpolation values and the possibility to calculate the 8 half-pixel SADs in the same loop (unrolling the loop). In addition another small loop is created in order to choose the minimum SAD.

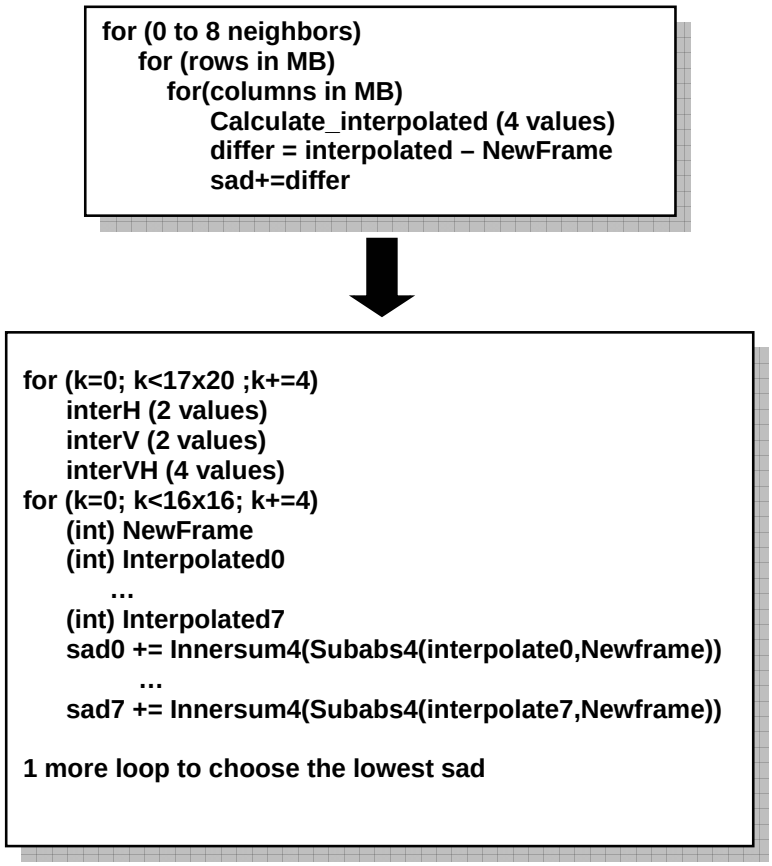


Figure 28 Code transformations in Half-Pel ME

Another two options were tested with worst results (Graphic 6). These options consist on calculate the vertical and horizontal interpolated values from the reference frame (memory) and then the 4 surrounded half-pixels by making the interpolation from the previous half-pixels. The reason to try these options was to avoid to accesses to the memory. Unfortunately to perform these implementations, it is necessary to increase the number of conditions inside the loop or to create several loops.

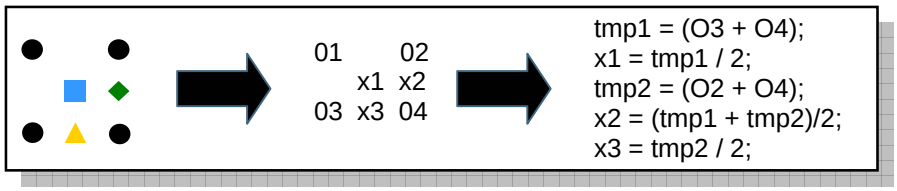
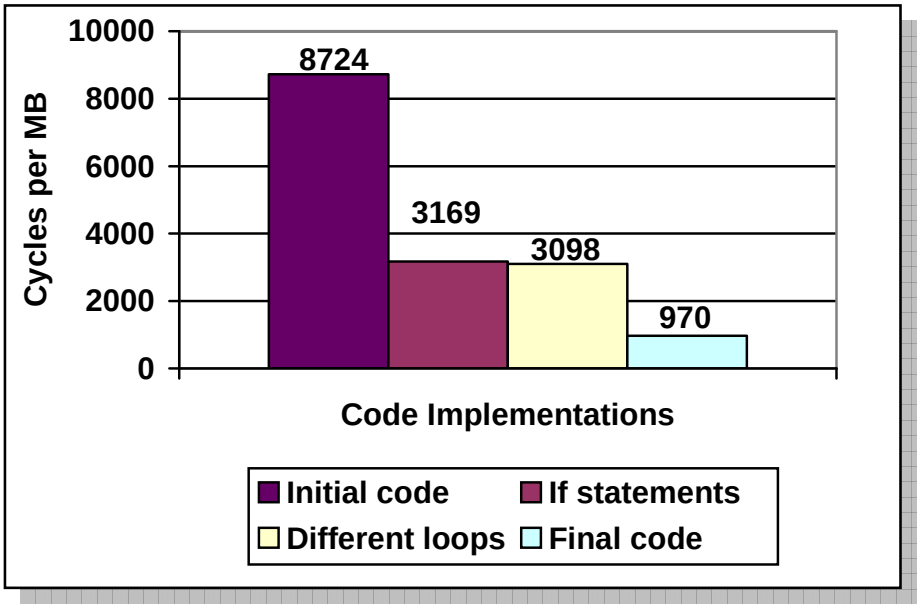


Figure 29 Scheme of interpolation value using previous calculated values



Graphic 6 Total cycles in Half-Pel ME for different option tested.

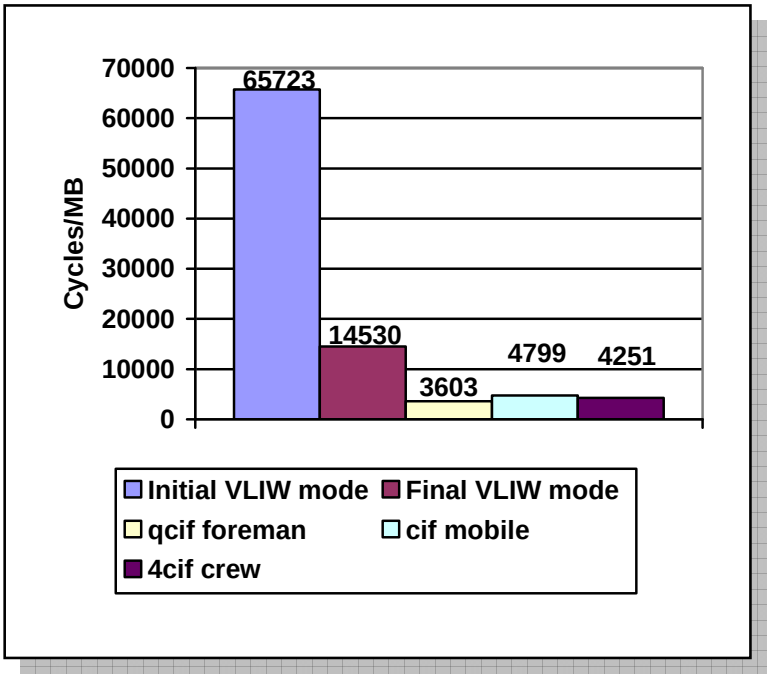
Table 17 shows the scheduling results for Half-Pel ME with the final implementation. The final loop for choosing the minimum SAD has not a high

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
Interpol_values	626	73,21%	7	5	11,7143	30
Calc_SADs	346	67,50%	5	5	10,5	25
Choose_min_SAD	37	33,33%	3	4	5,33	12

Table 17 Half-Pel ME schedule results

6.3 General statistics

Graphic 2 shows a comparison between the initial code in VLIW mode, the final code in VLIW mode, and the final code mapped in CGA for 3 different sequences. Final code in VLIW mode has 77,89% less cycles than the initial code. Notice that there are big differences depending on the sequences tested. E.g. the ME block of the mobile cif sequence requires 24,9% more cycles than foreman qcif sequence. The reason is that the mobile sequence has a lot of motion in background and the algorithm needs more cycles to find the correct MB.



Graphic 7 ME comparison between different sequences and CGA/VLIW modes

As in the previous chapter, to have some means of comparison, here follows a simple calculation of the maximum number of cycles required to be able to have HDTV Motion Estimation operating realtime on a single ADRES:

- Cycles per MB = 300.000.000 / 108.000 = 2.778
- Current number of ME cycles per MB = **3.510**

The necessary clock-speed required for a single ADRES given the current number of cycles:

- Number of ME cycles per second = 3510 * 108.000 = **379,08** MHz

Therefore to implement the motion estimation block in a single ADRES, the necessary clock will be 689,09 MHz.

Chapter 7
Texture Update

Texture Update block represents de 8,12% of the total code in VLIW. In this block the MBs from the Texture Coding block are added with the MB coming from the Motion Compensation block, in order to create the MBs that will make the reference frames.
Unfortunately it is not so easy because there are different kinds of blocks, which mean different codifications. This chapter starts with the description of the initial code and follows in next sections with different options tested, giving only scheduling details for the final implemented code.

