

# Scalable MPEG-4 Encoder on FPGA Multiprocessor SOC

**Ari Kulmala, Olli Lehtoranta, Timo D. Hämmäläinen, and Marko Hännikäinen**

*Department of Information Technology, Institute of Digital and Computer Systems, Tampere University of Technology, P.O. Box 553, Korkeakoulunkatu 1, 33101 Tampere, Finland*

Received 15 December 2005; Revised 16 May 2006; Accepted 27 June 2006

High computational requirements combined with rapidly evolving video coding algorithms and standards are a great challenge for contemporary encoder implementations. Rapid specification changes prefer full programmability and configurability both for software and hardware. This paper presents a novel scalable MPEG-4 video encoder on an FPGA-based multiprocessor system-on-chip (MPSOC). The MPSOC architecture is truly scalable and is based on a vendor-independent intellectual property (IP) block interconnection network. The scalability in video encoding is achieved by spatial parallelization where images are divided to horizontal slices. A case design is presented with up to four synthesized processors on an Altera Stratix 1S40 device. A truly portable ANSI-C implementation that supports an arbitrary number of processors gives 11 QCIF frames/s at 50 MHz without processor specific optimizations. The parallelization efficiency is 97% for two processors and 93% with three. The FPGA utilization is 70%, requiring 28 797 logic elements. The implementation effort is significantly lower compared to traditional multiprocessor implementations.

Copyright © 2006 Ari Kulmala et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

## 1. INTRODUCTION

Video is becoming an essential part of embedded multimedia terminals. There are, however, many contradicting constraints in video codec implementations. One challenge is the rapid evolution of compression standards with several different algorithms. This requires programmability that is easy to achieve with processor-based platforms. However, achieving the best power, energy, and silicon area efficiency requires custom hardware implementations. On the other hand, hardware (HW) design is more demanding than software (SW) development, and modifications are very expensive and time consuming. For example, nonrecurring engineering (NRE) costs, especially photo mask fabrication costs, increase rapidly with each technology generation making frequent HW upgrades less favorable. Software only implementation solves the flexibility and upgradeability problem but is not an optimal solution from a performance versus silicon area point of view.

The work in this paper solves the HW video codec design flexibility and upgradeability problem with a fully programmable, scalable MPSOC approach [1, 2]. The key idea is to use synthesizable soft-core processors and a synthesizable system-on-chip (SOC) interconnection network, which allows prototyping and implementation on any FPGA platform, or in an ASIC technology with a rapid design cycle. In addition, our implementation framework enables a seamless

trade-off between performance and area without creating an extra burden in system design by scaling the number of identical processors. Furthermore, the architecture is designed to be easily reusable for any kind of application.

A data parallel MPEG-4 simple profile (SP) software encoder is implemented on the MPSOC to demonstrate the effectiveness and scalability of the presented solution. The video images are divided into independent horizontal slices which are mapped to different processors for encoding. A master-slave configuration is used where the master processor is responsible for overall control and the slaves perform the video encoding.

In this paper, we use an Altera Stratix FPGA as the target platform [3], Altera Nios processors, and our heterogeneous IP block interconnection v.2 (HIBI) [4] as the communication network. No MPEG-4 specific HW accelerators, for example, for motion estimation, are currently used, but HIBI provides a very convenient plug-and-play method to add intellectual property (IP) blocks independent of the vendor.

Topics of interest in this paper include practical implementation issues, such as utilized FPGA resources and achieved performance, design cycle improvement, scalability, and encoder specific issues like memory optimization due to scarce on-chip memories. The implementation works in practice with an FPGA board attached to a PC that sends source video streams and receives compressed data.

This paper is organized as follows. Related work is reviewed in Section 2. The MPSOC architecture is described in Section 3. The video encoding software and our encoder parallelization approach are presented in Section 4. Section 5 explains the integration of the HW architecture and software. In Section 6 the results are presented. Finally, Section 7 summarizes the paper and discusses future work.

## 2. RELATED WORK

In this section we consider the related work in two categories, parallel video encoding and FPGA-based MPSOC architectures.

### 2.1. Parallel encoder implementations

Due to high computational complexity of video encoding [5], several parallel solutions have been developed in contrast to traditional sequential program flow [6]. There are at least four general parallelization methods used: functional, temporal, data, and video-object parallelism. For functional parallelism [7] different functions, such as DCT and motion estimation, are connected in a functional pipeline to be executed in parallel by different processing units. However, scaling a functional parallel application requires a lot of work (high scaling effort). When each processor executes a specific function, adding or removing processors requires a whole system redesign to balance the computational load in the pipeline. For temporal parallelism (i.e., parallel in time) a full frame is assigned to every CPU. The scalability of this style is high. However, as the number of parallel encoders increase, it introduces a significant latency in encoding, since one frame is buffered for each encoding CPU. Works in this category include [8–10]. In data parallelism, the image is divided into slices that are assigned to different CPUs. The slices are encoded in parallel frame-by-frame. This approach is used in [11–13]. For video-object parallelism, which is specific to MPEG-4, arbitrary sized shapes referred to as video-objects in the image are assigned to different CPUs. The objects can be considerably unequal in size, which may lead to unbalanced execution time between different CPUs if care is not taken. Such work is presented, for example, in [14].

We are mainly interested in real-time encoding. Functional, data, and video-object parallelism are all eligible for real-time, low-latency video encoding, because they do not require frames to be buffered. We chose data parallelism because functional parallelism has a high scaling effort and video-object parallelism is strictly MPEG-4 specific. Scalability is the most feasible criterion used to compare different architectures and parallel implementations, because reported results typically vary in accuracy. The scalability of different parallelization methods is compared in Section 6.

Contemporary FPGA designs tend to use single encoder cores with HW accelerators arranged in a functional pipeline. Our implementation is one of the first utilizing multiple parallel encoders on an FPGA in a data parallel configuration. In [15], an FPGA-based H.263 encoder is demonstrated re-

quiring 400 kgates for HW accelerators while providing 30 QCIF frames/s at 12 MHz. Reference [16] presents FPGA implementations of hardware accelerators for an H.264 video codec. In [17], an H.264 coder is designed and verified with an FPGA emulator platform. An interface between a host PC and an FPGA-based MPEG-4 encoder is built in [18] enabling fast prototyping and debugging.

### 2.2. FPGA multiprocessor architectures

Although multiprocessor systems have been researched for a while, most of the work has concentrated on ASIC implementations. FPGAs have only recently grown large enough to hold such implementations, which is one reason for a low number of reported FPGA-based MPSOCs. However, two trends of research can be identified.

First, FPGA multiprocessor systems are used to develop parallel applications. In these works, the main emphasis is usually on the application and its parallelization. The hardware architectures are briefly summarized. Typical implementations rely on vendor-dependant solutions, because they are usually easy to use. The hardware restrictions to scalability or flexibility and the ease of adding or removing components are often not addressed.

In [19], Martina et al. have developed a shared memory FPGA multiprocessor system of digital signal processor (DSP) cores of their own design that run basic signal processing algorithms. The implemented bus-interconnection is not described. There are no synthesis results for the whole system, but the system runs at 89 MHz on a Xilinx XCV1000 FPGA. Wang and Zivras presented a system of six soft-core Nios processors [20]. Processors are interconnected with a multimaster Avalon bus [21]. No figures of the area required for the whole system are presented. However, the maximum clock frequency is 40 MHz using an Altera EP20K200EFC484-2x FPGA board.

Second, a hardware-oriented point of view for future multiprocessor requirements is presented. Reconfigurability is often emphasized [22]. Also, IP-block-based systems are stressed and a need for a scalable, standard interface interconnection network is anticipated. Kalte et al. [22] have presented a multilayer advanced microcontroller bus architecture (AMBA) interconnection architecture used in an FPGA. In AMBA, access to slaves is multiplexed between masters and different masters can use different peripherals simultaneously. A conceptual view of a system is depicted although not implemented. The interconnection architecture is synthesized separately, as is the processor. No application is described.

This work combines both of the above categories, since an application and a working prototype is implemented on the proposed architecture. The architecture itself is constructed of IP blocks and a general purpose interconnection architecture with support for an OCP-IP interface [23], which is a standard interconnection interface. A standardized IP block interface along with high scalability ensures the future use of the architecture.

