Architectural Exploration of Heterogeneous Multiprocessor Systems for JPEG*

Seng Lin Shee[1,2,3], Andrea Erdos[1], and Sri Parameswaran[1,2]

Multicore processors have been utilized in embedded systems and general computing applications for some time. However, these multicore chips execute multiple applications concurrently, with each core carrying out a particular task in the system. Such systems can be found in gaming, automotive real-time systems and video / image encoding devices. These system are commonly deployed to overcome deadline misses, which are primarily due to overloading of a single multitasking core. In this paper, we explore the use of multiple cores for a single application, as opposed to multiple applications executing in a parallel fashion. A single application is parallelized using two different methods: one, a master-slave model; and two, a sequential pipeline model. The systems were implemented using Tensilica's Xtensa LX processors with queues as the means of communications between two cores. In a master-slave model, we utilized a course grained approach whereby a main core distributes the workload to the remaining cores and reads the processed data before writing the results back to file. In the pipeline model, a lower

granularity is used. The application is partitioned into multiple sequential blocks; each block representing a stage in a sequential pipeline. For both models we applied a number of differing configurations ranging from a single core to a nine-core system. We found that without any optimization for the seven core system, the sequential pipeline approach has a more efficient area usage, with an *area increase to speedup ratio* of 1.83 compared to the master-slave approach of 4.34. With selective optimization in the pipeline approach, we obtained speed ups of up to 4.6× while with an area increase of only 3.1× (area increase to speedup ratio of just 0.68).

---

1. INTRODUCTION

The demand for fast, low cost, and energy efficient processors for streaming applications are steadily increasing due to the unending appetite for multimedia devices. Embedded systems such as MP3 players, DVD recorders, and mobile phones require more processing power as more applications are assigned to them. For example, a mobile phone today would not only need to process voice data, but also encode video, schedule appointments, and display interactive games on screen. Specialized components (i.e., DSPs and coprocessors) have been used to speedup the computation load. With tighter time to market deadlines, reduced availability of workforce per transistor, increased possibility of design errors, and phenomenal mask costs, Application Specific Instruction-set Processors (ASIPs) have been introduced into the embedded systems domain, to improve design productivity. An ASIP's instruction set and its underlying architecture can be configured to a specific application in order to improve efficiency. ASIPs provide a good trade-off between efficiency and flexibility, enabling the same design to be reused between different product variants, and updated with little additional cost.

A chip containing a single core would not scale sufficiently well to facilitate the increasing workload of an embedded system. Instruction level parallelism is not scalable due to the inherent limitation of its parallelism. Thus, single chip multiprocessor architectures have been researched to exploit a higher level of parallelism (i.e., task parallelism). One core would be able to handle one task while another would execute an independent function. In a mobile phone, audio would be implemented in one processor while video might be implemented in another, and the communication stack in yet another and so on.

This paper tries to explore the possibility of further parallelization at a lower granularity than at the level explored in the previous paragraph,

yet at a higher level than at the instruction level. We want to further explore the possibility of speeding up the application with heterogeneous components (i.e., ASIPs of different configurations) such that the speedup increases at a faster rate than at which the area increases (i.e., they have more performance per gate than in a single processor). To perform this parallelization, we take a single application (in this case a JPEG application) and try to parallelize it with a number of processors. We try two separate methods, one a master-slave model and the other a pipeline model. We start with identical processors (i.e., the most powerful standard processor available within the suite of tools we use), and then either enhance it with additional instructions (if a particular processor is the bottleneck), or if the workload is little, diminish it by the use of a smaller processor with reduced cache.

The paper is organized as follows: Section 2 gives a broad overview of the multiprocessor research thus far and Section 3 specifies the benchmark application and platform which this case study is based on. Section 4 specifies different configurations and methodologies used to partition and organize the multiprocessor configurations, and Section 5 reports the experimental methodology used in this case study. Section 6 analyzes the performance improvements for each system and presents a walk through on selective optimization of a particular configuration. Finally, in Section 7 the conclusions are summarized.

2. RELATED WORK

Multicore architectures are becoming prevalent in SoC designs. For example, the design in Kumar et al.[1] is common in DSP systems where multiple processing entities perform computation on different parts of the system concurrently.

A general purpose multiprocessor system (single ISA) enables programmability and speeds up design-to-market time. Pham et al. proposed a multi-ISA multicore architectures,[2] which requires different processors in the system to execute different instruction sets. Such cores typically address vector/data-level parallelism and instruction-level parallelism simultaneously. However, a single-ISA heterogenenous[3] system provides the advantage of easily mapping any application stage to any of the cores in the multicore system.

Various heterogeneous multiprocessor systems have been implemented, primarily in the automotive real-time systems[4] and video/image encoding domain. Strik et al.[5] explored the use of a heterogeneous system in a real-time video and graphics streams management system, while Zhang and Wu[6] applied an adaptive job assignment scheme to perform

data partitioning for a multiprocessor implementation of MPEG2 video encoding. A heterogeneous multiprocessor (five cores) for HDTV systems was developed in Berić et al.[7].

Gopalakrishnan and Caccamo[8] used heterogeneous systems in a different manner. The work generalizes the approach started by Baruah[9] which replicates recurring tasks on multiple processing units to ensure a degree of fault tolerance. Maintaining replicas of a task in different processors ensures that single processor failures will be tolerated well.

A multicore system would have various communication schemes to provide the neccesary link between each core in the system. Kim et al.[10] developed a new CDMA-based on-chip interconnection network using a Star NoC topology. To enable quick design of a multicore processor system and the evaluation of its interconnect system, Wieferink et al.[11] developed a methodology for retargetable MPSoC integration at the system level based on LISA[12] processor models and the SystemC framework.[13]

Single core applications utilize instruction level parallelism which is enabled by pipeline processors. Multiprocessors are able to exploit task level parallelism by executing different tasks on separate cores simultaneously. Different schemes have been developed, such as the reuse of the pipeline scheme.[14]

Several pipelining methods have been explored. Jeon and Choi[15] partitioned loops into several pipeline stages. The iterative algorithm proposed increased parallelism and reduced the hardware cost of the designed system. Kodaka et al.[16] combined both course grain and fine grain parallelism (which includes loop pipelining) using a single OSCAR chip multiprocessor. The work exploits course grain task, loop parallelism, and instruction level parallelism using the OSCAR compiler. The OSCAR chip is comprised of several processor-elements (PEs) connected to local memory and shared memory, facilitating data transfer among processors.