7.1 Initial code

In the initial code the texture update block operates in a block level, because each one of the 6 blocks in a MB (4Y and 2UV), can have a different kind of codification. In order to know which kind of block is coded, a parameter called TextureUpdateMode is passed to the block. This parameter comes from the Texture Coding block and its value depends on how the block is coded there:

- TextureUpdateMode=0 → Motion vectors are 0 and the block is skipped (no quantization perform, see chapter 5).
- TextureUpdateMode=1 → Motion vectors are not 0 and the block is skipped.
- TextureUpdateMode=2 → Only the first row it is codified.
- TextureUpdateMode=3 → Only the first column it is codified.
- TextureUpdateMode=4 → The block is mode INTER and all the coefficients are computed.
- TextureUpdateMode=5 → The block is mode INTRA but is not coded, only the DC coefficient.
- TextureUpdateMode=6 → The block is mode INTRA and all the coefficients are codified.

The consequence of the different block types is that to perform the block (that will construct the future reference frames), it has to be chosen the correct values from the reconstructed MBs (from Texture Coding block) and, if it is necessary, with the MB values that come from the Motion Compensation block.
The Figure 30 shows the schematic initial code of Texture Update section.

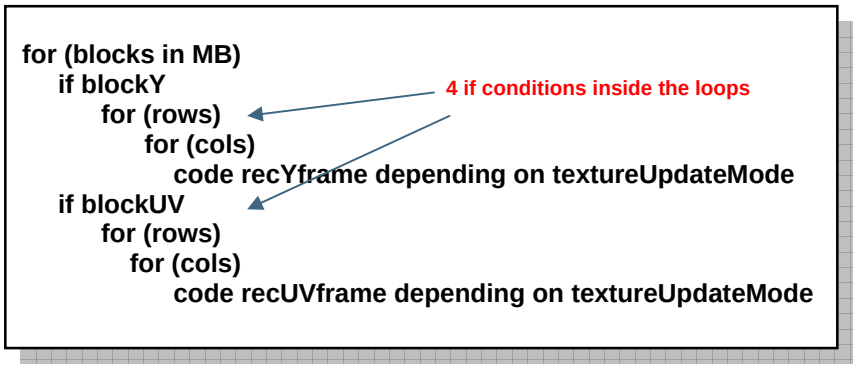


Figure 30 Schematic initial code in Texture Update block

7.2 Final code

With the experience acquired in previous sections I tried to:

- Make one big loop in order to take advantage of increasing the total number of iterations to map in the CGA. As there are 6 blocks in the initial code (even coalescing the loops over rows and columns), the initial code only maps the inner loop (64 iterations). Indeed it is the same than in the Texture Coding block (pass the function in a MB level).
- Remove if statements: in order to make more regular the loop and reduce the *II,* to reduce the cycles.

It is difficult to apply this 2 points at the same time, because the dependencies are not in a MB level (e.g. in the Quantization/invQuantization in Texture Coding block there are 2 different quantization depending on the MB mode). Therefore the selection of the different adds must be done before the loop.

It is easy to see in the code that:

- If the blocks are type 0 or 1 no information from the Texture Coding block has to be added.
- If the blocks are type 5 or 6 no information from the Motion Compensation block has to be added.

To avoid if statements and to avoid to add extra information, 2 new pointer arrays (of size 6 i.e. 6 blocks), one empty block (only 0s inside) and block to store the DC coefficient are created. The purpose of the pointers is that before the loop, and depending on the block type, they point to the block information (from MC or TC) or if it is a mode which does not need that information block they point to the 0s block. Figure 31 shows schematically this process. There are more issues but this is basically the process.

```
*Point1[blocks in MB];
*Point2[blocks in MB];

for (i=0:blocks in MB)
     Point1[i] = &zero_block[0];
     Point2[i] = &zero_block[0];

   if (textureUpdateMode < 5)
      Point1[i] = &MCblock[i][0];

   if (textureUpdateMode > 1)
     Point2[i] = &TCblock[i][0];
```

Figure 31 Schematic blocktype selection process

Aside from the pointers, some new variables are created to read in the correct order the values from the information blocks. For example, once we know that it's necessary to read the block from TC, it is necessary to know if we need to read the entire block or just the first column or first row (modes 3 and 2). For this reason there are new if conditions outside the loops. Although they have rather influence in total amount of Texture Update cycles (about the 33% of cycles are spent outside the loops), the results are still better than other options.

Once this code works properly next step is to reduce the amount of iterations by applying intrinsics that means other issues. There are several intrinsic operation to carry out the Texture Update with intrinsics, also depending on the mode it is possible to it with less intrinsics operations, but this means to add if statements. At the end it was not possible to use the same intrinsic code for all the block types and for the type 3 it is necessary to add if statements, but even with this drawback the results obtained are better (see for comparisons between different codes).

The shows the code (schematically) from the main loop. Aside from this loop 3 more loops are created: one (already explained) to select the where the pointers has to point depending on the block type, another loop is just to full of zeros the 0s_block. The third loop is created to fill the DC coefficients in case the block is type 5. This is performed in a separate loop because if it was made in the pointers loop the results would be wrong in CGA mode (read-store values in same iteration).

```
for (k=0 to MBvalues; k+=4)
    read *(int*)&point1[0][blocktype_params];
    read *(int*)&point2[0][blocktype_params];
    read *(int*)&point2[0][blocktype_params];

    if UpdateMode[0]=3
          Calculate RefFramevalues IntrinsicsMode3
    else
          Calculate RefFramevalues Intrinsics
      ...

    read *(int*)&point1[5][blocktype_params];
    read *(int*)&point2[5][blocktype_params];
    read *(int*)&point2[5][blocktype_params];

    if UpdateMode[5]=3
          Calculate RefFramevalues  IntrinsicsMode_3
    else
          Calculate RefFramevalues  Intrinsics
```

Figure 32 Main loop in Texture Update

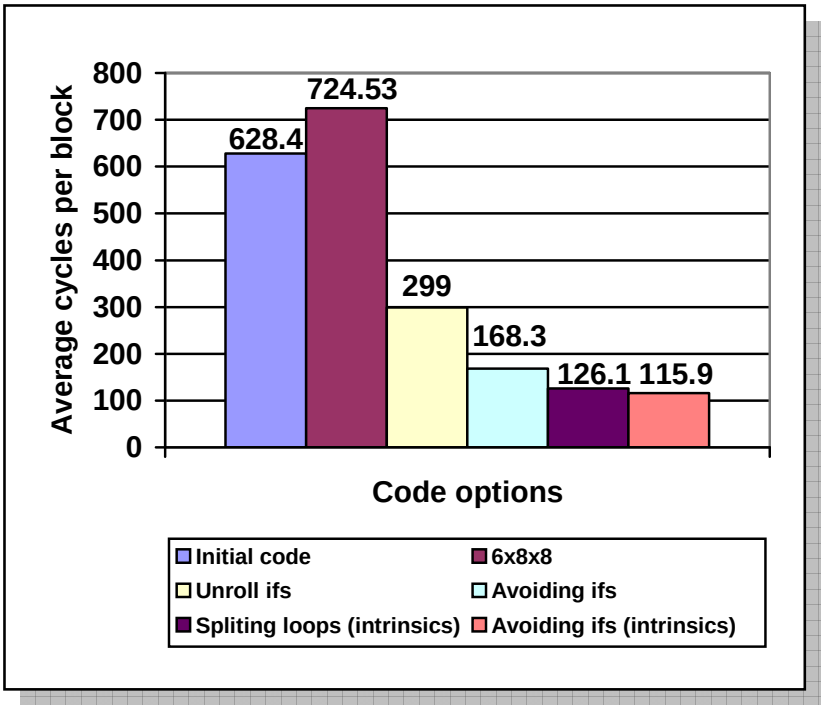
The Table 18 shows the final scheduling results. Notice that the *II* of the main loop is 20, which is a high number; on the other hand there are only 16 iterations. This *II* could be reduced if it was possible to remove if conditions inside the loop. This option was already tested and the *II* was 16, which mean a reduction of 64 cycles per MB. For specific intrinsics code see appendix A.

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
Array0s	47	43,75%	5	2	7	6
Mode parameters	27	71,88%	2	7	11,5	14
Fill mode5	32	40,63%	2	10	6,5	19
Main loop	361	76,56%	20	2	12,25	40

Table 18 Scheduling results for Texture Update block

Graphic 8 shows the different options tested during the progress to the best option. The first option is just the initial code coalescing the row-columns loops. The second option corresponds to the initial code but also coalescing the outside loop that is going over the different blocks. Indeed it means to pass from the block level to a MB level, increasing the iterations from 64 to 384. Here it is possible to appreciate the same problem than in the quantization function in TC (see the Texture Coding chapter), because the

architecture doesn't have enough resources to maintain the same *II* than in less iterations (ResMII).