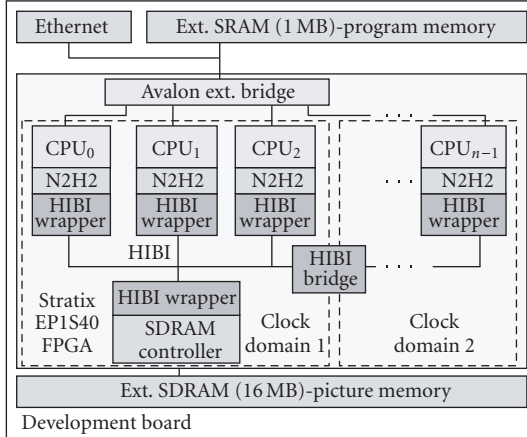


FIGURE 1: High-level view of the architecture on an FPGA development board.

### 3. MPSOC ARCHITECTURE

In this section we present our novel architecture for multi-processor systems. The key elements of the architecture are processor programmability and scalability and flexibility that are obtained with the HIBI on-chip network [4]. A high-level view of the architecture is depicted in Figure 1. It contains a parameterizable number of soft-core processors, an HIBI network, direct memory access (DMA) blocks (N2H2), and two external memories located on the development board. In this case, external SRAM memory is used for instructions for all processors. HIBI is used to access the external SDRAM memory, which is used for shared data, and to let processors to send messages directly to each other.

#### 3.1. Heterogeneous IP block interconnection v.2

HIBI is a hierarchical interconnection for SOCs, which was originally developed for ASICs. The objective of HIBI is to provide a topology-independent, scalable, yet high-performance on-chip network. It is highly configurable and supports multiple clock domains. To provide maximum efficiency, there are no empty cycles during bus transfers under a high load. Split transactions are used in read operations. Bus capacity is fully exercised by starting new transfers right after the preceding one has finished. The HIBI network can be reconfigured at run-time, using a configuration RAM. This should not be confused with FPGA reconfiguration.

The basic building block of the HIBI network is an HIBI wrapper. As HIBI utilizes distributed arbitration, each HIBI wrapper is responsible for seizing the bus at right time. The main characteristics of HIBI are presented in Table 1 [1]. In Figure 1, three processors and an SDRAM controller form one segment, Clock domain 1, and the rest of the processors form another segment, Clock domain 2. There is no master/slave configuration in the interconnection and, thus, every IP block can freely access any other IP block. An HIBI bridge is used to connect the segments together to allow

TABLE 1: Summary of HIBI characteristics.

Property	HIBI implementation
Topology	Hierarchical bus with wrappers and bridges between bus segments
Interface	FIFO, OCP
Clocking	Multiple clock domains
Arbitration	Distributed, pipelined
Arbitration algorithm	Priority, round-robin, time division multiple access (TDMA), or combination
Synthesis-time configurable parameters	FIFO sizes, data width, addresses, initial configuration, number of configuration pages and their type (RAM or ROM), included properties
Run-time configurable parameters	All arbitration parameters and algorithm, cycle counters, power mode
Quality-of-service (QoS)	TDMA, send limit + priority/round-robin, multiple priorities for data, fast reconfiguration
Bus resolution	OR-network
Addressing	Multiple addresses per IP, multiplexed in the bus with data, allows multicast

transfers between segments. Every wrapper is assigned an address space, which can vary depending on the number of wrappers and the need for different addresses per wrapper. When data is sent via HIBI, only the destination address is delivered. In order to distinguish between different sources, each source must use a separate destination address within the recipients address space.

There are several different HIBI wrapper interfaces for IP blocks. It is left to the designer to decide which interface is the most appropriate, but there may be a mix of different kinds of interfaces. For example, FIFO and OCP interfaces can be utilized in the same architecture. Also, HIBI supports multiple priorities for data. Optionally, there can be different FIFOs for every priority. In that case, the highest priority FIFOs are always treated first. Thus, they can interrupt lower priority data transfers to get service immediately.

#### 3.2. Soft-core processors

Currently, we have used soft-core processors Nios [24] and Nios II [25, 26] in our architecture. The master processor is Nios with a configuration shown in Figure 2. The peripherals are timers, LEDs button parallel I/Os (PIOs), and an UART. Nios can be used as a 16-bit or 32-bit processor, which affects the data bus width. Nios always uses 16-bit instruction words, which restricts the immediate values to five bits. With prefix-instructions this is increased to 16 bits. A large number of prefix instructions reduces code efficiency.

Nios II is a 32-bit CPU with 32-bit instructions. Nios II has more limited configurability. We use the fastest version,

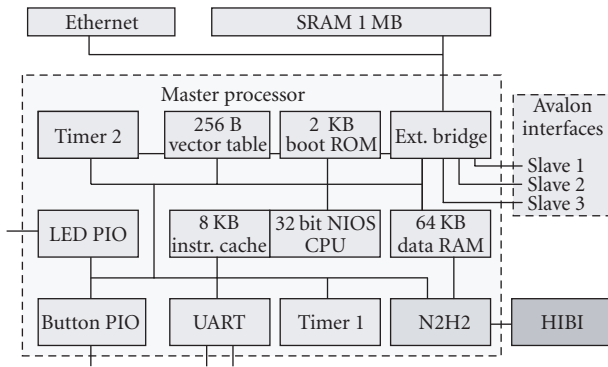


FIGURE 2: Master processor configuration.

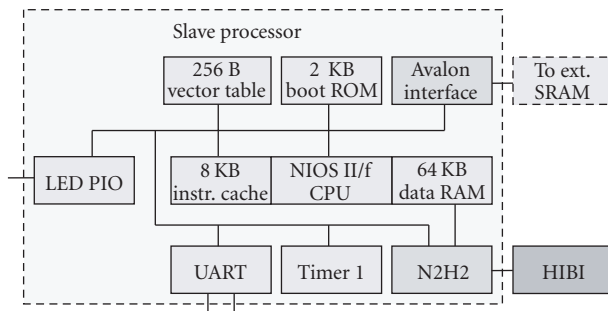


FIGURE 3: Slave processor configuration.

Nios II/f (fast) that provides 1.16 DMIPS/MHz and an area consumption of around 2000 LEs. Nios II is used for video encoding slaves with the configuration depicted in Figure 3. Unlike Nios, Nios II uses *tightly-coupled memory* between the processor and data RAM. Data does use the Avalon [21] bus that significantly reduces the memory access time.

Nios natively use the Avalon bus to connect to memories and peripherals. Avalon bus masters can command slaves, but slaves can only get the attention of a master by raising an interrupt. A typical master is a processor. A drawback is that masters cannot communicate directly with each other.

Both Nios processors have separate instruction and data buses. In Figures 2 and 3 only the data bus is drawn for simplicity. The instruction bus connects to the boot ROM, instruction memory (ext. SRAM), and vector table. With the Avalon bus, there are 20 bus master *address* lines, 32 *data* lines, and *waitrequest* and *read* signals. Data bus master has 20 *address* lines, two 32-bit *data* buses for read and write, and signals *waitrequest*, *read*, *write*, *irq* and *irq number*. This makes 92-signal lines in total. There are possibly other signal lines as well, but even with this practical minimum, there are 146-signal lines in the buses. As a comparison, 32-bit HIBI bus consists of only 38-signal lines (32-bit *data*, 3-bit *command*, *address\_valid*, *lock*, and *full*). In addition, HIBI supports interprocessor communication without restrictions. The features of Avalon are not sufficient for data intensive multiprocessor communication, motivating the use of HIBI.

### 3.3. Nios-to-HIBI v.2 DMA

Nios processors do not have a native support for the HIBI on-chip network. Therefore, a DMA block, Nios-to-HIBI v.2 (N2H2), was implemented to attach the processors to HIBI. DMA minimizes CPU intervention in transfers. This allows the CPU to execute the application while DMA transfers data on the background.

N2H2 includes three Avalon interfaces. The slave interface is used by a CPU to control and query N2H2. These configuration and status registers include the state of the DMA block, DMA transfer instructions, and DMA receiving instructions. Two master interfaces are used separately for receiving and transmitting. In order to increase the reusability, these interfaces have been isolated from the other as much as possible. Thus, with minimal modifications, the same block can be applied to different processors.

The transmitter side is fairly straightforward. First, the CPU writes the memory address (e.g., a pointer), the amount of data, priority, and destination address to the configuration register. Following this, the transmitter sends the address to the HIBI. The transmitter then reads the data from memory and instantly transfers the data.