Banarjee et al.[17] incorporated heterogeneous digital signal processors with macropipelining based scheduling. The technique utilized a signal flow graph (SFG) as a basis for partitioning. The work shows that heterogeneous multicores are able to improve the throughput rate several times that of the conventional homogeneous multiprocessor scheduling algorithms.

Our work differs from all of the above by the informal exploration of the design space of two differing multiprocessor core architectures: one, a master-slave model and another, a pipelined model. The exploration of the pipeline model not only explores differing cores, but also enhances the cores with additional instructions and diminishes them by reducing cache sizes, until the pipeline stages are roughly balanced. In comparison

with the interconnect network model in Refs. 10 and 11, and system bus models,[18] we develop our system using the ports and queues architecture introduced in the Xtensa LX framework. By careful design, we show that our pipeline system improves performance by approximately five times while consuming three times the area.

The contributions include:

(1) A novel multicore architecture is proposed based upon a pipeline structure, which is compared against a master-slave multicore architecture.

(2) Partitioning sequential programs into parallel task, which are executed in separate cores connected in a pipelined structure via port and queue interfaces of the Tensilica Xtensa LX architecture.

(3) For the first time, we study the mapping of different application partitions into disparate architecture configurations, thus investigating on the overall performance and area behaviour of such an approach.

3. BACKGROUND

This case study is based on mapping different parts of the benchmark program and employing a set of industrial tools to rapidly optimize and simulate the entire system in a multiprocessor configuration. The partitioning of the program (initially based on functions in the source program) is performed by analyzing the benchmark results of the simulation. The set of the industrial design tools enable us to quickly explore the extent of improvements and area usage of a heterogeneous multiprocessor system.

3.1. Case Study Application

A freeware JPEG compression algorithm implementation is used in this case study. The simplistic nature of the program benefits this case study as various sections of the code can be distinguished, partitioned, and separated into a multiprocessor configuration. The program partitions are created based on the different stages of processing from a standard JPEG encoding algorithm (such as *DCT*, *quantization*, *zero run-length encoding*, and *Huffman encoding*).[19] This partitioning was done in such a manner to easily upgrade cores, if one of the algorithms changed. For example, if a better DCT algorithm was implemented, only a single stage had to be altered. This level of partitioning allows the architecture to be kept while the software and core of a particular stage is changed to match the changes in the algorithm.
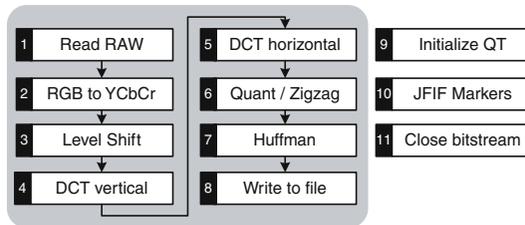
Fig. 1. The main stages in a JPEG encoder.

Figure 1 shows the various partitions or stages of the program which have the possibility to be allocated to different processors. The arrows indicate the flow of RAW bitstreams through the various stages of the encoding process before being written out to file.

The JPEG encoder program initially accepts a configuration file which specifies the name of the RAW file to read, the quality factor, and the format of the RAW image. These tasks are performed in stage 1. The program then proceeds to initialize the quantization tables and write the appropriate JFIF header information to the output file, which includes *Quantization* and *Huffman* tables (stages 9 and 10). The program allocates two main buffers; one for the complete RAW image which is read from file and the other for the resulting JPEG file.

The JPEG program then starts reading RGB values from the buffer and converts them to YCbCr values (stage 2). These values are then value shifted (stage 3) (based on JPEG specifications). A macroblock is then selected one at a time in sequence of Y, Cb, and Cr to be DCT transformed and quantized with the values ordered in a zigzag manner (stages 4–6). The pixel streams are fed into the Huffman encoder (stage 7) which processes these streams serially. The generated code is finally output to a file (stage 8).

3.2. Baseline Processor Description

This case study utilizes Tensilica's Xtensa LX processor.[20] The Xtensa LX is part of the line of Tensilica's microprocessor core family which is configurable, extensible, and supported by automatic hardware and software generation tools. The synthesizable core is configurable to allow designers to precisely tailor each processor implementation to match the target application requirements. The Xtensa core ISA has a 24-bit instruction set base and allows 16-bit instructions for higher code density. All instructions can operate on 32-bit data.

The Xtensa LX, like previous Xtensa processors, is able to support extended instructions, which are written in Tensilica Instruction Extension (TIE) language. Such instructions are able to do the work of multiple instructions of a general-purpose processor. Extended instructions include fusion instructions,[21] SIMD/vector instructions and FLIX[22] instructions. Flexible Length Instruction Xtensions (FLIX) are VLIW-like instructions whereby multiple operations can be performed in a single instruction.

TIE queues and ports have been introduced in Tensilica's Xtensa LX processors. These features are used to communicate to the world outside of the processor and can communicate at a much wider bandwidth than existing interconnects. Queue interfaces are used to *pop* an entry from an input queue for incoming data or *push* data to an outgoing queue. The logic to stall the processor when it wants to read an empty input queue or write to a full output queue is automatically generated by the Xtensa Toolset. Ports are *wires* that the processor uses to directly sample the value of an external signal or drive the value of any TIE state on external signals.

Functions are created to *push* and *pop* from the queues. The functions are blocking functions; as a *push* into a full queue or a *pop* from an empty queue results in a stall of the particular pipeline stage. These functions are TIE instructions and form part of the extended instructions of the Xtensa LX processor architecture (refer to Fig. 2).

The configuration of the base processor used in the case study has been *optimized* to provide satisfactory results when executing the benchmark application under a single processor system and is shown in Table I as *LX1*. Also shown is a highly stripped down version of the Xtensa LX processor *LX2* which will be used to replace under-utilized cores to save area and power (see Section 4.2).