Graphic 8 Comparison between different options tested

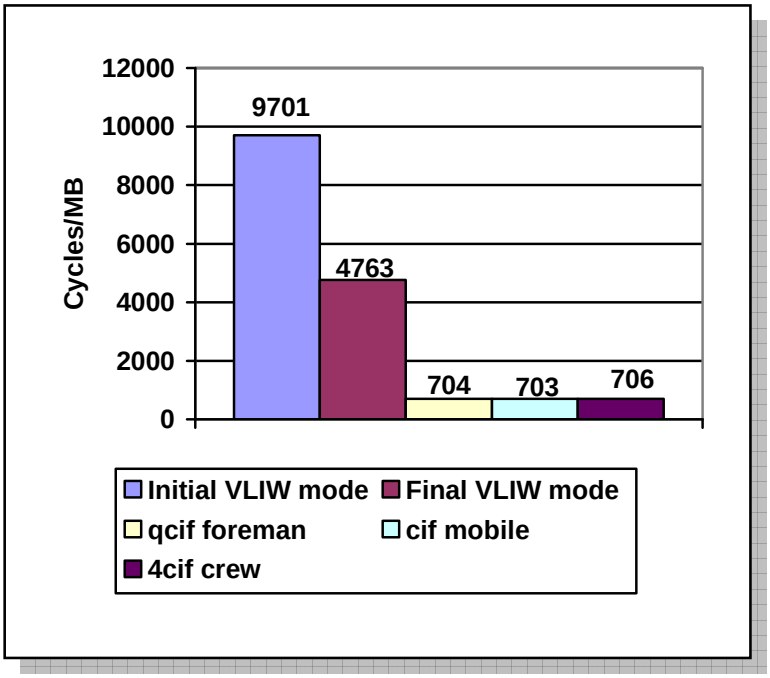
The result of next option (unroll ifs in the graph) corresponds at the original code (all ifs statements inside loop) but unrolling the loop 6 times. It means that all 6 blocks in the MB are updated together.

The next option consists on avoiding the if statements, by creating pointers and the Zero array. Finally, there is the final code with the application of intrinsic functions.

It has to be noticed the option of "*splitting loops*". In this option the code is implemented in different loops depending on the block type. In this way it is no possible to unroll the loop to perform the 6 blocks together but on the other hand there are not if statements inside loops mapped in the CGA and it takes profit to the intrinsic functions.

7.3 General statistics

The total amount of cycles per MB spent in the Texture Update block, adding the different loops and the control code is 704 for a foreman qcif sequence, 703 for a mobile cif sequence and 706 for crew 4cif sequence (Graphic 9).



Graphic 9 TU comparison between different sequences and CGA/VLIW modes

Like in the other previous chapters here follows a simple calculation of the maximum number of cycles required to be able to have HDTV motion estimation operating realtime on a single ADRES:

- Cycles per MB = 300.000.000 / 108.000 = 2.778
- Current number of TextureUpdate cycles per MB = **705**

The necessary clock-speed required for a single ADRES given the current number of cycles:

- Number of TextureUpdate cycles per second = 705* 108.000 = **76,14MHz**

Therefore to implement the motion estimation block in a single ADRES, the necessary clock will be 76,14 MHz.

Chapter 8
Motion Compensation

Motion Compensation block together with Motion Estimation block are the key blocks for the MPEG-4 standard (also in others standards) because they allow to reduce the redundancy between transmitted frames by forming a predicted frame and subtracting this from the current frame.
The motion compensation process consists on, once the “best” matching region in the reference frame is selected, typically a macroblock, the region is subtracted from the current macroblock to produce a residual macroblock (luminance and chrominance). After the Motion Compensation block, this macroblock is encoded and transmitted together with a motion vector describing the position of the best matching region (relative to the current macroblock position).Within the encoder, the residual is encoded and decoded and added to the matching region to form a reconstructed macroblock which is stored as a reference for further motion-compensated prediction. It is necessary to use a decoded residual to reconstruct the macroblock in order to ensure that encoder and decoder use an identical reference frame for motion compensation.
Therefore in the Motion Compensation block the goal is to subtract the current macroblock from the reference macroblock or if it is an intra macroblock to copy the values to the reference frame.

8.1 Initial code

In the initial code, the function where the motion compensation is performed is called 6 times. Actually there are two different functions: one called four times per macroblock for luminance blocks and the other function, for the chrominance blocks. However the motion vectors, which indicate the “best” matching region are transmitted per each macroblock (the regions are 16x16 pixels), each per block in a macroblock. The two functions are doing the same by their respective kind of blocks.
Firstly the code distinguishes between the different modes (inter or intra). If the macroblock is coded as an intra macroblock, then the process only consists on copying the values from the current frame to the reference frame.
Otherwise, if the macroblock is transmitted as inter mode, the process is more complicated. It is necessary to know if the motion vector coordinates correspond with an entire position or with half-pixel position. Each entire pixel position is considered to be equivalent to 2 units whereas a half-pixel position is 1 unit. Therefore if the coordinates

received are not multiple of 2, half-pixel motion estimation, the pixel interpolation must be carried out.

The original code has 2 main loops for the inter macroblocks. They distinguish if the macroblocks need vertical interpolation (see Figure 27 in Motion Estimation chapter) or not. In order to distinguish for those blocks where the horizontal interpolation is needed to obtain the correct values, there is an if statements inside each loop.

Once the correct values are obtained 3 things are made:

- Subtract the current frame with the obtained values to obtain the *"error macro block"*, which after being codified is sent to decoder.
- Calculate the Sum of Absolute Differences (SAD) between the current macroblock and the estimated one. The SADs are used in texture coding block (see texture coding for more details).
- Store the reference values in a macroblock array. The reason to store the values is that they are necessary in order to *"construct"* the reference frames (see texture update chapter).

Figure 33 depicts schematically how the code flows in the initial implementation. Actually it is more complicated (more variables…) but it shows how the code operates. The code is exactly the same for the luminance(Y) and for the chrominance (UV) blocks with the only difference that the values are taken from the UV frames.

```
for (Y blocks in MB)
 if (mode==INTRA)
       for (rows in block)
          for (columns in block)
             errorMB = CurrentFramePixel
 else if (NO vertical interpolation)
          for (rows in block)
             for (columns in block)
                if (horizontal interpolation)
                   refPixel = Calculate horizontal interpolation value
                 else
                   refPixel = Read value
                endif

                MB_TextureUpdate = refPixel;
                errorMB = CurrentframePixel – refPixel;
                SAD += absolute(CurrentframePixel – refPixel);
        else
          for (rows in block)
             for (columns in block)
                if (horizontal interpolation)
                   refPixel = Calculate V&H interpolation value
                else
                   refPixel = Calculate vertical interpolation value
                endif
```

Figure 33 Motion Compensation schematic code

8.2 Final Code

The Motion Compensation block code has suffered so many transformations in order to reduce the total amount of cycles to make the calculations. During the evolution process to the final code, many different options have tested. compares some these options.
The final code maintains the structure but making transformations in loops to find the best option. There are also two functions (one for Y blocks and one for UV blocks). For the Y blocks, and for UV blocks as well, in intra mode the copy process is carried out in all blocks together (loop unrolled and of course coalescing previously rows and columns).
Then the code is different for the two kinds of blocks. The great difference between the final and the initial code is the way to calculate the interpolated values. In the final code the interpolated values are not calculated again (average of 2 or 4 pixels) because they have been already calculated in the motion estimation block. Therefore a new array where these values are stored is passed to the function. However this is only possible for the Y blocks, as the motion estimation is only executed for this blocks.
The code for the inter-Y blocks only distinguishes whether the interpolation is needed or not. To select the correct option another parameter is passed to the function. This parameter is used to indicate which half-pixel neighbor from the entire position has been choosen. If the parameter is 0 no interpolation is required. This means that the values are read from the reference frame whereas the others are read from the interpolation array. The interpolation array allows reducing the cycles because it avoid to read more times from the memory and to make again the average. On the other hand more bytes (2040 bytes) are stored to the memory.