Receiving is more complicated. Data sent through HIBI may get fragmented. To circumvent this, we have implemented multiple receiving channels that wait for a given amount of data before interrupting the CPU. Each channel can be configured to receive data from any source and save it to memory locations defined by the CPU. There can be several data transfers going on simultaneously, so N2H2 uses the HIBI address to distinguish between them. For example, if two sources are sending data simultaneously, two channels are used. When the expected number of data has arrived on a channel, the CPU is notified by an interrupt or via a poll register.

Figure 4 depicts a data transfer over HIBI. CPU1 sends four words to CPU2. On cycle 0, CPU1 gives a start command to the N2H2 DMA. IRQ is acknowledged in clock cycle 1 and the transfer is started immediately. The address is sent first and then the data. Clock cycles 3–8 are consumed by the HIBI wrapper and arbitration latency. During this delay, another transmission can be proceeding in HIBI, so the latency is hidden. When the access to the HIBI is gained, the address and the data are sent forward in clock cycles 9–13. The data propagates through the receiving unit and buffers until at clock cycle 15 N2H2 sees the address and determines the right channel. Clock cycles 16–19 are used to store the data in the memory of the CPU2. After all the data expected has been received, an IRQ is given to the CPU2 at clock cycle 21.

## 4. MPEG-4 SOFTWARE IMPLEMENTATION

One of the key advantages of data parallel encoding methods is that they enable scalability by using macroblock row, macroblock, or block-level image subdivision. Moreover, spatial data parallelization can be performed with vertical, horizontal, rectangular, or arbitrary shaped slices. The problem of

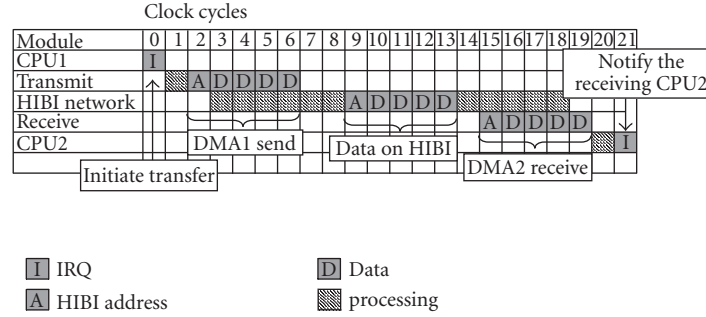


FIGURE 4: An arbitrary four-word data transfer over HIBI between two CPUs.

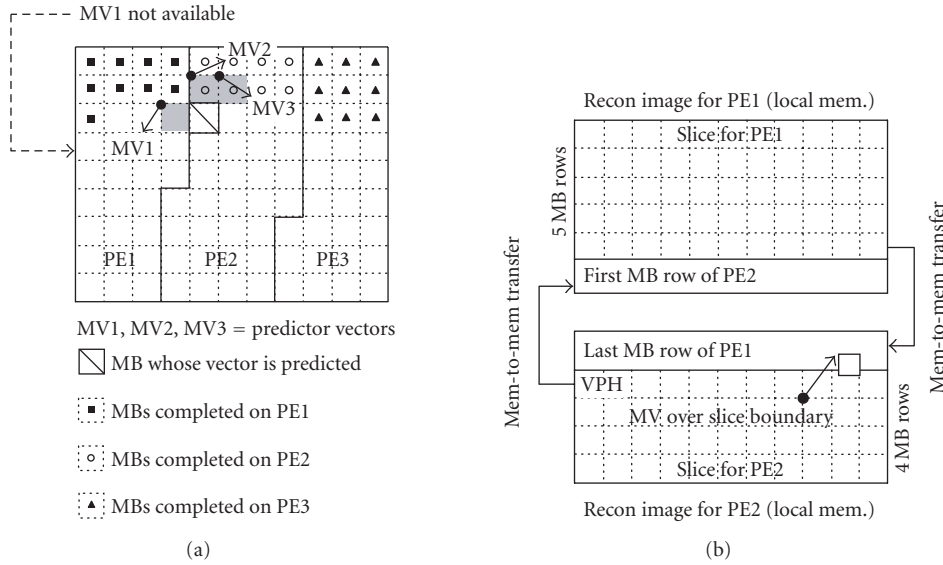


FIGURE 5: (a) Motion vector dependency problem in vertical parallelization and (b) horizontal parallelization for distributed memory machines.

vertical parallelization, shown on the left side of Figure 5, is that predictive coding is not considered, leading to motion vector (MV) and DQUANT (denoting changes in quantization parameter (QP)) dependency problems [27]. For example, H.263/MPEG-4 vector prediction is performed by computing the median of three neighboring vectors referred to as MV1, MV2, and MV3 in Figure 5. Due to data-dependent computations and the different shape of slices, computations do not proceed in a synchronized manner in different slices. For this reason, a data dependency problem arises in the slice boundaries where one of the predictor vectors may not be available.

Horizontal spatial partitioning, however, is natural to raster scan macroblock (MB) coding. The right side of Figure 5 depicts our previous implementation on a distributed memory DSP using MB row granularity [27]. The reconstructed images are made slightly overlap to allow motion vectors to point over slice boundaries. The overlapping areas are also exchanged between processors after local image reconstruction. Prediction dependencies are eliminated by in-

serting slice headers such as H.263 group-of-block (GOB) or MPEG-4 video packet headers (VPH) in the beginning of a slice. Clearly, this results in some overhead in the bit stream but prediction dependencies are avoided. In addition, inter-processor communication and extra memory is needed to implement the overlapping.

However, a drawback of [27] is a somewhat coarse granularity leading to unbalanced computational loads due to the unequal size of slices. For this reason, the original approach is improved by subdividing images using macroblock granularity as in Figure 6. Interprocessor communication and overlapping are further avoided by exploiting a shared memory in an MPSOC type of platform. The new method is highly scalable since the whole image is assignable to a single processor while the largest configuration dedicates a processor for each MB. No interprocessor communication is needed since data can be read directly from the global memory buffer. The shared memories, however, are potential bottlenecks, and thus efficient techniques for hiding transfer latencies are needed.

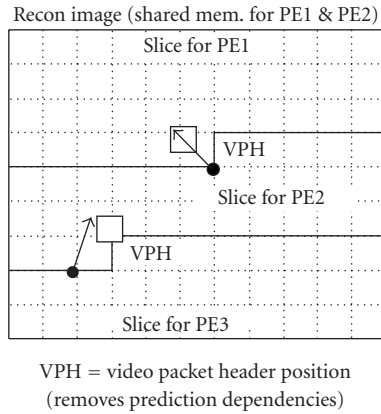


FIGURE 6: Horizontal data parallelization for shared memory.

The data parallelization in Figure 6 was implemented with a master control program and slave MPEG-4 encoders. In addition, a host PC program has been implemented as a user interface. It should be noted that the master and the slave refer to the encoding processors, not, for example, to the Avalon bus master or slave used for Avalon communication. The flow graphs and the synchronization of SW are depicted in Figure 7 while the implementation and the integration details of the master and the slave processor are discussed in Section 5. The master processor controls and synchronizes the encoding. The tasks of the host PC, the master, and the slaves are presented in the following.

#### 4.1. Software implementation

The host PC implements a user interface for inputting encoding parameters to the master. The user interface enables the selection of a video format (resolution and frame rate), bit rate control mode (constant or variable), quantization parameter (QP), as well as the number of slaves used in the encoding. The host PC and the master can communicate via a custom UDP/IP-based messaging protocol, which supports its own flow control, retransmissions, packet structures, fragmentation, and assembly. Our messaging protocol allows real-time modification of the frame rate, QP and bit rate parameters during the encoding.

The tasks of the host PC also include capturing and loading a raw video image, sending the raw data to the master and decoding the output. Received bits are stored to the local disk for debugging. In addition, the host PC measures statistics such as the average encoding frame rate and bit rate. At any time, the host PC can issue a reinitialization command to stop the encoding, release dynamically allocated SW resources, for example, memory, and return to the initial state. For example, this feature enables changes in the video resolution and the number of slaves without rebooting the platform. Also, prototyping and testability are improved since several video formats can be successively tested by changing the parameters.

The tasks of the master are illustrated in the middle of Figure 7. To encode a frame, the master first waits for the parameters from the host PC. Next, the PC sends the raw image (one frame at a time). The master slices the received image, configures the slaves, and signals them to start the encoding. As the slaves complete, each informs the master that it has finished. After all the slaves have completed encoding, the master finds out the sizes of the bit streams of the slaves, merges the bit streams, and sends the merged bit stream (encoded image) to the PC.