4. METHODOLOGY

We explore the various ways a multiprocessor system can be configured to speed up a simple application. In this section, we outline the multiprocessor architecture as we increase the number of cores in the system. We show two methods to exploit the parallelism within the code structure of the program. Our methodology utilizes the queue interfaces which are available on Tensilica's Xtensa LX[20] processors. A simplified JPEG encoder is modified and partitioned to execute on such a system.

queue FIFO_OUT1 32 out /* this is an output queue from main to slave1 */

queue FIFO_IN2 32 in /* this is an input queue from main to slave1 */

/* writing to slave1 from main straight from memory */

operation WriteFifoMemMain1

{in AR *addr} { out FIFO_OUT1, out VAddr, in MemDataIn16} {

   /* addr is the pointer to the data to be sent */

   assign VAddr = addr;

   /* here the data to be sent is of size 16 bits */

   assign FIFO_OUT1 = MemDataIn16; }

/* reading from main to slave1 straight to memory */

operation ReadFifoMemSlave1

{in AR *addr} { in FIFO_IN2, out VAddr, out MemDataOut16} {

   /* addr is the pointer to where the received data is to be stored */

   assign VAddr = addr;

   assign MemDataOut16 = FIFO_IN2; }

Fig. 2. Sample TIE code implementing a TIE queue interface between two cores.

Table I. Processor Configuration.

Parameter	LX1	LX2
Speed	533 MHz	
Process	90 nm GT	
Pipeline length	5	
Size	63,843 gates	39,789 gates
Core size	0.32 mm^2	0.18 mm^2
Core power	74.35 mW	41.3 mW
Memory Area (mm^2)	1.76 mm^2	0.15 mm^2
Instruction Cache	32 kB	1 kB
Data Cache	32 kB	1 kB
ISA instruction options	MUL32, MUL16, density instructions, boolean registers, zero overhead loops, TIE wide stores, 32 bits sign extend, TIE arbitrary bytes	density instructions, boolean registers, zero overhead loops, TIE wide stores
Max instruction width	8 bytes	3 bytes
PIF interface width	128 bits	32 bits

4.1. Method I

Master-slave models have been used in parallel computing to enable task management and parallel and distributed data structures.[23] In a master-slave model, a master program is given the responsibility to spawn processes, initialize, collect, and display results while the slave programs perform the computations.[24] A master-slave model of a multicore system was implemented with a differing number (from three to seven) of Xtensa LX processor cores, which were instantiated using the Xtensa LX XTMP/ISS environment. In each model, there was only one main core, with $(N-1)$ slave cores, where N is the total number of cores in the system.

Xtensa Modeling Protocal (XTMP) was used to link the main core with the slave cores using TIE queue interfaces, with two interfaces per core, a TIE queue in and a TIE queue out. These TIE queues were implemented using a custom-designed TIE file for each core in the system. TIE instructions were written to retrieve the data from memory on one core and send it via a queue to the receiving core, which then stored the data into memory by means of another TIE instruction. This reduces the overhead of having to store the data in registers before sending it. The *MemDataIn<x>* and *MemDataOut<x>* TIE instructions were used to implement this feature, where x is the size of the data in bits.

In contrast to the other approaches utilizing bus topologies,[18] communication between cores is achieved via TIE queues. Such communication appears to be the fastest approach to transmitting data among physical cores. Values from slave cores which have been computed early can immediately be read/used by the master program (compared to the overhead involved when reading and writing through a shared bus).

Only the main core communicates with each slave core. The slave cores are not able to communicate between themselves. In each model, a send/receive protocol is implemented to check if a core is ready to receive data before sending the data. Special send and receive messages are sent from either the main core to a particular slave core, or vice versa, and the initiator of this exchange then stalls while it waits for a reply. When the non-initiator of this exchange is ready to receive the data (after having received the message from the initiator), it sends an acknowledgement to the initiator which then sends the data.

A JPEG encoder can be separated into different stages that can be run on individual cores. An independent program is compiled for each core, depending on which functions that particular core implements. The simulator loads these individually compiled programs onto each core. When encoding a JPEG image file, (as referred to by the control flow

Table II.   Master/Slave Processor Configuration

Cores	Stages				
	Three cores	Four cores	Five cores	Six cores	Seven cores
Main	1, 7-11	1, 7–11	1, 7–11	1, 7–11	1, 7–11
Slave1	2-6, 9	2–6, 9	2–6, 9	2–6, 9	2–6, 9
Slave2	2–6, 9	2–6, 9	2–6, 9	2–6, 9	2–6, 9
Slave3	–	2–6, 9	2–6, 9	2–6, 9	2–6, 9
Slave4	–	–	2–6, 9	2–6, 9	2–6, 9
Slave5	–	–	–	2–6, 9	2–6, 9
Slave6	–	–	–	–	2–6, 9

diagram in Fig. 1), the processors in each system implement each stage according to Table II. The JPEG encoder allows for parallelization since data is processed at a macroblock (i.e., 8×8 pixels) level. This means that several cores can be processing different macroblocks of data at the same time. Only *Huffman encoding* (stage 7) must be done serially, due to the nature of the Huffman encoding algorithm.

Once the input RAW image file is read (stage 1) by the main core, the quality factor of the RAW image file is sent out to all slave cores, which then begin initializing the quantization tables for the file (stage 9). Within the JPEG application, the *RGB to YCbCr conversion* (stage 2) is done in parallel by roughly dividing the RAW image file into $(N − 1)$ parts, where N is the total number of cores in the system. These different parts are then sent to the slave cores, which then complete the *RGB to YCbCr conversion*. The converted data is then sent back to the main core from the slave cores and written to memory in the order that it was sent out.