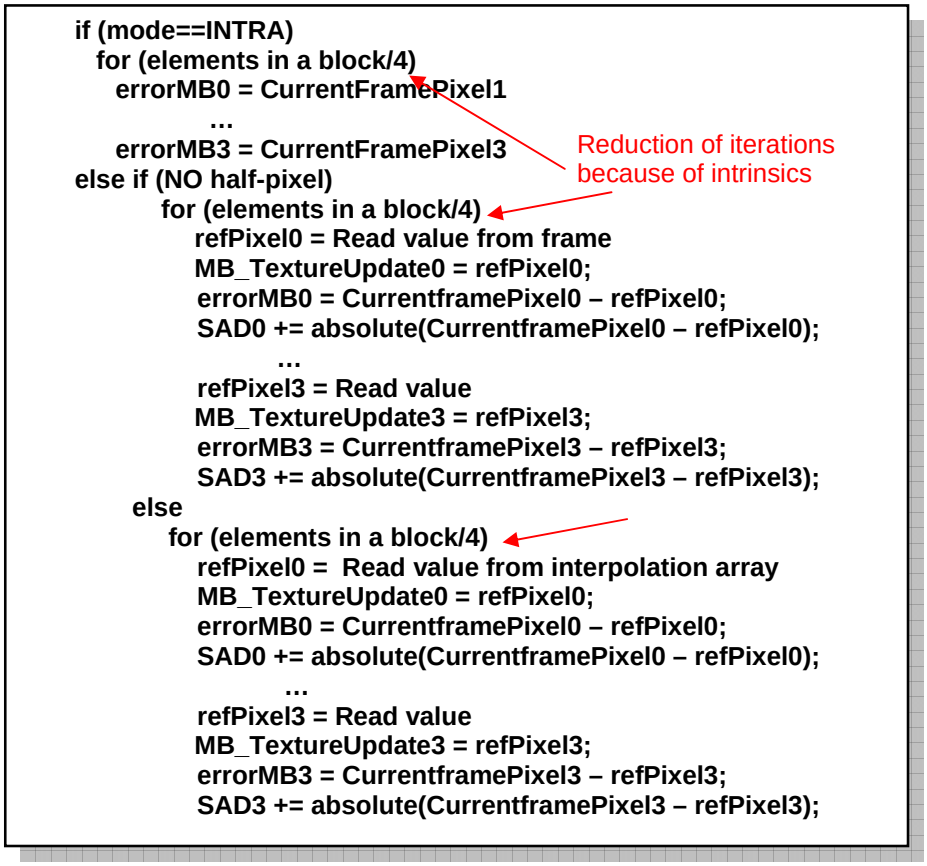


Figure 34 Final schematic code for Y-blocks

For the inter-UV blocks, the process to know if the interpolation is required remains the same as in the initial code (see initial code section). But then there are not if conditions inside the loops. There are 4 loops depending on the kind of interpolation:

1. No interpolation loop
2. Only horizontal interpolation is required.
3. Only vertical interpolation is required.
4. Horizontal/vertical interpolation is required.

On one hand, with this way it is possible to reduce cycles by removing if statements inside the loops. On the other hand the size code is much larger.

All the loops, either Y or UV blocks, are unrolled. That means that in case of Y blocks all 4 error blocks, 4 SADs… are calculated together and for the UV all 2 blocks are carried out at the same time.

A final important point is the application of intrinsics and the consequent reduction of cycles.

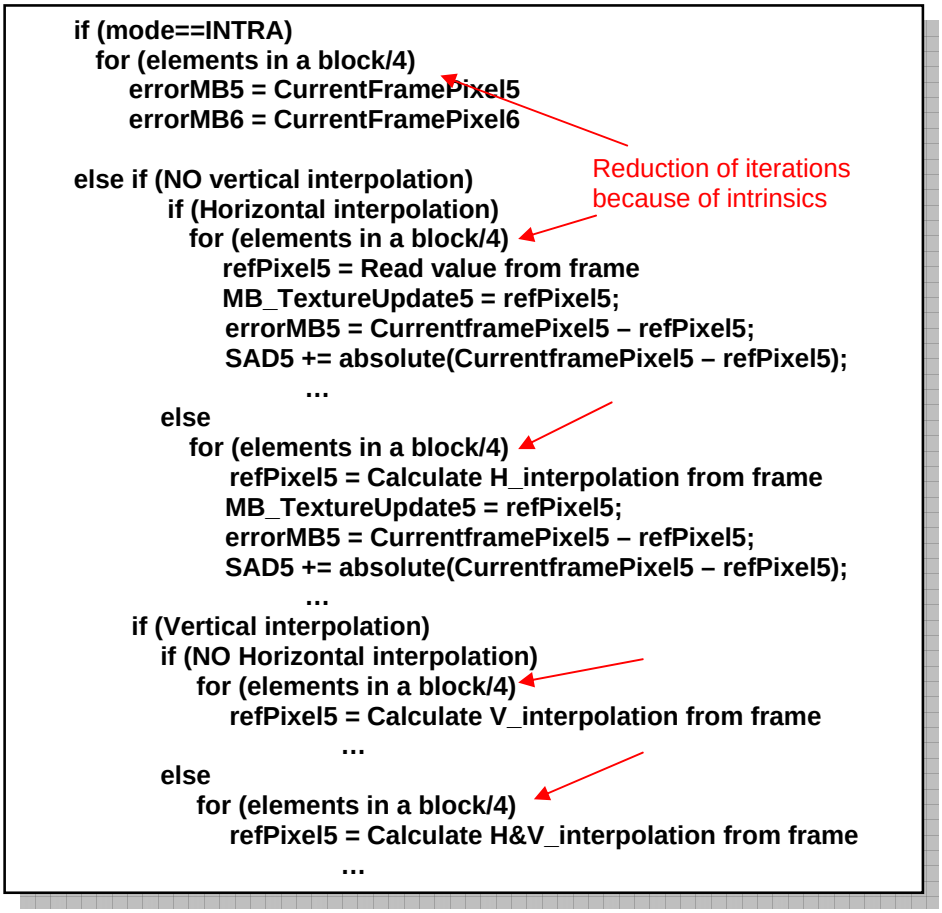
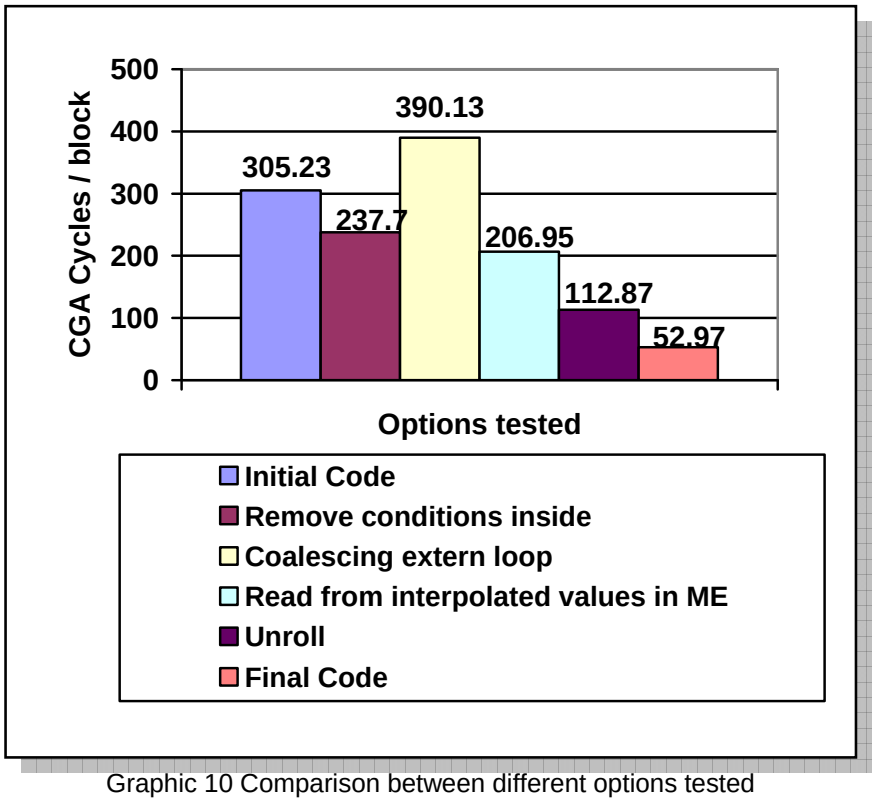


Figure 35 Final schematic code for UV blocks

The Graphic 10 compares CGA cycles for different options tested in Y-blocks. From left to right here are the options explained:

- Initial code: it is the starting code, only coalescing the rows-columns loops, but maintaining the rest of code.
- Remove conditions inside: in the original code there were if statements inside loops to select if the horizontal interpolation is necessary. Here the horizontal interpolation is always carried out when the vertical interpolation is not needed. When the interpolation vertical has to be done 2 loops were created to avoid conditions inside, because there were rounding errors.
- Coalescing external loop: it means increasing the iterations from 64 to 4x64 (pass to a MB level).
- Read from interpolated values in ME: instead of calculating the interpolated values, these are read from Motion Estimation.
- Unroll: in 64 iterations the 4-Y blocks are performed.
- Final code: change the code in order to introduce intrinsic functions, reducing iterations from 64 to 16.



Graphic 10 Comparison between different options tested

Concerning to the UV-blocks the steps followed are the same than with the Y-blocks, with the big difference that it is not possible to read already calculated interpolated values from ME block. Therefore there are not big differences in CGA Cycles per block comparing with the Y-blocks.

Table 19 shows the scheduling results for all loops in Motion Compensation block.