Slave tasks are illustrated on the right of Figure 7. First, the slave waits for the parameters from the master. Then, the slave downloads a local motion estimation (ME) window and pixels of the corresponding image macroblock. Then, it encodes the macroblock. This continues as long as there are macroblocks in the slice left to encode. If the local bit buffer goes full, the bits are uploaded to the external image memory. After all the macroblocks have been encoded, the slave uploads the bit buffer to the external memory and begins to wait for the next slice.

The video encoder can run in two different modes: first, it can run in real-time, so one frame at a time is transferred forth and back. Second, it can run in buffered mode, where the PC downloads a video sequence to the master. It is encoded as a whole and sent back to the PC. The video sequence length is parameterizable. Buffered mode mimics a situation where, for example, a video camera is attached to the system and feeding the encoder.

## 5. INTEGRATION

We have now presented a highly scalable hardware platform and a data parallel video encoder. Their integration is presented in the following. The main properties of the Stratix FPGA chip [28] used for this project are given in Table 2. A logic element (LE) contains a four input look-up table and a flip-flop. A digital signal processing (DSP) block contains multiply-and-accumulate (MAC) blocks. These blocks can also be used as fast embedded multipliers, which are utilized by the processors. Phase-locked loops (PLL) are used to generate different clock frequencies. Embedded memory provides on-chip storage.

Apart from the Stratix 1S40 FPGA, the development board offers two UARTs and an Ethernet connection. It has 8 MB of external Flash memory, 1 MB of external SRAM memory, and 16 MB of external SDRAM memory. Downloading and debugging is done via a JTAG connection.

### 5.1. Initial constraints for the architecture

The application requires 64 KB local data memories. As this memory is used for stack and local variables, it needs to be fast on-chip memory. Also, some memory is used for instruction caches. Thus, the limited amount of on-chip RAM in the FPGA bounds the maximum number of processors to four. Optionally, external memories could be used, but the development board does not contain any free, suitable

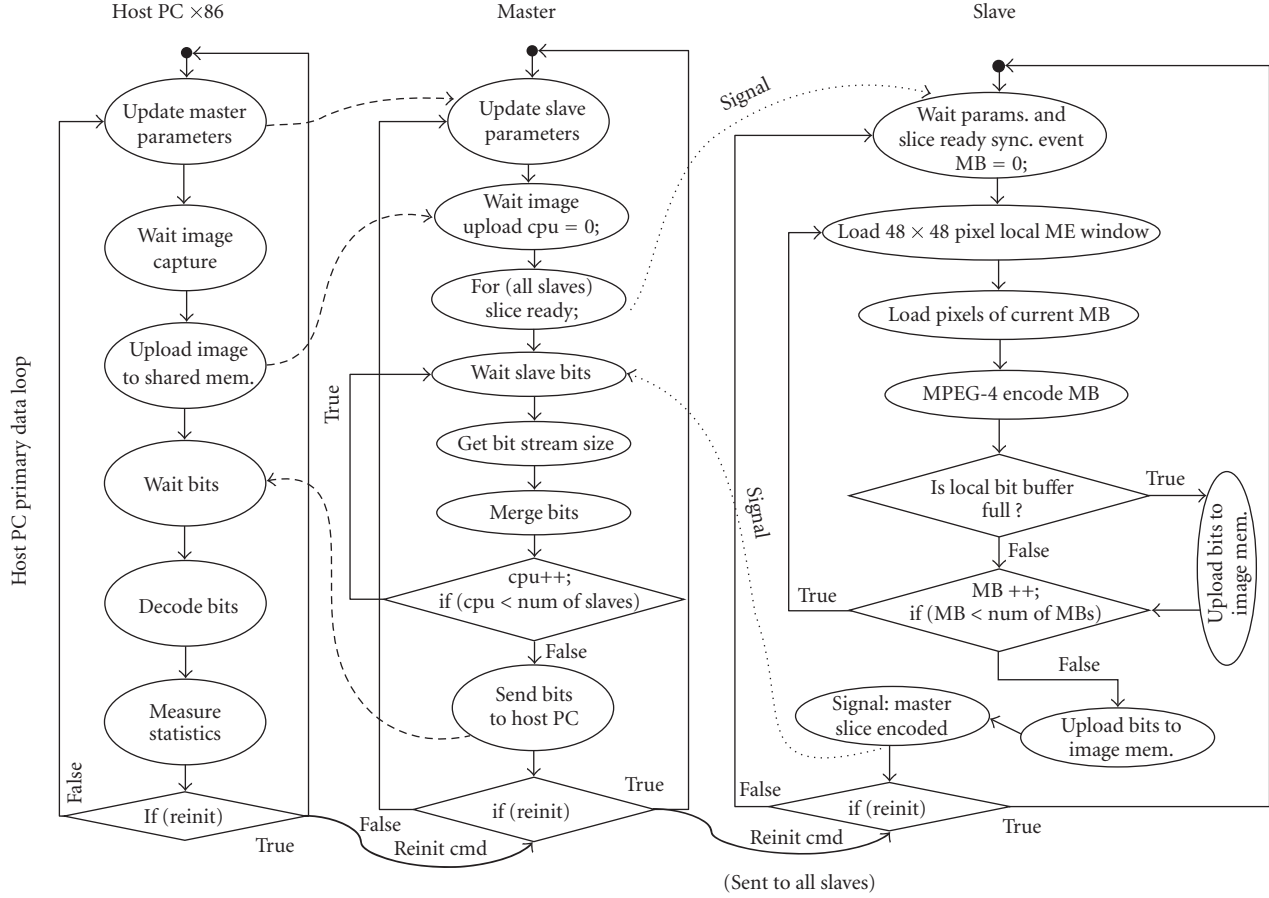


FIGURE 7: Software flow graphs and task synchronization.

TABLE 2: Stratix 1S40 FPGA properties.

Feature	Stratix 1S40 contains
Logic elements (LEs)	41 250
Embedded memory, RAM (bits)	3 423 744
DSP blocks	14
PLLs	12

memories as the external SRAM and external SDRAM are already utilized. Therefore, at maximum one master processor and three slaves are used. The amount of the LEs in the FPGA is more than sufficient to allow for scalability.

Three different memories are used for the video encoding: the on-chip embedded memory, the external SRAM, and the external SDRAM. The flash memory is only used to configure the FPGA upon power-up. The same encoding software can be used for all slaves, which provides an opportunity to use the same instruction memory for all of them. As the application fits to 512 KB, the 1 MB SRAM memory was divided evenly between the master and the slave processors. To reduce the shared memory contention, instruction memory caches were used. Each cache utilizes 8 KB of on-chip memory.

The video encoder was configured in such a way that a 64 KB of local data memory is sufficient for each processor. Small buffers and memories like 2 KB boot program ROMs were also assigned to the on-chip memory. The external 16 MB SDRAM was allocated as the frame memory. A custom SDRAM controller was implemented with special DMA functions. General block transfer commands are implemented with an option to support application-specific commands. For example, we have currently implemented a command for an automatic square block retrieval, for instance an  $8 \times 8$  block, for the video encoder application. In this way we need to commit only a single transfer instead of eight separate transfers. However, the SDRAM controller is fully reusable and can also be used without the application-specific features. The control interface also supports sequential block writes and reads, increasing the efficiency comparing to single write/read operations. The SDRAM DMA is configured with the highest priority messages. However, in practice, using higher priority does not have a notable effect on performance in this application.

## 5.2. Configurations of the components

The exact configurations of the processors are shown in Figures 2 and 3. The bus to the external SRAM goes through the

master processor in practice, as opposed to the architecture shown in Figure 1. Slaves have no access to the address space of the master. Master, however, can access the slave portion of the memory. That gives an opportunity to reconfigure the slaves at run-time, changing the program on the fly. This feature is not utilized in the presented encoder. The data RAM in all CPUs is dual-port, the alternate port is used by the N2H2. Slave UARTs are multiplexed so that the PC can monitor any of the slaves. Timers are used for benchmarking and profiling the program. The master needs an extra timer to be used with the Ethernet.

HIBI is configured as a 32-bit wide single bus and used with the single-FIFO interface. HIBI uses priority-based arbitration, in which the master processor has the highest priority. Each of the HIBI wrappers has a five words deep low priority data FIFO and a three words deep high priority data FIFO. The HIBI bus single burst transfer length is limited to 25 words. Each N2H2 has eight Rx channels, supports 256 separate addresses for channels and has a maximum packet size of 65536 32-bit words. HIBI bridges are not used, because there is no need for multiple clock domains. The MPEG-4 encoding SW exploits the MPSOC features as explained in the following sections.