The main core then begins initializing the quantization tables (stage 9), and writes the JFIF[25] headers to the output file (stage 10). When the main core has reached the encoding stage, it organizes the division of data to be encoded by sending out the first macroblock of 64 Y's, 64 Cb's, and 64 Cr's to the first slave, the second macroblock to the second slave, and so on. Once each slave has received its macroblock of data, it begins encoding it by performing the *LevelShift, DCT*, and *Quantization/Zigzag* (stages 3–6) functions on the data sequentially, then sends back the encoded data to the main core. Once the main core has sent data to core $(N−1)$ in the sequence, it then receives the encoded data from each slave core in the same order that it was sent out. The main core then performs *Huffman encoding* (stage 7) on the received data. The cycle continues, until all macroblocks are processed. If the number of macroblocks is not a multiple of $(N − 1)$, the final macroblocks are divided up unevenly between the number of slave cores.
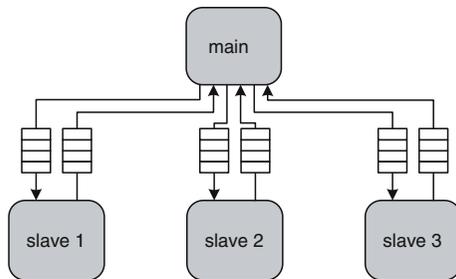
Fig. 3.   A four core master/slave system.

Each system detailed below follows the basic three-core model, with extra slave cores only allowing the data to be divided up further. Each slave core performs the same functions on the data that it is given.

Three Cores: For a three-core system, one main (producer) core organizes the distribution of data between the two slave cores, and also completes the input and output file manipulations, as well as the Huffman encoding. Two slave cores communicate with the main core via TIE queues, and only complete that part of the encoding process that can be parallelized. In particular, as shown in Table II, these functions are the *RGB to YCbCr conversion* function, the *LevelShift* function, the *DCT* function, and the *Quantization/Zigzag* function. Four TIE queue interfaces are used to communicate between the main core and Slave1, and the main core and Slave2. After the RGB to YCbCr conversion, the main core sends the first macroblock of data to Slave1, and the second macroblock to Slave2. The slave cores then process their share of the data before sending the processed data back to the main core. After Huffman encoding the data, the main core then sends the next two macroblocks to the slave cores and the cycle continues.

Four Cores: Similarly to the three core system, one main (producer) core organizes the distribution of data between three cores. This main core communicates with each slave core via TIE queues as shown in Fig. 3. Six such TIE queues are used, with three input and three output queue interfaces on the main core, and one input and one output queue interface on each slave core.

Every three macroblocks of the input image file are divided up and sent to the three slave cores in order. This is visualised in Fig. 3. Macroblocks 1, 4, 7, etc., are processed by Slave1; macroblocks 2, 5, 8, etc., are processed by Slave2; and macroblocks 3, 6, 9, etc., are processed by Slave3. Note that for higher numbers of cores in the system, the ordered blocks to be processed is $(N-1)k + x$ for *Slave<x>*, where N is the total

number of cores, k is an integer value, and x is the particular slave core number. The rest proceeds as per the three processor system.

As shown in Table II, each slave core implements the same functions (those which could be parallelized) whereas the main core implements the input and output file operations as well as other initializations, writing markers, and the Huffman encoding.

Five, Six, and Seven Cores: Extending the four core system, five, six, and seven cores are now used, where macroblocks are grouped into four, five, and six destination groups, respectively, and then sent to the designated slave cores.

See Table II for details of functions processed on each core. Each extra slave core implements the same functions as all other slave cores, but with less data. The only differences are that the RAW image file is divided into more parts to be distributed to more cores during the *RGB to YCbCr conversion* stage, and each slave core processes fewer macroblocks of data during the *LevelShift*, *DCT*, and *Quantization* stages.

4.2. Method II

We next investigate a different multiprocessor architecture in a pipeline configuration. The system is comprised of different processors, each running a portion of a pipeline stage of a program. Each processor has a possibility of being configured optimally, instantiating only those resources which are appropriate for the particular stage of the pipeline. A heterogeneous processor system minimizes the redundancy of resources, as processors with complex computations may be parameterized with more resources. Communication among processors is facilitated using ports and queues which are provided by the Xtensa LX[20] processor architecture.

The multiprocessor pipeline architecture design requires programs which can be broken up into computationally independent blocks. This resembles computational blocks in a pipeline processor architecture.[26] Transfer of data from one processor to another is facilitated by a *queue*.

In the case of the JPEG encoder, the pipeline architecture is ideal as the JPEG encoder displays characteristics of a pipeline nature. The encoding process is divided into stages which are independent of each other. The proposed architecture consists of standalone processors which run sections of the JPEG encoder program which have been recompiled as individual programs. These subprograms which reside in these processors accept data via the queues of the Xtensa processor, perform the necessary computation, and finally push it to the output queue into the next stage of the pipeline. The computed data traverses the pipeline stages until it is finally written out to file by the last processor in the pipeline. It should be noted
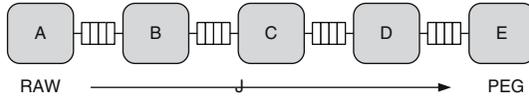
Fig. 4. A five core system interconnected by queues. Each processor is assigned a stage of the JPEG pipeline.

that while one processor is computing a workload, the rest of the system is still busy processing workloads for other stages.

The scalability of a multiprocessor pipeline architecture depends entirely on the suitability of the targeted program's data structure and control flow. A particular configuration is considered efficient when all processors have equal computational workload (i.e., no processors in the pipeline should be waiting for the next stage to complete).

Five Cores: The JPEG encoding process (Fig. 1) exhibits sequential routines which can be broken up into stages, thus allowing the possibility of pipelining the encoding process. These stages represents functions within the original program and is extracted and compiled as a single program which is executed in a single core.

The main program of the encoder sends the required information to the appropriate stages of the pipeline in order to initialize the quantization tables and JFIF[25] headers which are written out to a file. As each pipeline stage only has to wait for the data from the previous stage, the partition program is constructed such that one core reads the RAW image while another writes the encoded JPEG into a new file. Each stage processes data at a macroblock level (i.e., 8×8 pixels).

We start with a five core multiprocessor configuration (Fig. 4), pipelined into five major stages. The quantization table initialization code shares the same core as that which implements the quantization stage of the pipeline. This stage receives initial values from the main program (core A) which reads in encoding parameters that define the quality of the resulting image. Core D has the necessary code to initiate the writing of JFIF markers and closing the JPEG bit stream. The last core (Core E) is initialized by the first core with the name of the output file and writes any receiving bytes from the previous stage (Core D) to file. Table IV summarizes the allocated stages to the respective cores.