Kernel	*NrCycles*	*SD*	*II*	*PS*	*IPC*	*Length*
Y-blocks Intra	125	73,96%	6	5	11,83	28
Y-blocks No Interpolation	213	80,11%	11	4	12,82	36
Y-blocks Interpolation	213	80,11%	11	4	12,82	36
UV-blocks Intra	76	93,75%	3	9	15	27
UV-blocks No Interpolation	128	86,44%	6	6	13,83	31
UV-blocks H-Interpolation	149	91,96%	7	6	14,71	36
UV-blocks V-Interpolation	166	83,59%	8	5	13,38	37
UV-blocks V&H-Interpolation	251	82,22%	13	4	13,16	42

Table 19 Motion Compensation scheduling results

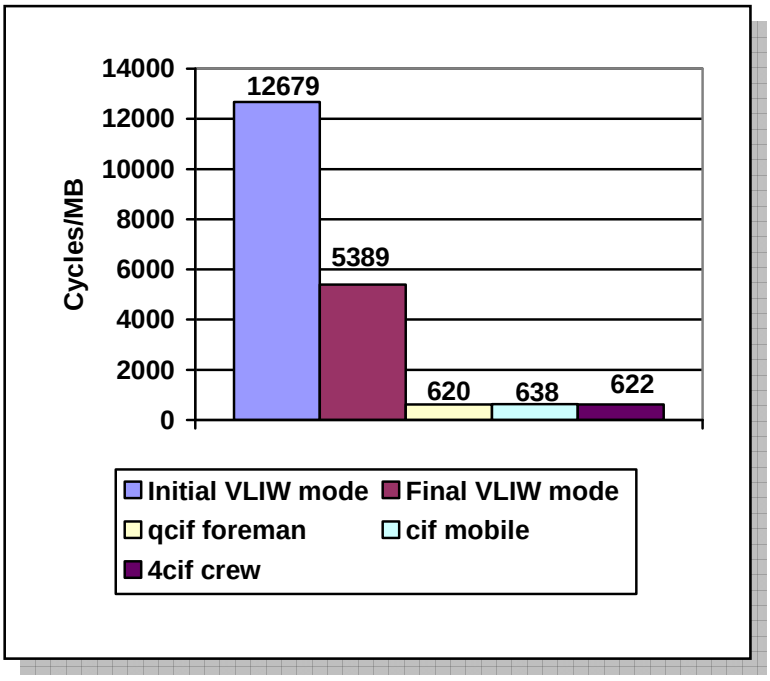
Notice that the number of cycles column is not cycles per block. Therefore, it is not possible to compare between the results from Y-blocks and results from UV-blocks.
It is also important to mark that the 2 inter Y-blocks loops have the same number of cycles (213). That means there is not difference between interpolate or not. In contrast, inter UV-blocks the difference between interpolated and no-interpolated blocks is at least 21 cycles.

8.3 General statistics

The total amount of cycles per MB spent in the entire Motion Compensation block, which means adding the different loops and the control code for difference sequence is:

- For a foreman qcif sequence: 620,32 cycles
- For a mobile cif sequence: 638,55 cycles
- For a crew 4cif sequence: 622,40 cycles

Graphic 11 MC comparison between different sequences and CGA/VLIW modes compares these sequences with original and final code in VLIW mode.



Graphic 11 MC comparison between different sequences and CGA/VLIW modes

Like in the other previous chapters here follows a simple calculation of the maximum number of cycles required to be able to have HDTV motion estimation operating realtime on a single ADRES:

- Cycles per MB = 300.000.000 / 108.000 = 2.778
- Current number of Motion Compensation cycles per MB = **620,32**

The necessary clock-speed required for a single ADRES given the current number of cycles:

- Number of Motion Compensation cycles per second = 620,32 * 108.000 =
  = 66.995.087,27 = **66,99MHz**

Chapter 9
Conclusions and future work

The conclusions can be divided in 3 different parts: regarding to results achieved, concerning methodology to optimize code, for mapping kernels, reading at memories, etcetera and finally with reference to ADRES and used tools.

9.1 Results achieved

As it was expected, having a look to the general results (Graphic 12) and the results showed during the different chapters, it is possible to appreciate that the implemented code decreases the original implemented code. The magnitude of speed-up depends on the part of code:

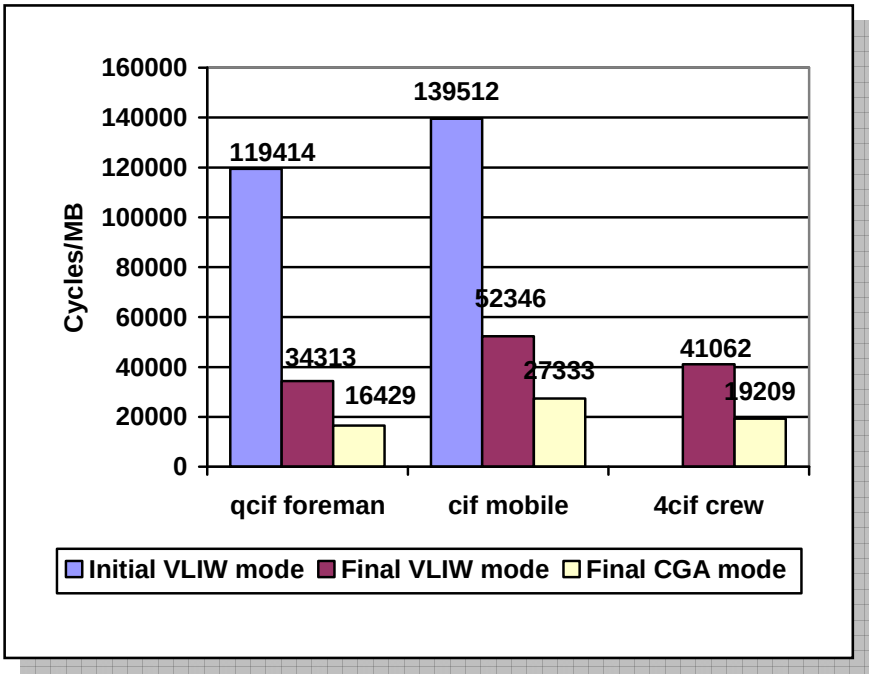
- Texture Coding is 2,17.
- Motion Estimation is 4,52.
- Motion Compensation is 2,03.
- Texture Update 2,35.

Whereas the total application speed-up is 3,48. When the code is mapped in the CGA the VLC encoder block acquires more weight in the total code, it passes from a 4,57% in the original code in VLIW mode to a 29,29% in the final code in CGA mode (the main part). Therefore, it should be recommendable to map in the CGA the VLC encoder loops.

Mapping all applications in one single ADRES for a foreman 300 frames qcif sequence these numbers are obtained:

- Current total number of cycles per 300 QCIF frames = 487.946.546
- Total number of cycles per QCIF frame = 487.946.546 / 300 = 1.626.488,48
- Total number of cycles per MB = 1.626.488,48/ 99 = 16429,17
- Total number of cycles per second required for HDTV = 16.429,17 * 108.000 =1.774.351.076,36 = **1,77 GHz**

This frequency is still far from the ADRES clock (300MHz),but if the different blocks are mapped in different ADRES then the results are close to be achieved.



Graphic 12 Cycles/MB Full MPEG-4 encoder

9.2 Code analysis

With the experience acquired and the results obtained during the optimization code I see that one of the main gain in terms of reducing cycles is with the use of intrinsic functions. In IDCTrows and Q_invQ (Texture coding block) cycles did not decrease so much using intrinsic function. This is because in these particular functions, intrinsics are used to avoid if statements, but no for taking advantage of reading integers instead of shorts, 4-bytes instead of 1-byte. In contrast in Motion Estimation, Texture Update and Motion Compensation blocks it is possible to reduce rather the amount of cycles. E.g. in ME interpolation loop, the cycles pass from 1.474 cycles to 346 cycles. The loss of cycles is higher even if it is necessary to add more operations to obtain the correct code (i.e. packing, unpacking…) or to add some if statements, like in the Texture Update block where the cycles in the main loop decrease from 738 to 361. In this loop the II pass from 11 to 20, but the there are only 16 iterations instead of the 64. Therefore intrinsic functions always are something to take into account when you are optimizing a code.