### 5.3. Implementation of master's tasks

The tasks of the master are discussed in Section 4 and illustrated in the middle of Figure 7. All parameterization is performed via the external shared SDRAM. However, since N2H2 DMA is used to access the shared data memory, SDRAM, one cannot refer to SDRAM via pointers. For example, the C language statement `sdramVariable = pSdramAddr[0]` is not possible. Instead, data between external and local memories is moved with help of dedicated software library calls, for example, `sdramRead()` and `sdramWrite()`.

The current 64 KB limitation of the local data memory, however, presents a more demanding challenge considering that a raw QCIF image takes 37.1 KB. Our solution is to allocate large memory buffers, such as the currently encoded image, the reconstructed images, and the output bit buffers, on the external SDRAM. Two additional 1KB buffers are allocated on the on-chip RAM, which are used for processing data from the large buffers a small segment at a time.

For example, bit stream merging is implemented as a three-step process, illustrated in Figure 8, which is repeated in a loop. As long as there are slave bits remaining, the master first reads a small portion of the slave's bit stream into the input buffer A. Second, the master concatenates and shifts the data after the tail of the master's global bit buffer in buffer B. Third, the master writes the result to the SDRAM and resynchronizes the buffer B with the updated tail of the global bit buffer. The tail contains the bits that did not form a full 32-bit word. The tail is stored to the SDRAM to keep the buffers synchronized, but the master uses the local copy to avoid unnecessary memory traffic. This allows merging of large, arbitrary sized bit streams, and realizes a general merging procedure for variable length coded (VLC) streams that are not aligned to byte boundaries. A similar buffering scheme is also

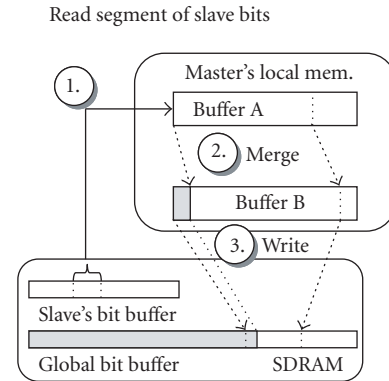


FIGURE 8: Master's bit stream merge task.

used in the "send bits" phase of the master, except no bit shifting is needed since the bit stream is ensured to be byte aligned by the implementation.

### 5.4. Implementation of slaves' tasks

Our MPEG-4 encoder software is identical to [29] except that all DSP specific optimizations have been omitted. A platform-independent portable ANSI-C implementation has been used for Nios II. A single program multiple data (SPMD)-approach has been used, so the slave programs are identical. The program execution is, however, greatly dependant on the data, so the execution flows differ from one CPU to another. Slave tasks are discussed in Section 4 and illustrated in the right side of Figure 7.

The slaves have to operate under low memory conditions. To circumvent this issue, the ME process is carried out in a  $48 \times 48$  sliding window under the programmers control as illustrated in Figure 9(a). The ME window moves in a raster scan pattern centered on the current MB position. An exception is an image boundary, where the window is clipped inside the image. The ME window loading is optimized by packing four consecutive pixels into a 32-bit word. Furthermore, the overlapping of subsequent ME window positions could be used to minimize accesses to the external memory.

Figure 9(b) shows two approaches for updating the ME window. First, one can load a whole window from SDRAM every time the MB position changes, for example, with DMA. In the second approach, only the rightmost column is loaded from SDRAM while the remaining pixels are copied inside the on-chip memory. The data transmissions are executed beforehand in the background in order to minimize the time consumed by waiting for data to be processed.

The drawback of the first approach is high SDRAM bandwidth requirement. For example, the ME of a 4CIF video ( $704 \times 576$ ) at 30 frames/s demands 104 MB/s. In comparison, the second approach requires 34 MB/s but the drawback is that 136 million operations per second (MOPS) are consumed by load/store operations needed for copying. Considering that current SDRAM is fast, for example, 133 MHz



TABLE 3: FPGA utilization statistics.

HW module	Module count	Total mem [KB]	% of total mem	Module area [LE]	Total area [LE]	% of LE
Master Nios I	1	80.8	19.3	2 720	2 720	6.6
Slave Nios II	3	225.1	53.9	2 324	6 972	16.9
N2H2 DMA	4	0	0	1 894	7 576	18.4
HIBI network	1	0.4	0.1	8 506	8 506	20.6
SDRAM DMA	1	0.3	0.1	3 205	3 205	7.8
Utilization		306.6	73.3		28 979	70.2

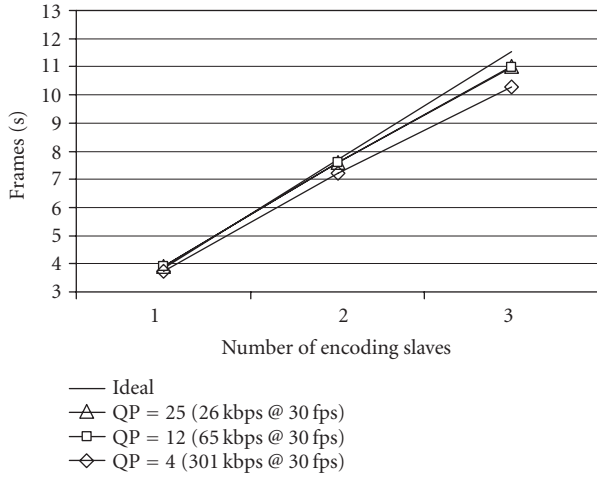


FIGURE 10: Frame rates for sequence carphone.qcif (176 × 144).

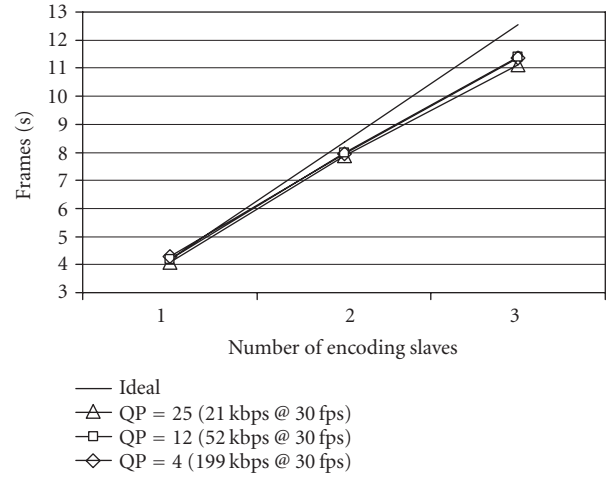


FIGURE 11: Frame rates for sequence news.qcif (176 × 144).

are four read and four write ports, ensuring that no CPU has to unnecessarily wait. N2H2 has eight channels to provide flexibility for the software programmer. There are spare LEs on the FPGA, since only 70% have been utilized.

## 6.2. Encoding speed and software scalability

Figures 10 and 11 present average encoding frame rates as a function of QP and the number of slaves for the *carphone* and *news* QCIF sequences. The bit rates are reported relative to the fixed 30 frames/s sequence output rate. The straight lines depict an ideal speed-up, which is obtained by multiplying the frame rate of a single slave with the number of slaves. The frame rates are measured from the master's main encoding loop.

As scalability was one of our main objectives, the results indicate very good success. The parallelization efficiency of *carphone* using two slaves is within 97% of the ideal result. If we further increase the number of slaves to three, the parallelization efficiency is 93%. As the current FPGA restricts the number of processors to four (one master and three slaves), we estimate the performance of larger configurations in the following.

### 6.2.1. Performance estimation for larger configurations

The complexity of image encoding task depends on slice encoding times as well as the overhead of the master. This

information yields

$$C_{\text{img}}(x, y, n, C_{\text{mb}}) = C_{\text{slice}}(x, y, n, C_{\text{mb}}) + C_{\text{master}}(n), \quad (4)$$

where  $C_{\text{img}}$  is the clock cycles required to encode a frame,  $x$  and  $y$  are the width and height of the luma image in pixels,  $n$  is the number of encoding processors, and  $C_{\text{mb}}$  is the average clock cycles required to encode a macroblock. The term  $C_{\text{master}}$  denotes the master's overhead resulting from the sequentially computed parts.  $C_{\text{slice}}$  represents parallelized computations and is the number of clock cycles required to encode the largest slice in the system. Mathematically  $C_{\text{slice}}$  is computed as

$$C_{\text{slice}} = \left\lceil \frac{\lceil x/16 \rceil * \lceil y/16 \rceil}{n} \right\rceil * C_{\text{mb}}, \quad (5)$$

where the rounding for  $x$  and  $y$  takes care that the image is always divisible to macroblocks, for example,  $x$  and  $y$  do not need to be divisible by 16. The overall rounding finds the size of the largest slice in case the number of macroblocks is not evenly divisible by the number of processors.