Six Cores: We next introduce a new core into the system and allocate the *LevelShift* stage to the new processor (refer to Table IV). The Level-Shift stage accepts YCbCr values from the previous stage, level shifts the values and then pushes it out to the queue in macroblocks of 64 Y's, 64 Cb's, and 64 Cr's. As will be shown in Section 6, the introduction of this stage into the pipeline does not increase the overall performance of the encoding process.

Table III.   Utilization in a Nine Core Multipipeline system

	Cores								
	A	B	C	D	E	F	G	H	I
Utilization (%)	76	51	51	51	28	28	28	95	99

Seven Cores: DCT transformations are known to be very computation intensive and there are special circuits which performs just such a function. In our next approach, the two-dimensional *DCT* function can be split up into two stages; a one-dimensional *DCT* vertically and a one dimensional *DCT* horizontally. A seven core processor configuration benefits from such an approach (refer to Table IV).

Multiple pipelines: The rationale of having pipeline implementations is to increase throughput during execution. Similar to a pipeline microprocessor, a pipeline implementation of a JPEG encoder would be able to encode images at a faster rate.

Combining both approaches of method I (parallel computations of macroblocks) and method II (pipelining), we are able to exploit further parallelism within the JPEG compression algorithm. The pipelining nature of the previous core systems are maintained. However, from stage four onwards (DCT), macroblocks for luminance (Y), chrominance red (Cr), and chrominance blue (Cb) are processed in separate parallel pipelines. This reduces the processing bottleneck in the DCT and quantization/zigzag stages.

These parallel pipelines include *DCT* and *Quantization* (with *zigzag*) (QZ) stages. The outputs of these parallel pipelines then converge into a single pipe where *Huffman* encoding is performed. *Huffman* encoding depends on serial input and thus, cannot process separated JPEG streams independently.

Nine Cores: Following the five pipeline stage multiprocessor approach in Method II, we try to increase the throughput of the middle stages of the JPEG compression pipeline by replicating the *Quantization* stage of the pipeline due to heavy utilization rates in the *DCT* and *Quantization* stages (refer to Table IV). Pipeline flow diverges only at stages four and five (refer to Fig. 1) into three separate pipeline flows, before being fed into a single processor during the *Quantization* stage. *E*, *F*, and *G* cores are the *Quantization* stages and process the Y, Cr, and Cb macroblocks separately. These cores initialize their respective quantization tables (Y, Cr, Cb). This results in a nine core processor system.

The utilization of each core in the system is shown in Table III.

Table IV.   Processor Configuration with Multiple Pipeline Flows

Cores	Stages (single pipeline)			Stages (multiple pipelines)	
	Five cores	Six cores	Seven cores	Seven cores	Nine cores
A	1, 2, 3	1, 2	1, 2	1, 2, 3	1, 2, 3
B	4, 5	3	3	4, 5	4, 5
C	6, 9	4, 5	4	4, 5	4, 5
D	7, 10, 11	6, 9	5	4, 5	4, 5
E	8	7, 10, 11	6, 9	6, 9	6, 9
F	–	8	7, 10, 11	7, 10, 11	6, 9
G	–	–	8	8	6, 9
H	–	–	–	–	7, 10, 11
I	–	–	–	–	8

Seven Cores: Table III shows that the utilization rates of the *Quantization* cores (*E*, *F*, and *G*) in the nine core system are very low, prompting us to replace all three cores with just one. This is due to the bottleneck in the second last stage of the pipeline (*H*), whereby the *Huffman* encoding has already reached its maximum throughput. Note that Stage *I* is not considered since it is constantly looping. Outputs from the three separate *DCT* cores are now channeled into a single core which will perform quantization and zigzag transformations.

With the seven core multiprocessor system, we have methodologically reduced the area consumption of the system. Based on the utilization rates of each core in the pipeline, we were able to selectively optimize the required cores using Tensilica's XPRES compiler, which automatically generates TIE instructions (SIMD, FLIX, vector, fusion).

When the runtime of the selectively optimized system closely matches the fully XPRES compiled version, we replace the cores which have very low utilization rates with simpler ones. These include replacing LX1 cores with *LX2* cores (refer to Section 3.2) and progressively reducing the instruction and data caches (until they reach the same performance of the fully XPRES compiled version, or reach the minimal configuration of 1 kB). This methodology results in a heterogeneous multiprocessor system which provides high-performance improvement to area increase ratio (Table IV).

5. EXPERIMENTAL METHODOLOGY

We used Tensilica's Xtensa RA2006.4 Toolset for the Xtensa LX family of processors. The toolset also provides a set of compilation tools to compile C/C++ code, targeted to our specially configured Xtensa LX

cores (refer to Section 3.2). The Tensilica Instruction Set Simulator (ISS) and XTMP environment were used to run the multicore systems. The XTMP is an API and runtime environment for rapid multiprocessor description and analysis. XTMP utilizes its own simulation engine and generates SystemC-compatible models.

For each system, multiple Xtensa cores were instantiated and XTMP was used to connect them to peripherals and interconnect. The ISS directly models the Xtensa pipeline and operated as a system-simulation component using the XTMP environment. With XTMP, different multi-processor configurations could be simulated in a short amount of time.

The simulator allows for communication between the cores and peripherals using a cycle-accurate, split-transaction simulation model without using a clock. The ISS was used to generate profiling data for all cores in the system, which were then profiled using Tensilica's *gprof* profiler. The profiles can include the cycle counts for all functions executed by the cores. The ISS can also print a summary of the total cycle count and global stalls of each core.

Each individual core is connected via the queue interface provided by the Xtensa LX core using the XTMP environment. We create C-code functions and data structures to model the queues within the XTMP environment. The queues are simple FIFO (first-in, first-out) components that mainly operate via the functions *push* and *pop* called by each of the connected cores in the simulation environment. Queues transmitting RAW bit streams between processors are modeled to have 64 entries. A full queue or an empty queue would effectively stall the section of the pipeline (Fig. 5).