Another important thing to consider is try to carry out the calculations for all blocks together in the same loop. What it means is to do all the calculations with the same number of iterations. Unfortunately this is not good or not possible in all the occasions. For example in texture coding block is not possible to calculate 6 times in one loop the DCT over rows because there are too many operations and it would be necessary a lot of new variables to store… On the other hand, to calculate 9 SADs, to perform the

Texture Update (6 blocks) or Motion Compensation (4 blocks), all are carried out in the same loop without increase the iterations. Therefore this is like unroll the loop.
Coalescing loops is a technique extensively applied in the code, especially for mapping rows-columns loops. This is always a good technique to apply wit columns and rows to increase the iterations mapped in the CGA without increasing the II. But when the loop to coalesce is the macroblock loop (to go over the 6 blocks for example), then it is necessary to consider some things. As it is always necessary to add some control instructions (e.g. for reading correct values from the memory), sometimes it is not possible to maintain the II because there are not enough available resources in the CGA (e.g. more multipliers are needed or it is necessary to store more variables… ResMII<RecMII) and then the advantage to reduce 6 times the length is not worth (illustrated in Figure 36). Coalescing the loops there are only 1 prologue-epilogue (length) instead of 6 (if we are talking about coalescing with the MB loop), but our II has been increased. Therefore, before coalesce loops, it is necessary to evaluate, if it is possible to maintain the II, the number of iterations that we are going to map…

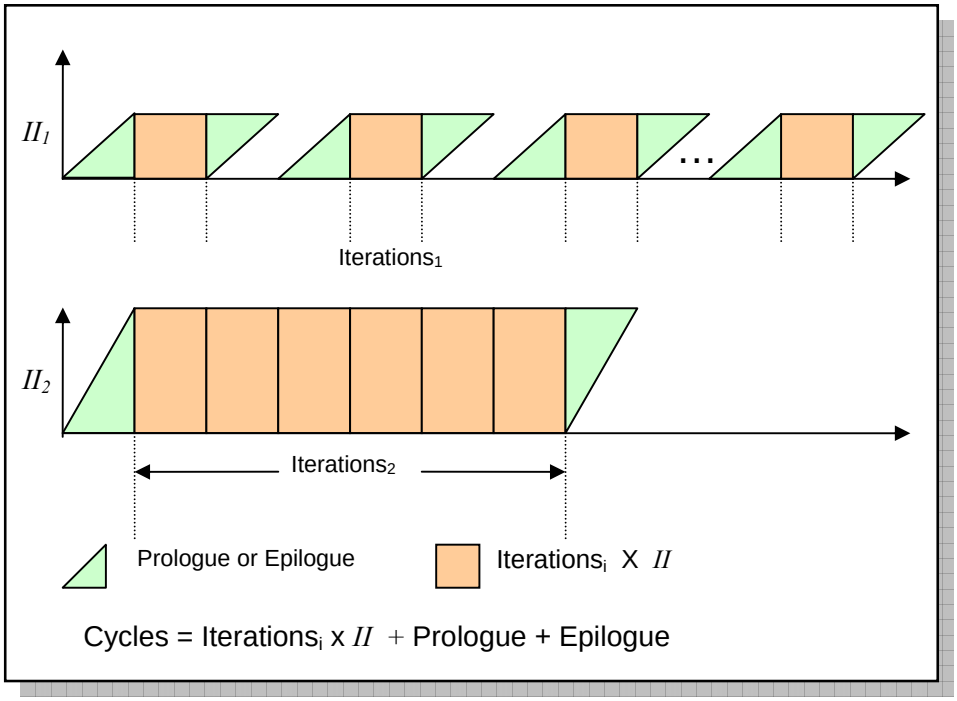


Figure 36 Total cycles coalescing or not coalescing loops

Another important thing (already explained in Texture Coding chapter) is the way to read at memory. Whenever it could be possible, it is better to read integers (4-bytes) instead of shorts or another 1-byte type, if no much operations has to be added later (see Texture Coding chapter).

To talk more concretely about specific code optimizations here is a list of top code optimizations:

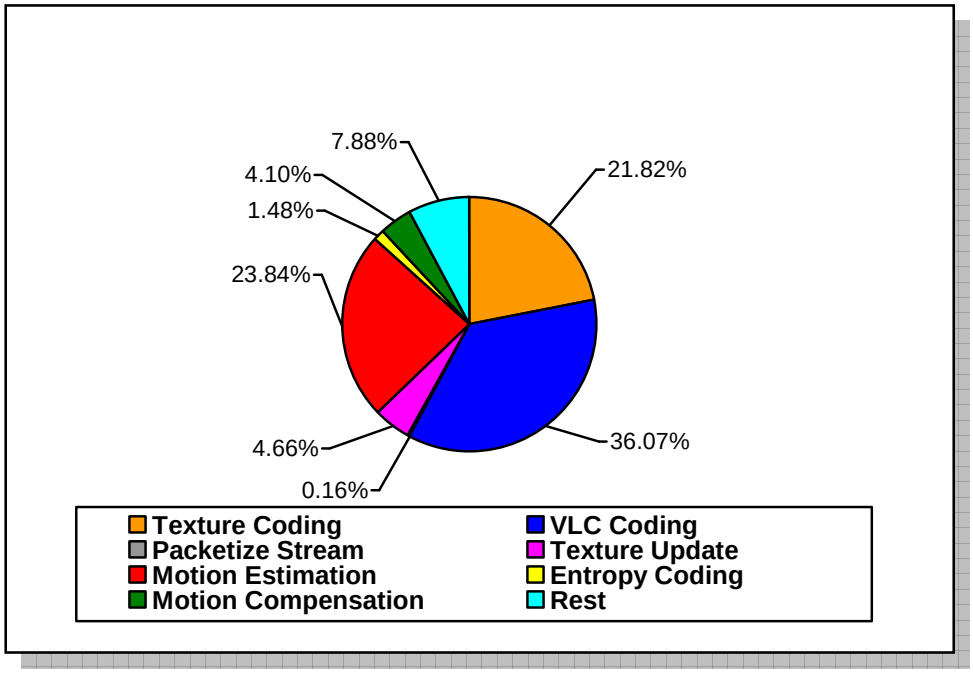
- General use of intrinsic functions. In general the main gain in cycles is produced calculating integer operations instead of short operations by the use of existing intrinsic functions, even if the code adds more complexity, because intrinsic allow us to reduce iterations.
- Use of intrinsics to calculate interpolated values for the half-pel ME.
- Calculate the 9 SADs for the first spiral in full-pel ME.
- Coalesce the loops in TC block in order to go over MBs instead over rows or columns.
- Avoid conditions in TU block and unroll the loop to Update all blocks together.

9.3 About ADRES and its compiler

The uses of new intrinsic functions are very useful to decrease cycles but sometimes it is necessary to apply several intrinsics to make a single operation. For example to perform interpolation, many different intrinsics are applied. It could be possible to create a new intrinsic to calculate faster the interpolation. In Texture Update block and Motion Compensation it is necessary to read some *unsigned char* as *integers.* There is not an intrinsic function to do it; therefore it is necessary to make it in different steps.
Concerning to DRESC compiler, it is very useful to exploit loop-level parallelism, because it makes some automatic optimizations, such as reduce the prologue-epilogue length or the possibility to know which alap factor give us the best results. On the other hand it should be useful to incorporate in DRESC automatic tools that allow the programmer to roll unroll the loops automatically.

9.4 Future work

Once the different blocks (Texture Coding, Motion Estimation, Motion Compensation and Texture Update) are mapped in the CGA, the total number of cycles decrease, and consequently, the percentages of code spent in each block change too. New percentages (Graphic 13) shows us that now VLCencoder block has more important weight, therefore, this block should be useful to map this block in the CGA too.



Graphic 13 Cycles percentages spent in different parts of MPEG-4 encoder (CGA mode, final code)

In Motion Estimation the use of intrinsic functions is generalized. It is necessary to read integer values instead of short or unsigned char values. For this reason there are several memory accesses that are unaligned. It is necessary to make some modifications in the final code to avoid this. Modifications consist on read 2 values aligned and then pack the correct values from the 2 integers in another one to operate correctly. This will add more complexity and some more cycles to the code.

Appendix A

Intrinsics

Here it follows more specific description about code with intrinsics implemented in Motion Estimation, Motion Compensation and Texture Update blocks.

Motion Estimation

This is the process to calculate the 4 interpolated values in the Half-Pixel Motion Estimation function:

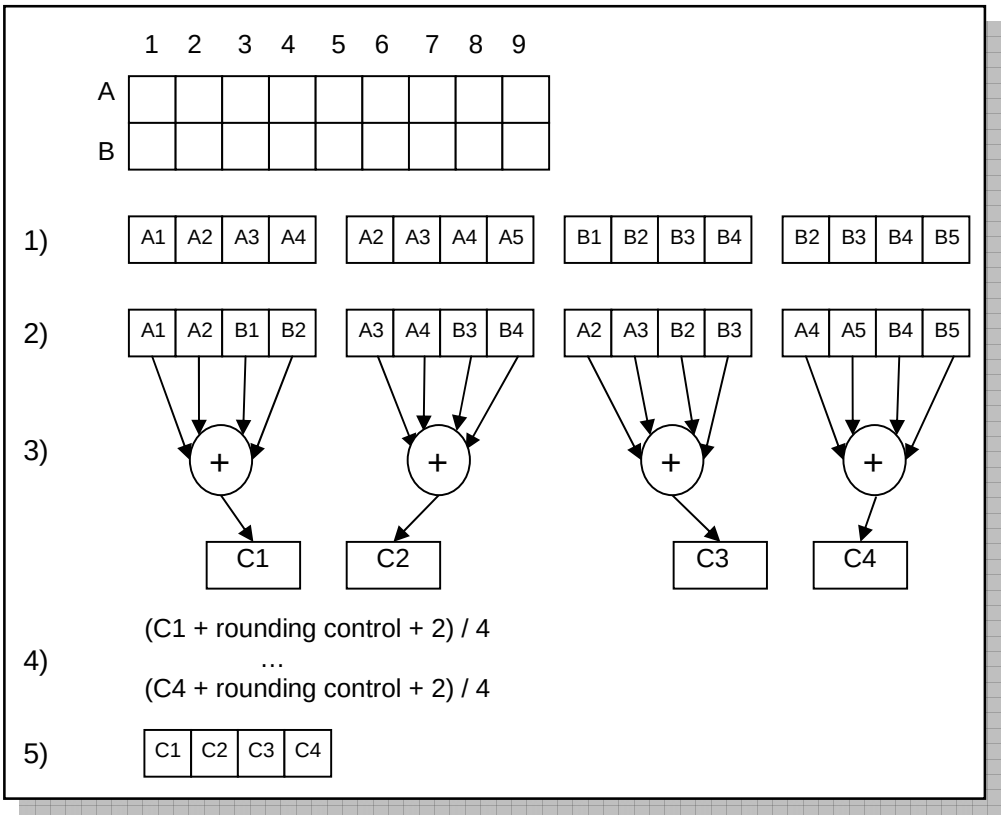


Figure 37 intrinsic process to make interpolation in Half-Pixel ME

The coefficients to read are type short (8-bit), therefore it is possible to use a set of intrinsic functions to use 32-bit operations.