The master's overhead results from four subfunctions which can be combined as

$$C_{\text{master}}(n) = C_{\text{config}}(n) + C_{\text{getBitStreamSize}}(n) + C_{\text{merge}}(n) + C_{\text{oth}}(n), \quad (6)$$

where  $C_{\text{config}}$  is due to the configuration of encoding parameters for the slaves,  $C_{\text{getBitStreamSize}}$  results from reading the

TABLE 4: Measured clock cycles for Master's subfunctions.

$f$ (subfunction)	1 slave CPU	2 slave CPUs	3 slave CPUs
Merge	32876.4150	38009.8750	43160.7267
Config	2821.6367	5313.8333	7878.8450
GetBitStreamSize	1264.2967	2517.6617	3772.6450
Oth	117016.7367	117465.5550	117982.0483

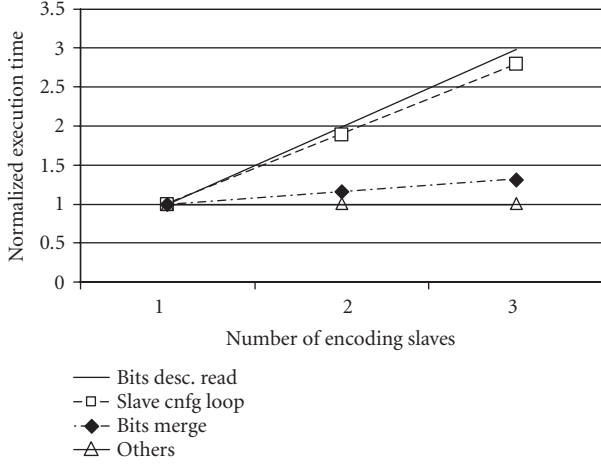


FIGURE 12: Growth rate of complexity of master's subtasks.

sizes of slave bit streams from SDRAM,  $C_{\text{merge}}$  is the number of clock cycles due to merging of slave bits streams, and others are related to the IRQs of the master and an internal state management. It is pointed out that for an optimized system, all Ethernet related tasks are omitted. The measured average clock cycles for the aforementioned subfunctions are presented in Table 4 as a function of the number of encoding processors. In Table 4,  $f$  identifies the subfunction.

For the mathematical model, it is necessary to model the growth of the master task complexity as a function of  $n$ . The complexity change is illustrated in Figure 12, which is plotted using the values in Table 4. For each subfunction, a curve was obtained by plotting the clock cycles with  $n$  encoding processors divided by the clock cycles required for one encoding CPU.

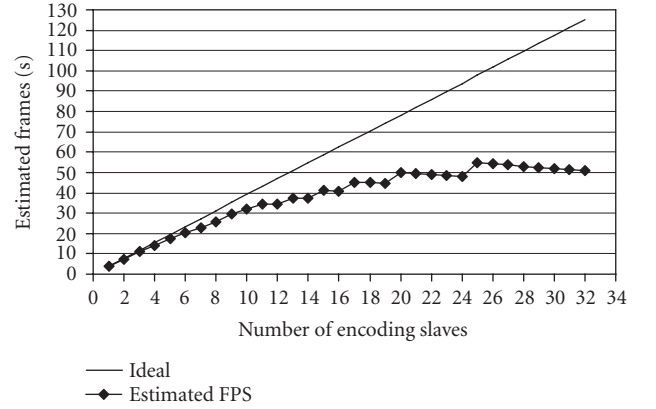
The results in Figure 12 show a linear increase in complexity for all subfunctions of the master. Therefore, the complexities of each subfunction as a function of  $n$  are approximated with

$$C_{f \in \{\text{merge}, \text{config}, \text{getBitStreamSize}, \text{oth}\}}(n) = (a(f) * n + b(f)) * c(f), \quad (7)$$

where  $a$  is the *slope* of the line (the gradient) and  $b$  is the *intercept* on the vertical axis in Figure 12, and  $c$  is the number of clock cycles of a subfunction with one encoding processor. In practice, the clock cycles of one encoding processor are scaled with the linear model to obtain a prediction for an  $n$  CPU system. The subfunction specific parameters for (7) are presented in Table 5.

TABLE 5: Parameterization of linear equations for complexity modeling.

$f$ (subfunction)	$a(f)$	$b(f)$	$c(f)$ (CPU cycles)
Merge	0.1564	0.8436	32876.4150
Config	0.8961	0.1039	2821.6367
GetBitStreamSize	0.9920	0.0080	1264.2967
Oth	0.0041	0.9959	117016.7367

FIGURE 13: Estimated frame rate for  $n$ -processor MPSOC system.

Due to the simultaneous access to the shared data memory at the beginning of each frame encoding, the slave's start-up latency, that is, the time to get enough data to start processing, also increases as the number of slaves increase. This time is not included in the estimate. Each slave requires one motion estimation window, 2560 bytes (640 words), to start processing. It can be assumed that this amount can be transferred from SDRAM to CPU in around 1000 cycles. Thus, since the frame encoding time is millions of clock cycles, the impact is quite insignificant.

Finally, the encoding frame rate estimation on the MPEG-4 MPSOC system is computed with

$$\text{FPS}_{\text{MPSOC}}(x, y, n, C_{\text{mb}}, f_{\text{cpu}}) = \frac{f_{\text{cpu}}}{C_{\text{imge}}(x, y, n, C_{\text{mb}})}, \quad (8)$$

where  $f_{\text{cpu}}$  is the clock frequency of 50 MHz. With benchmarking it was found that  $C_{\text{mb}}$  is on the average of 133394.8 clock cycles per macroblock for *carphone* if a QP value of 12 is used.

Figure 13 presents the predicted encoding frame rate for the optimized MPSOC as a function of  $n$  for the QCIF video format. The values are obtained with (8) using the parameters in Table 5. The system scales nearly linearly when  $n$  is smaller than 12. After 12 encoding processors, the complexity of the master's sequential overhead starts to increase faster than is the benefit of data parallelization and the frame rate saturates after 24 slaves. The small variation at large  $n$  is due to the unbalanced sizes of slices.

One solution to smooth out the variations would be to use a finer subdivision granularly, for example,  $8 \times 8$  or  $4 \times 4$  blocks, but this is impractical from an implementation

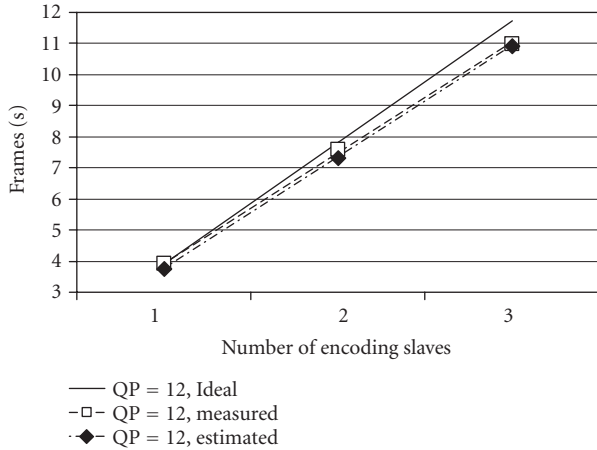


FIGURE 14: Ideal, measured, and estimated frame rates for *carphone.qcif* ( $176 \times 144$ ).

point of view since a macroblock would span two processors. In practice, advanced horizontal parallelization scales better than the row-wise approach used in [27] due to finer granularity.

Figure 14 shows that the model applies well to real, measured performance. It is slightly lower than the measured frame rates with the maximum error being about 4%. The model is applicable to QCIF video streams and also for larger formats by changing the measured execution times appropriately. It takes into account the number of macroblocks and as the number of processors increases, the sizes of slices may not be equal in terms of number of macroblocks resulting in computational unbalance. However, if we use a larger video size, for example CIF, the number of macroblocks also increases. Therefore, with larger video sizes, we can benefit from having more processors than is currently practical for QCIF.