We created multicore processor systems by identifying hotspots within the single processor benchmark application (see Fig. 6(a)). The hotspots
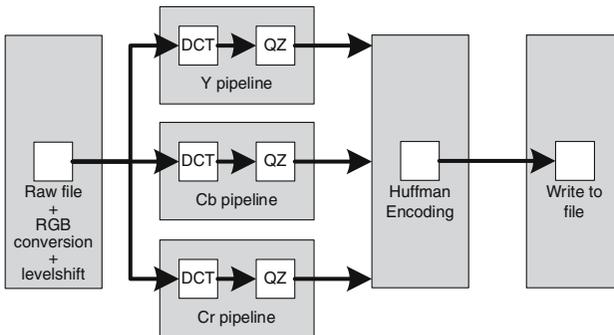

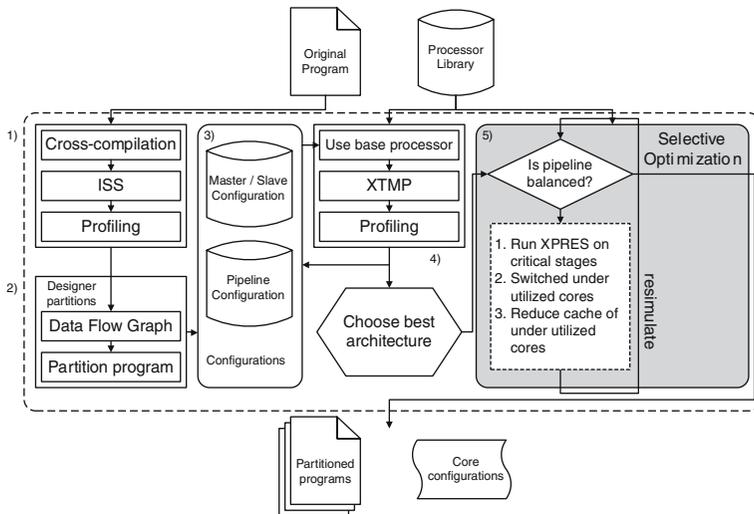Fig. 5.   A nine core system with three internal pipeline flows.

Fig. 6.   Experiment methodology.

were identified by cross-compiling the benchmark application using the Tensilica Xtensa LX compilation tools and by simulating it on a selected configuration, namely *LX1* (refer to Table I). The hotspots were mainly functions identified in Section 3.1. We partition and allocate these functions based on the methodology defined in Section 4 (see Fig. 6(b)). Two sets of configurations were created manually namely, the Master/Slave and Sequential Pipeline configurations (see Fig. 6(c)). An XTMP simulation program, specially customized to generate profiling and other relevant benchmark information is created for each of these multiprocessor systems, which are then simulated and the performance and area utilization recorded (see Fig. 6(d)).

An architecture which has the best performance increase per area increase ratio is selected for further optimization. Our approach manually investigates the most appropriate stage to optimize one step at a time. The selective optimization would strive to produce an architectural configuration which has well-balanced utilization among all stages in the pipeline (see Fig. 6(e)). The generated systems include similar configured cores, not including parameterized components such as the number of outgoing and incoming queues.

The toolset also includes the XPRES (Xtensa PRocessor Extension Synthesis) compiler which creates tailored processor descriptions for the Xtensa processors from native C/C++ code. The XPRES Compiler was used to create custom instructions (by generating RTL models) for each

core in the system. Using the designer-defined input of C programs to be analyzed, XPRES extends the base processor with new instructions, operations, and register files using TIE extensions. It does so by automatically generating a new TIE file which can be included when recompiling the source code. XPRES was used to create a distinct TIE file for each core in each system, to optimize each individual core using only the C files that are used on a particular core. Each individual core in the multiprocessor system is compiled through XPRES to explore the extent of improvements that can be obtained via extended instructions.

Area counts include the base processor, instruction and data caches and the TIE instructions. Each multiprocessor system generated in the case study reads a RAW file and saves it as a JPEG format file. The file generated is viewable by any standard image viewing application.

6. RESULTS AND ANALYSIS

Figure 7 shows the runtime improvements and area increase with respect to the original core, *LX1* (refer to Section 3.2). The graph shows the three main architectures used in this case study; master/slave architecture (3–7 cores), pipeline architecture (5–7 cores) and multipipeline architecture (7 and 9 cores). It can be observed that the area increase to runtime improvement ratios for each of the multiprocessor systems have values more than one and actually increases as more processors are added to the system. With multiprocessor pipeline systems, the improvements seem to level off for seven cores and above. This is due to the fully saturated Huffman encoding stage. Unless the Huffman stage could be further partitioned and parallelized, this would remain a critical stage in the pipeline.
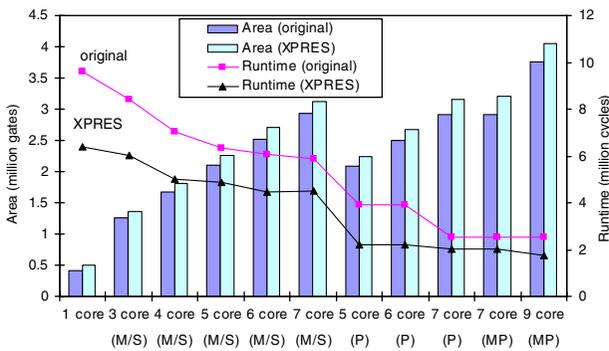

Fig. 7. Performance of multiprocessor systems without optimizations.

As a form of measurement, the systems have been compiled with XPRES to obtain maximum improvement if all cores were optimized. The maximum performance improvement is obtained from the nine processor system, with a performance increase of 3.8×; and 4.7× when run through the XPRES compiler for each of the nine processors (refer to Fig. 7).

It is not viable to continue this approach of adding processors to improve performance, as area increases faster than improvement in performance. However, by reducing resources on non-critical processors, we can reduce area, yet keep the same amount of performance. We selectively optimized critical stages of the pipeline.

We selected the seven core multipipeline system for further optimizations as it performs almost as well as the nine core architecture while using much less area. Figure 8 shows the utilization of each of the seven cores in a multipipeline architecture (refer to Section 4.2). The first two graphs in Fig. 8(b) on the left shows the utilization of the system without optimizations and with XPRES optimizations, respectively. Area increase is 7× and 7.7×, respectively, with performance improvement of 3.8× and 4.7× relative to the base processor implementation (represented by the decreasing and steady lines in the graph).