1) 4 integers are read in each iteration with an offset of 1 position right or down.

2) Using the PACK2 intrinsic function it is possible to obtain 4 integers with the correct position to perform the average (see matrix).

3) INNERSUM4 intrinsic is applied, in order to sum the 4 shorts packed inside the integer (C1, C2, C3, C4).

4) The number is averaged.

5) Finally the 4 numbers obtained are packed with the PACK4 intrinsic and stored in the array.

Here follows the code:

```
4interpolated =
=PACK4((I_INNERSUM4(I_PACK2(pix00,pix10),0)+2-rounding_control)/4,
        (I_INNERSUM4(I_PACK2(pix01,pix11),0)+2-rounding_control)/4,
        (I_INNERSUM4(I_PACK2(pix00>>16,pix10>>16),0)+2-rounding_control)/4,
        (I_INNERSUM4(I_PACK2(pix01>>16,pix11>>16),0)+2-rounding_control)/4);

Vinterpolated = (rounding_control==0)?I_AVGU4(pix01,pix11):((pix01 & pix11) +
(((pix01 ^ pix11) & ~BYTE_VEC32(0x01)) >> 1));

Hinterpolated = (rounding_control==0)?I_AVGU4(pix10,pix11):((pix10 & pix11) +
(((pix10 ^ pix11) & ~BYTE_VEC32(0x01)) >> 1));
```

Figure 38 Intrinsic code in Half-Pixel Motion Estimation

To calculate horizontal and vertical interpolated values it is only necessary to read the correct values and apply the AVGU4 intrinsic function. In this case if rounding control it is not 0 is necessary to make an OR operation between the 2 integers read and one AND with a 01010101 vector in order to make correctly the integer division.

Motion Compensation

Here it follows the intrinsic process in Motion Compensation:

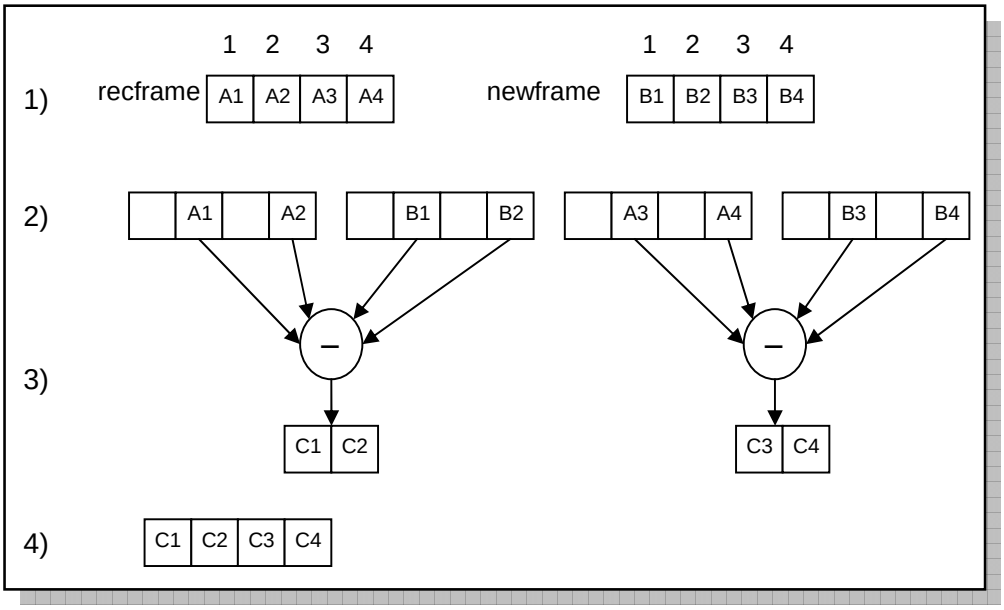


Figure 39 Intrinsic process in MC

1) First it is necessary to read the values as integer, from the current frame and from the reference frame.

2) Then using the PACK2 function and SHLMB the LSB and the MSB part of the integers are split in 2 new integers.

3) Then using the SUB2 function the new integers from the current frame and from the reference frame are subtracted.

4) Finally the integers are stored in the correct place.

This figure shows the code:

```
recPix0 = *(unsigned int*)&recYframe[prevRecFrameIdx][yu+j][xl+i];
newPix0 = *(unsigned int*)&newFrameY[Off_Row+j][Off_Col+i];

recPixA0 = I_PACK2(I_SHLMB(recPix0<<16,0), recPix0&0xff);
recPixB0 = I_PACK2(I_SHLMB(recPix0,0), I_SHLMB(recPix0<<8,0));
newPixA0 = I_PACK2(I_SHLMB(newPix0<<16,0), newPix0&0xff);
newPixB0 = I_PACK2(I_SHLMB(newPix0,0), I_SHLMB(newPix0<<8,0));

partA0 = I_SUB2(newPixA0, recPixA0);
partB0 = I_SUB2(newPixB0, recPixB0);

*(int*)&errorMacroBlock[0][j*8+i+0] = partA0;
*(int*)&errorMacroBlock[0][j*8+i+2] = partB0;
```

Figure 40 Motion Compensation Intrinsic Code

Texture Update

For the Texture Update there are 2 different code depending on the block type. If the block is type 3 the code is different (see Texture Update chapter).

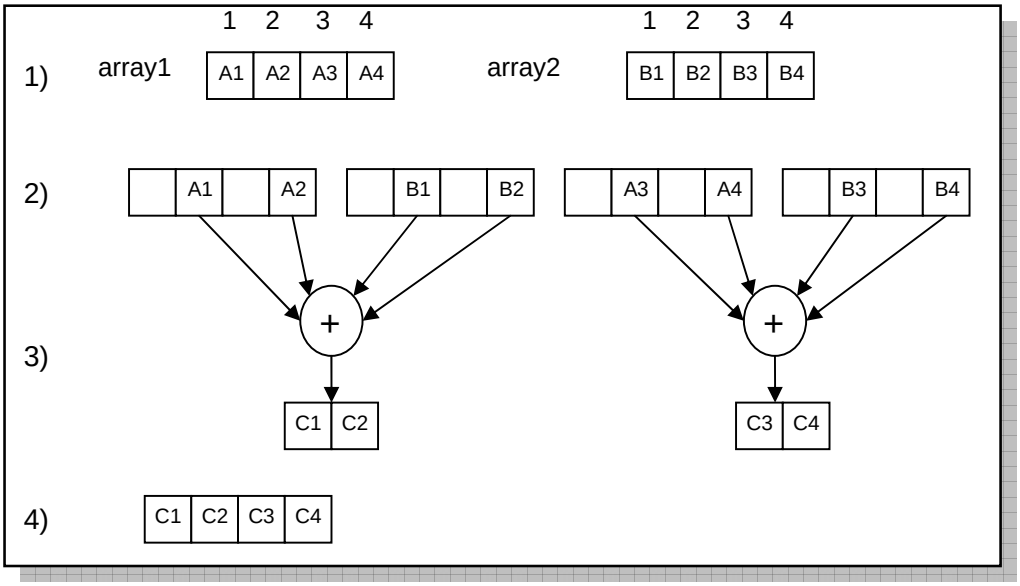


Figure 41 Intrinsic process for TU block

1) First the correct integer values are read from the arrays.

2) Then using the PACK2 function and SHLMB the LSB and the MSB part of the integers are split in 2 new integers.

3) Then using the ADD2 function the new integers from the 2 arrays are added.

4) Finally using the function SPACK4 the values are packed in one integer and stored in the correct place.

If the block is type 3 the it is only used the positions A1 and A2. These positions are copied and packed in a integer and then the process remains the same.