### 6.3. Relative complexity of encoding tasks

The complexities of different CPU tasks show how computing power is shared within one processor. For the master processor, up to 96% of the time is spent waiting the slaves to complete. In buffered mode, frame transmissions over the Ethernet are not executed until the whole video sequence is encoded and thus, this time is not included in the utilization figures. In this case, the master operates as a state machine. However, the master processor is designed to handle all the external tasks that are not directly related to the video encoding. This can include I/O handling (e.g., Ethernet protocol stack), audio coding, and user interfaces, like changing quantization parameters at run-time. The greatest requirement is fast response time for the user interface processor. Double buffering could also be used.

Figure 15 illustrates how the execution time is divided for one slave processor to encode one macroblock. *Motion estimation* (ME) is by far the most computationally challenging task. Other time consuming tasks are *interpolation*, *quantization* (Q), *inverse quantization* (IQ), *DCT*, and *IDCT*. The

time for *MasterWait + Poll* is consumed by waiting for the master to collect and merge the bit stream, deliver a new raw slice, and provide the MPEG-4 coding parameters for next frame.

### 6.4. Hardware scalability

HIBI offers good scalability. For this architecture, adding or removing processors is simple—it only takes minutes to parameterize and prepare for synthesis. HIBI also offers a convenient way to add new IP components, for example new processors or hardware accelerators. It is possible to add new features to the system without altering the video encoding. A simple example is the addition of a hardware utilization monitor. The addition does not require any changes to the encoding and it is easy to plug into the system by just adding one HIBI wrapper to the interconnection. Encouraging preliminary results have been obtained from attaching a hardware accelerator (DCT, IDCT, quantizer, and inverse quantizer) to the system (around a couple of FPS increase).

The utilization of the HIBI bus is measured to be only 3%. Therefore, the interconnection is not likely to become a bottleneck even with larger configurations. From a performance point of view, the interconnection architecture should be as invisible as possible in hiding data transmission time from the CPU.

The speed-up gained by adding processors (e.g., Figure 10) shows that the interconnection architecture performs well. Figure 15 shows that only 4% (SDRAM config + comm wait) of the CPU encoding time is spent on data transmissions and waiting for data.

As the HIBI utilization is low, it is expected that the shared data memory (SDRAM) will not become the bottleneck in the future. We have determined that the interconnection is capable of transmitting the required data for a CIF image, which is four times larger at 25 frames/s using a clock frequency of 28 MHz without forming a significant performance bottleneck. The application can perform data fetches from memory in parallel with computation. Therefore, memory latencies can be tolerated.

Shared instruction memory utilization, via the Avalon bus, is at most 20% of available bandwidth. We have investigated the effect of shared instruction memory on performance and preliminary results indicate that it is currently negligible with respect to total frame encoding time. However, four processors could share one instruction memory port. The absolute maximum for sufficient performance [30] is ten processors per port.

### 6.5. Comparison of scalability to related work

In Figure 16, the scalability of different video encoders is presented. The optimal situation would be a gain of 100% parallel efficiency, that is, a speed-up of 2.0 for 2 CPUs and 3.0 for 3 CPUs. Of the four general categories of parallelization, results from three are presented. No publications were found that describe a functional parallel encoder that scales with a varying number of processors.

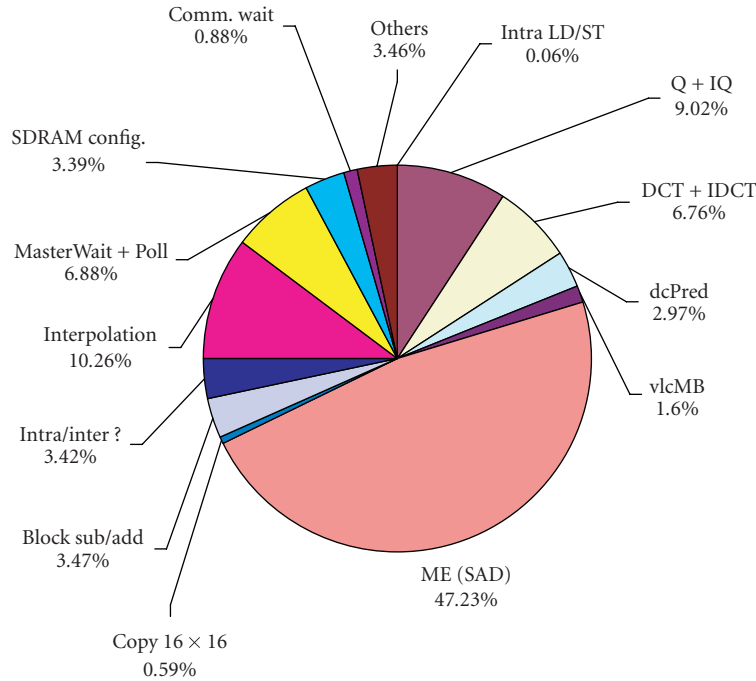


FIGURE 15: Relative complexities of encoding tasks on one slave CPU.

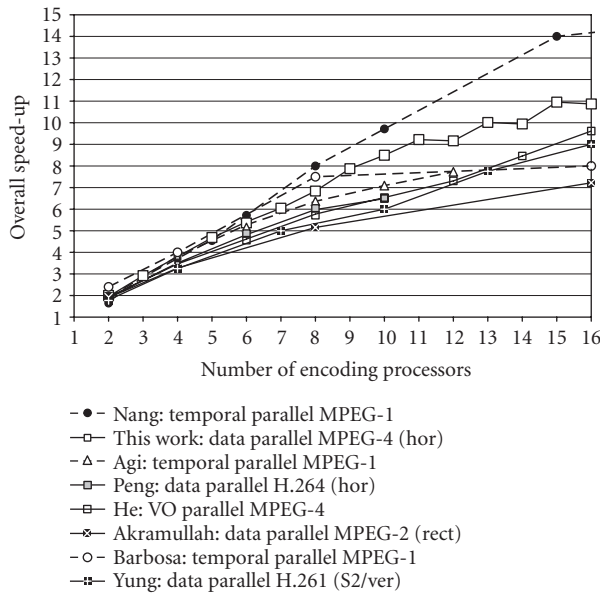


FIGURE 16: Speed-ups of different video encoder implementations.

Nang and Kim [8], Agi and Jagannathan [9], and Barbosa et al. [10] use temporal parallelism. Nang et al. have received great parallelization efficiency. However, as temporal parallelism needs frames to be buffered (increasing the latency) it is not considered to be low-latency real-time video encoding. In our work we present a data parallel application. Similar works are Peng and Zhao [11], Akramullah et al. [12], and

Yung and Leung [13]. One work based on video-object parallelism, which is closely related to data parallelism, is presented in He et al. [14].

The results from related research are frequently presented as speed-up or frame rate curves, so the exact numerical values cannot be obtained. We have reproduced the curves in Figure 16 to be as accurate as possible. Results from Nang and Kim, Agi and Jagannathan, He et al., and Yung and Leung are plotted. Barbosa et al. have given results for three cases. The result for 16 CPUs is estimated from the sketched figure. Our results are from the model described earlier. Results for Peng and Zhao and Akramullah et al. are plotted from the given numerical figures.

Figure 16 shows that the scalability of our implementation is the highest of all data parallel implementations. That implies both good parallelization efficiency achieved with software and an efficient hardware architecture.

## 7. CONCLUSIONS

A highly scalable MPSOC architecture with an MPEG-4 encoder has been presented. The parallelization efficiency of the application, when the number of encoding processors is increased from one to two, is 97% and to three, 93%. No real-time video encoder was found that has such a high scalability for real-time video encoding. Our benefit is due to both a well-designed architecture and application. The architecture efficiently hides the data transmissions from the processors. The software takes full advantage of the parallelism by elegantly sharing the encoding load. The software does not need

any changes when the number of processors is altered, thus the scaling effort is very low.

The scalability and flexibility of the MPSOC architecture is gained by using an HIBI on-chip network and soft-core processors in a plug-and-play fashion. The performance and area usage can be flexibly compromised by changing the number of processors. It takes only minutes to change the number of processors and then the new system is ready to be synthesized. Since the architecture is implemented in an FPGA, the amount of on-chip memory becomes a limiting factor.