It should be noted that the last pipeline stage is always at 100% utilization due to its software implementation which repeatedly polls for incoming data on every simulation cycle. In *Partial XPRES 1*, we replaced the original core of the Huffman encoding pipeline stage with an XPRES version. This optimization step moves the critical stage to another stage. In *Partial XPRES 1*, the critical path has moved on to the *Quantization* stage. We replace the Quantization pipeline stage with an XPRES version in *Partial XPRES 2*, once again, making Huffman encoding a critical stage. In this implementation, it can be seen that the parallel pipelines of *DCT* stages are not fully utilized. We replaced these cores with *LX2* cores, resulting in a utilization jump from 62.3% to 88.3% while area is reduced from 7.1× to 3.8×. At this point, we achieve an area increase to performance improvement ratio of 0.82. Further optimizations were achieved when reducing the cache sizes of the first core in the pipeline from 32 kB to 1 kB. This does not significantly affect performance as the core mainly reads the RAW files and outputs it to the pipeline. The ratio is further reduced to 0.68 while still maintaining a performance improvement of 4.62×.

A complete design space exploration is not feasible due to the huge number of combinations which can be obtained when each stage in the pipeline is configured independently (cache configuration ranges from 1 kB to 32 kB for each instruction and data cache, base instruction set of 80 instructions is extensible to include MAC, coprocessors, SIMD
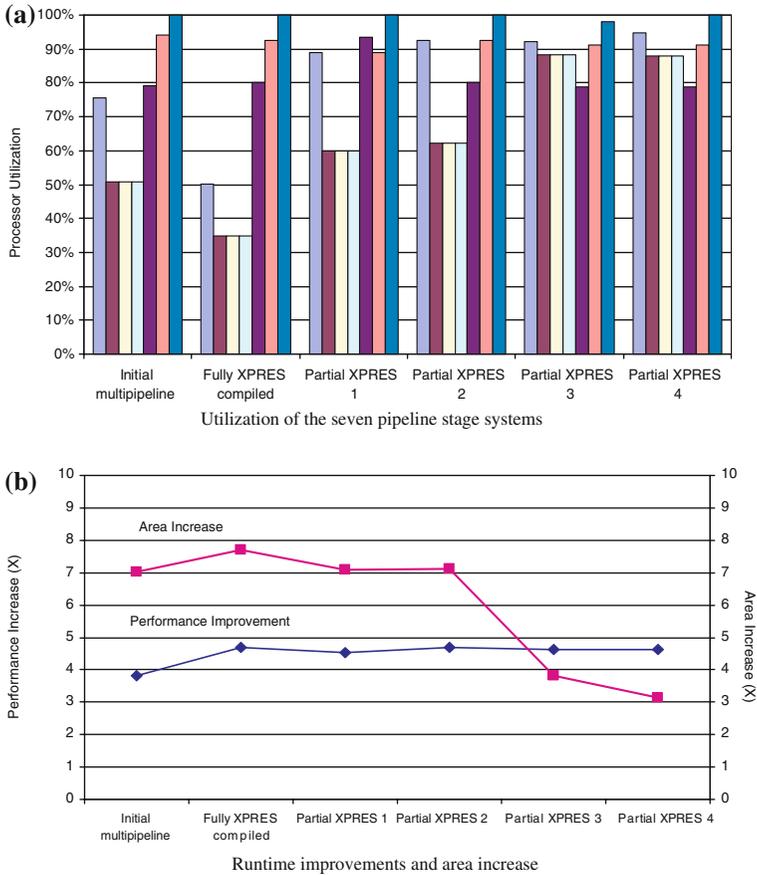
Fig. 8.   Selective optimization on a seven core multipipeline multiprocessor system.

instructions, and custom instructions). This would require vast amount of time to simulate each and every configuration. However, certain stages in the pipeline (i.e., critical stages) should be optimized first compared to the others. Thus, the selective approach to optimization as shown in Fig. 6 is used. This method has greatly reduced the time needed to find a configuration which produces a good area increase to performance improvement ratio without optimization.

To obtain a holistic view of the optimization problem, an approximation of the complete design space is shown in Fig. 9. The figure shows the performance improvement of the various configurations. These configurations include the *LX1* (XPRESed and non-XPRESed versions) and *LX2* cores with various cache configurations. The line in Fig. 9 denotes the
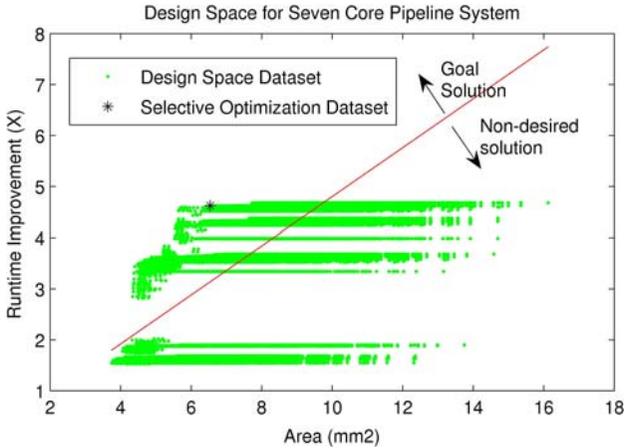
Fig. 9.   Design space for JPEG encoder.

section where the performance increase equals the area increase relative to the single processor benchmark configuration (refer to Section 3.2). Runtime improvements above this line represent configurations which have a good area increase to performance improvement ratio. Data sets appearing below the linear line are configurations which are not fully utilized and thus, have the possibility of being down-scaled to decrease area use while maintaining performance improvement. It is also noted that our selective optimization method manages to closely match the maximum performance improvement in the whole design space.

A thorough design space exploration would have taken months of simulation. The approximation in Fig. 9 was obtained by simulating and profiling each stage of the pipeline independently. The runtime of a particular configuration is

$$\mathbf{R} = R^{\text{init}}(v_1) + R^{\text{process}}(v_{crit}) + R^{\text{final}}(v_J), \tag{1}$$

where $R^{\text{init}}(v_1)$ is the initialization stage of the first stage of the pipeline, $R^{\text{process}}(v_{crit})$ is the time of the longest execution time of a kernel in the pipeline, and $R^{\text{final}}(v_J)$ is the finalization time of the last stage of the pipeline.