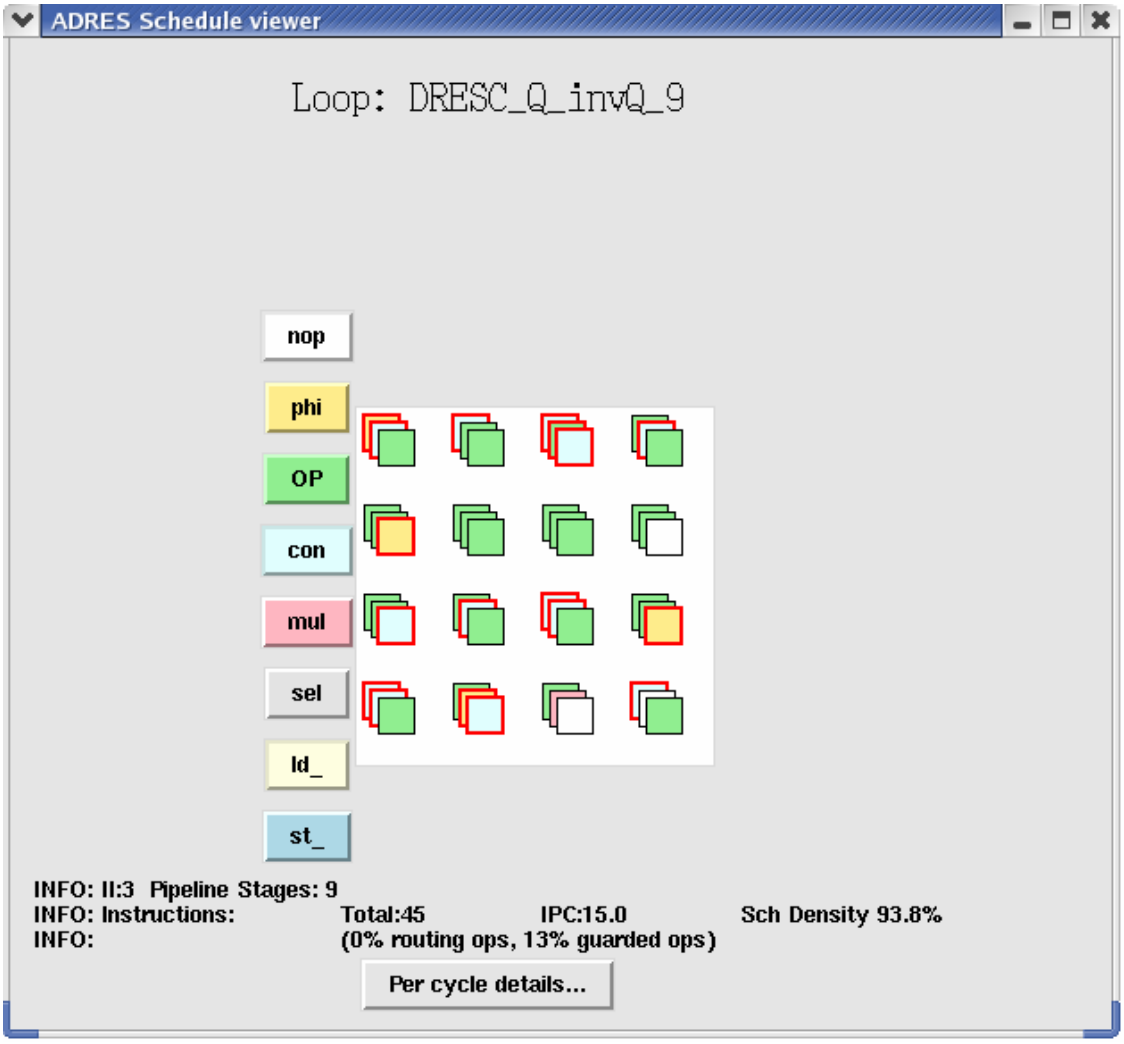
```
    PixelA0 = *(unsigned int*)&(TmpUChar[0][d0*j+i*e0]);
    PixelB0 = *(int*)&(TmpShort[0][f0*j+i*g0]);
    PixelE0 = *(int*)&(TmpShort[0][f0*j+i*g0+2]);

    PixelC0 = I_PACK2(I_SHLMB(PixelA0<<16,0), PixelA0&0xff);
    PixelD0 = I_PACK2(I_SHLMB(PixelA0,0), I_SHLMB(PixelA0<<8,0));
    if (textureUpdateMode[0]==3)
        {
           PixelF0 = I_PACK2(PixelB0,PixelB0);
           partA0 = I_ADD2(PixelC0, PixelF0);
           partB0 = I_ADD2(PixelD0, PixelF0);
         }
     else
        {
           partA0 = I_ADD2(PixelC0, PixelB0);
           partB0 = I_ADD2(PixelD0, PixelE0);
         }
    rec_pix0 = I_SPACKU4(partB0, partA0);
     *(unsigned
int*)&recYframe[recFrameIdx][YPADDING+v*16+j][YPADDING+h*16+i] =
rec_pix0;
```

Figure 42 Texture Update Intrinsic Code

Appendix B

Example of scheduled kernel



```
DRESC_Q_invQ - loop id:9 (II=3, Stages=9)
============================ Context 0 ============================
rphicon12_p st_i_pcon12_p subcon12_p lsl
lsl       asr       pred_lt    asr
pred_lt    add       st_c2_pcon12_p lsl
mulcon12_p intr19_gp  add       addcon12_p
============================ Context 1 ============================
con12_p    lsl       pred_gtstart_ctrl_p movcon12_p
add_u      and       add_u      lsl
pred_eq    lslcon12_p lslcon12_p mov
divcon12_p rphicon12_p mul        ...
============================ Context 2 ============================
pred_eq    eq        ld_c2_pcon12_p add
phicon12_p mov       sub       ...
st_c2_pcon12_p add       and       rphicon12_p
pred_ne    add_ucon12_p ...       mov
Operation            Cont:0   Cont:1   Cont:2    Loop Total
=================================================
     add:              3        0         2         5
   add_u:              0        2         1         3
     and:              0        1         1         2
     asr:              2        0         0         2
 con12_p:              0        1         0         1
     div:              0        1         0         1
      eq:              0        0         1         1
intr19_gp:             1        0         0         1
 ld_c2_p:              0        0         1         1
     lsl:              3        4         0         7
     mov:              0        2         2         4
     mul:              1        1         0         2
     phi:              0        0         1         1
 pred_eq:              0        1         1         2
pred_gtstart_ctrl_p:   0        1         0         1
 pred_lt:              2        0         0         2
 pred_ne:              0        0         1         1
    rphi:              1        1         1         3
 st_c2_p:              1        0         1         2
  st_i_p:              1        0         0         1
     sub:              1        0         1         2
=================================================
Total OPs:             16       15        14        45
  % array:          (100%)   (94%)     (88%)     (94%)
(routing):*         -0       -0        -0        -0
Real OPs:           16       15        14        45
 % array:           (100%)   (94%)     (88%)     (94%)

INFO: II:3  Pipeline Stages: 9
INFO: Instructions:          Total:45          IPC:15.0          Sch Density 93.8%
INFO:                        (0% routing ops, 13% guarded ops)
```

Bibliography

[1] R. Hartenstein. *A decade of reconfigurable computing: a visionary retrospective*, in Proc. of Design, Automation and Test in Europe (DATE), pages 642-649, 2001.
[2] Stephen Brown and Jonathan Rose. *Architecture of FPGAs and CPLDs: A Tutorial.* IEEE Design and Test of Computers, Vol. 13, No. 2, pp. 42-57, 1996.
[3] H. Corporaal. *Microprocessor Architecture: from VLIW to TTA*. John Wiley & Sons,1998.
[4] B. Mei. *A coarse-grained reconfigurable architecture template and its compilation techniques.* PhD thesis, Katholieke Universiteit Leuven-Belgium, 2005.
[5] B. Mei, S.Vernalde, D.Verkest, Hugo De Man, Rudy Lauwereins. *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures*. IMEC, Leuven-Belgium Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference pp166- 173, 2002.
[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. *IMPACT: An architectural framework for multiple-instruction-issue processors*, in Proceedings of the 18th International Symposium on Computer Architecture (ISCA), pp. 266–275, 1991.
[7] B. R. Rau, Iterative modulo scheduling, tech. rep., Hewlett- Packard Lab: HPL-94-115, 1995.
[8] 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.
[9] I.E. Richardson. *H.264 and MPEG-4 Video Compression Video Coding for next-generation multimedia. John Wiley & Sons, 2003.*
[10] F. Pereira, T. Ebrahimi. MPEG-4 The book. Pretince Hall PTR IMSC Multimedia series, 2002.
[11] 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 Proc. of International Symposium on Microarchitecture (MICRO), pages 45-54,1992.
[12] C. De Vleeschouwer, T. Nilsson, K. Denolf, and J. Bormans. *Algorithmic and Architectural Co-Design of a Motion-Estimation Engine for Low-Power Video Devices* IEEE Trans. on Circuits and Systems for Video Technology, vol. 12, Nº 12, December 2002 pp 1093-1105