In the future, platform flexibility will be demonstrated with the use of different soft-core processors and hardware accelerators. The connection to external instruction memory will be replaced with an HIBI interface to achieve better clock frequencies. Furthermore, scalability will be further evaluated with larger FPGA chips and by connecting several ones together to form a larger system.

## REFERENCES

- [1] E. Salminen, A. Kulmala, and T. D. Hämmäläinen, "HIBI-based multiprocessor SoC on FPGA," in *IEEE International Symposium on Circuits and Systems (ISCAS '05)*, pp. 3351–3354, Kobe, Japan, May 2005.
- [2] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hämmäläinen, and T. D. Hämmäläinen, "A parallel MPEG-4 encoder for FPGA based multiprocessor SoC," in *15th International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 380–385, Tampere, Finland, August 2005.
- [3] Altera Corporation, *Nios Development Board: Reference Manual, Stratix Professional Edition*, Ver. 1.1, July 2003.
- [4] E. Salminen, T. Kangas, J. Riihimäki, V. Lahtinen, K. Kusuilina, and T. D. Hämmäläinen, "HIBI v.2 communication network for system-on-chip," in *Computer Systems: Architectures, Modeling, and Simulation*, A. D. Pimentel and S. Vassiliadis, Eds., vol. 3133 of *Lecture Notes in Computer Science*, pp. 412–422, Springer, Berlin, Germany, July 2004.
- [5] P. Kuhn, *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*, Kluwer Academic, Dordrecht, The Netherlands, 1999.
- [6] I. Ahmad, Y. He, and M. L. Liou, "Video compression with parallel processing," *Parallel Computing*, vol. 28, no. 7-8, pp. 1039–1078, 2002.
- [7] O. Cantineau and J.-D. Legat, "Efficient parallelisation of an MPEG-2 codec on a TMS320C80 video processor," in *IEEE International Conference on Image Processing (ICIP '98)*, vol. 3, pp. 977–980, Chicago, Ill, USA, October 1998.
- [8] J. Nang and J. Kim, "An Effective parallelizing scheme of MPEG-1 video encoding on ethernet-connected workstations," in *Proceedings of the Conference on Advances in Parallel and Distributed Computing (APDC '97)*, pp. 4–11, Shanghai, China, March 1997.
- [9] I. Agi and R. Jagannathan, "A portable fault-tolerant parallel software MPEG-1 encoder," *Multimedia Tools and Applications*, vol. 2, no. 3, pp. 183–197, 1996.
- [10] D. M. Barbosa, J. P. Kitajima, and W. Weira Jr., "Parallelizing MPEG video encoding using multiprocessors," in *Proceedings of the XII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAP '99)*, pp. 215–222, Sao Paulo, Brazil, October 1999.
- [11] Q. Peng and Y. Zhao, "Study on parallel approach in H.26L video encoder," in *4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '03)*, pp. 834–837, Chengdu, China, August 2003.
- [12] S. M. Akramullah, I. Ahmad, and M. L. Liou, "Performance of software-based MPEG-2 video encoder on parallel and distributed systems," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 4, pp. 687–695, 1997, Transaction Briefs.
- [13] N. H. C. Yung and K.-K. Leung, "Spatial and temporal data parallelization of the H.261 video coding algorithm," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 1, pp. 91–104, 2001.
- [14] Y. He, I. Ahmad, and M. L. Liou, "MPEG-4 based interactive video using parallel processing," in *International Conference on Parallel Processing (ICPP '98)*, pp. 329–336, Minneapolis, Minn, USA, August 1998.
- [15] M. J. Garrido, C. Sanz, M. Jiménez, and J. M. Menasses, "An FPGA implementation of a flexible architecture for H.263 video coding," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 4, pp. 1056–1066, 2002.
- [16] R. Kordasiewicz and S. Shirani, "Hardware implementation of the optimized transform and quantization blocks of H.264," in *Canadian Conference on Electrical and Computer Engineering (CCECE '04)*, vol. 2, pp. 943–946, Niagara Falls, Ontario, Canada, May 2004.
- [17] Q. Peng and J. Jing, "H.264 codec system-on-chip design and verification," in *5th International Conference on ASIC (ASICON '03)*, vol. 2, pp. 922–925, Beijing, China, October 2003.
- [18] Y.-L. Lin, C.-P. Young, Y.-J. Chang, Y.-H. Chung, and A. W. Y. Su, "Versatile PC/FPGA based verification/fast prototyping platform with multimedia applications," in *Proceedings of the 21st IEEE Instrumentation and Measurement Technology Conference (IMTC '04)*, vol. 2, pp. 1490–1495, Como, Italy, May 2004.
- [19] M. Martina, A. Molino, and F. Vacca, "FPGA system-on-chip soft IP design: a reconfigurable DSP," in *Proceedings of the 45th Midwest Symposium on Circuits and Systems (MWSCAS '02)*, vol. 3, pp. 196–199, Tulsa, Okla, USA, August 2002.
- [20] X. Wang and S. G. Ziavras, "Parallel direct solution of linear equations of FPGA-based machines," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS '03)*, pp. 113–120, Nice, France, April 2003.
- [21] Altera Corporation, "Avalon Interface Specification," Ver. 2.4, January 2004.
- [22] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, and U. Rückert, "Dynamically reconfigurable system-on-programmable-chip," in *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP '02)*, pp. 235–242, Canary Islands, Spain, January 2002.
- [23] OCP-IP Alliance, "Open Core Protocol specification," Release 2.0, 2003.
- [24] Altera Corporation, "Nios 3.0 CPU datasheet," October 2004.
- [25] Altera Corporation, "Nios II Processor Reference Handbook," May 2005.
- [26] Altera Corporation, Nios II, Site visited 28.11.2005, <http://www.altera.com/products/ip/processors/nios2/ni2-index.html>.
- [27] O. Lehtoranta, T. D. Hämmäläinen, V. Lappalainen, and J. Mustonen, "Parallel implementation of video encoder on quad DSP system," *Microprocessors and Microsystems*, vol. 26, no. 1, pp. 1–15, 2002.
- [28] Altera Corporation, "Stratix Device Handbook," January 2005.

- [29] O. Lehtoranta and T. D. Hämäläinen, “Feasibility study of a real-time operating system for a multichannel MPEG-4 encoder,” in *Multimedia on Mobile Devices*, vol. 5684 of *Proceedings of SPIE*, pp. 292–299, San Jose, Calif, USA, January 2005.
- [30] A. Kulmala, E. Salminen, O. Lehtoranta, T. D. Hämäläinen, and M. Hännikäinen, “Impact of shared instruction memory on performance of FPGA-based MP-SoC video encoder,” in *The 9th IEEE workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS '06)*, pp. 59–64, Prague, Czech Republic, April 2006.

**Ari Kulmala** received the M.S. degree in computer science from the Tampere University of Technology (TUT), Finland, in August 2005. He is currently pursuing Ph.D. degree and working as a research scientist in the DACI Research Group in the Institute of Digital and Computer Systems at TUT. His research interests include system-on-chip architectures, interconnections, and low power design.



**Olli Lehtoranta** was born in Vantaa, Finland, on 2nd of June 1976. He received his M.S. degree “with distinction” in computer science from Tampere University of Technology, in November 2001. In 1999, he joined the Institute of Digital and Computer Systems Laboratory where he currently works towards his Ph.D. degree. He has authored two journals and nine conference papers for refereed scientific publications. His current research interests include parallel processing for video encoding, error resilient video encoding methods, computational complexity analysis of video encoding algorithms, and low-level assembly optimization of video codecs for media instruction set architectures (ISA) as well as DSP.



**Timo D. Hämäläinen** received the M.S. degree in 1993 and the Ph.D. degree in 1997, both from Tampere University of Technology (TUT). He acted as a Senior Research Scientist and Project Manager at TUT during 1997–2001. He was nominated to be Full Professor at TUT, Institute of Digital and Computer Systems, in 2001. He heads the DACI Research Group that focuses on three main lines: wireless local area networking and wireless sensor networks, high-performance DSP/HW-based video encoding, and interconnection networks with design flow tools for heterogeneous SoC platforms.



**Marko Hännikäinen** received the M.S. degree in 1998 and the Ph.D. degree in 2002, both from Tampere University of Technology (TUT). Currently he acts as a Senior Research Scientist in the Institute of Digital and Computer Systems at TUT, and a Project Manager in the DACI Research Group. His research interests include wireless local and personal area networking, wireless sensor and ad hoc networks, and novel web services.