The equation is used to permute the various configurations and to calculate the total runtime of a set of configurations (i.e., different stages have different configurations). Such calculations would result in a slight error (less than 2%), though this would not affect the overall trend of the graph.

Extending the selective optimization technique at the beginning of this section, a formal methodology could be developed to automatically select the appropriate configuration for the critical stage of the pipeline after each configuration change. This would be equivalent to filtering out the redundant configurations which would otherwise worsen the area increase to performance improvement ratio.

7. CONCLUSION

In conclusion, we have performed an interesting case study by exploring the use of multiple cores in master/slave and pipeline configurations. Communications amongst these cores are facilitated using queues which are introduced in Tensilica's Xtensa LX[20] configurable cores. We have also analyzed the effect of increasing the number of cores in the system, to analyze the achievable performance improvement. The XPRES tool has been used to selectively optimize over-utilized cores, while under-utilized cores were replaced by cores with less resources. The result of the opimization is contrasted against the design space of the benchmark application. Our selective optimization approach closely matches the maximum performance gain achievable in the overall design space. We have found that such a multicore architecture has the potential to minimize the area of cores which requires less computation in the system. The pipeline architecture can be exploited to provide an increase in peformance which has the possibility to far outweigh the increase in area. We have shown that a heterogeneous multiprocessor system is able to provide the necessary speedup while minimizing gate count, providing a very low area increase to performance improvement ratio.

REFERENCES

1. R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, Heterogeneous Chip Multiprocessors, *Computer*, **38**(11):32–38 (2005).
2. D. Pham *et al.*, The Design and Implementation of a First-generation Cell Processor, in *Proc. of the ISSCC 2005*, IEEE CS Press, pp. 184–186 (2005).
3. T. D. Braun, H. J. Siegel, and A. A. Maciejewski, Heterogeneous computing: Goals, Methods, and Open Problems, in *Proc. of the HiPC 2001*, Hyderabad, India, Springer, Berlin, Vol. 2228, pp. 302–320 (2001).
4. J. Axelsson, A Case Study in Heterogeneous Implementation of Automotive Real-Time Systems, in *Proc. of the CODES'98*, Seattle (1998).
5. M. T. J. Strik, A. H. Timmer, J. L. van Meerbergen, and G.-J. van Rootselaar, Heterogeneous Multiprocessor for the Management of Real-time Video and Graphics Streams, *IEEE J. Solid-State Circuits*, **35**(11):1722–1731 (2000).

6. N. Zhang and C.-H. Wu, Study on Adaptive Job Assignment for Multiprocessor Implementation of MPEG2 Video Encoding, *IEEE Trans. Ind. Electron.* **44**(5):726–734 (1997).
7. A. Berić, Ramanathan Sethuraman, Carlos Alba Pinto, Harm Peters, Gerard Veldman, Peter van de Haar, and Marc Duranton, Heterogeneous Multiprocessor for High Definition Video, in *Proc of the ICCE'06*, pp. 401–402 (2006).
8. S. Gopalakrishnan and M. Caccamo, Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems, in *Proc of the RTAS'06*, pp. 199–207 (2006).
9. S. Baruah, Task Partitioning upon Heterogeneous Multiprocessor Platforms, in *Proc of the RTAS'04*, pp. 536–543 (2004).
10. M. Kim, D. Kim, and G. E. Sobelman, MPEG-4 Performance Analysis for a CDMA Network-on-chip, in *Proc of the 2005 International Conference on Communications, Circuits and Systems, 2005*, pp. 493–496 (2005).
11. A. Wieferink, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, G. Braun, and A. Nohl, System Level Processor/Communication Co-exploration Methodology for Multiprocessor System-on-Chip Platforms, *Comput. Digit. Tech. IEE Proc.* **152**(1):3–11 (2005).
12. V. Stefan V. Živojnović, S Pees, and H. Myer, LISA-machine Description Language and Generic Machine Model for HW/SW Co-design, in *Workshop on VLSI Signal Processing*, pp. 127–136 (1996).
13. SystemC Initiative. (http://www.systemc.org).
14. K. S. Chatha and R. Vemuri, A Tool for Partitioning and Pipelined Scheduling of Hardware-Software Systems, in *Proc. of the 11th International Symposium on System Synthesis, 1998*, Hsinchu, pp. 145–151 (1998).
15. J. Jeon and K. Choi, Loop Pipelining in Hardware-Software Partitioning, in *Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific*, Yokohama, Japan, pp. 361–366 (1998).
16. T. Kodaka, K. Kimura, and H. Kasahara, Multigrain Parallel Processing for JPEG Encoding on a Single Chip Multiprocessor, in *Proc. of the IWIA'02*, pp. 57–63 (2002).
17. S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, Macro Pipelining Based Scheduling on High Performance Heterogeneous Multiprocessor Systems, *IEEE Trans. Signal Process.* **43**(6):1468–1484 (1995).
18. T. A. Giuma and K. W. Hart, Microcomputer Bus Architectures, in *Southcon Conference*, Orlando, FL, pp. 431–437 (1996).
19. Independent JPEG Group. IJG (http://www.ijg.org).
20. Xtensa Processor. Tensilica Inc. (http://www.tensilica.com).
21. F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, Custom-instruction synthesis for extensible-processor platforms, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **23**(2):216–228 (2004).
22. Flix: Fast relief for performance-hungry embedded applications, Tensilica Inc. (http://www.tensilica.com/pdf/FLIX_White_Paper_v2.pdf) (2005).
23. K.-C. Huang and F.-J. Wang, Design Patterns for Parallel Computations of Master-Slave Model, in *Proc. of the International Conference on Information, Communications and Signal Processing*, Vol. 3, pp. 1508–1512 (1997).
24. T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ (1992).
25. E. Hamilton, JPEG File Interchange Format. Technical report, C-Cube Microsystems, September 1 (1992).
26. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd Ed., Morgan Kaufmann Publishers, Los Atlos, CA (2003).