Novel Architecture for Loop Acceleration : A Case Study

Seng Lin Shee†, Sri Parameswaran†, Newton Cheung‡

†School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia
†National Information and Communications Technology Australia (NICTA), Sydney, Australia*
‡VaST Systems Technology, Sydney, Australia

{senglin, sridevan}@cse.unsw.edu.au   n.cheung@vastsystems.com.au

Abstract

In this paper, we show a novel approach to accelerate loops by tightly coupling a coprocessor to an ASIP. Latency hiding is used to exploit the parallelism available in this architecture. To illustrate the advantages of this approach, we investigate a JPEG encoding algorithm and accelerate one of its loop by implementing it in a coprocessor. We contrast the acceleration by implementing the critical segment as two different coprocessors and a set of customized instructions. The two different coprocessor approaches are: a high-level synthesis (HLS) approach; and a custom coprocessor approach. The HLS approach provides a faster method of generating coprocessors. We show that a loop performance improvement of 2.57x is achieved using the custom coprocessor approach, compared to 1.58x for the HLS approach and 1.33x for the customized instruction approach compared with just the main processor. Respective energy savings within the loop are 57%, 28% and 19%.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Performance, Design, Experimentation

Keywords

Hardware/software partitioning, coprocessor, architecture, ASIP, loop optimization, loop pipelining, loop acceleration, latency hiding, tightly coupled

1. Introduction

The industry has long sought to improve performance of systems to satisfy increasing computational demands of consumer applications. Such demands have been traditionally met by using more powerful microprocessors, using custom designed ASICs or by the use of special purpose techniques. Through the years, techniques have been developed to achieve exponential speedup for microprocessors. Such techniques [19] include caching, pipelining, instruction-level parallelism and superscalar architectures.

General Purpose Processors (GPPs) are programmable but consume more power than ASICs. Application Specific Integrated Circuits (ASICs) are designed for specific applications, where the area and performance can be easily optimized but are costly to design and are non-programmable, making upgradability an impossible task.

Application Specific Instruction-set Processors (ASIPs) combine the flexibility of GPPs and the performance of ASICs by replacing under-utilized resources by components which can directly improve the performance of a particular application. An ASIP will execute an application for which it was designed for with great efficiency (although they are capable of executing any other program - albeit with reduced performance). ASIPs are suitable for systems which require customization to improve performance, and removal of unnecessary components to reduce area utilization and power consumption. The flexibility that an ASIP inherits from a programmable base processor provides it with the ability for future upgrades. Each new instruction and component added or removed would change the existing Instruction Set Architecture (ISA) for the application, thus the name: *Application Specific Instruction-set Processors*. Tools such as Nios [1], ARCtangent [2], Jazz [5], SP5-flex [7], Tensilica [8] and ASIPmeister [22] allow the rapid creation of ASIPs.

Motivation

The current trend to speedup ASIPs is to provide extended instructions [8, 34] to execute a group of instructions frequently occuring in the code. Coprocessors are also used when the application needs special functionality such as floating point arithmetic and sum-absolute-difference instructions. The authors in [15] have shown that supplementing an ASIP with a dedicated coprocessor will reduce power consumption but with less flexibility if the algorithm has to be changed.

In this paper, we show a novel method to accelerate a loop by attaching a coprocessor to an ASIP, which is specially designed for the JPEG encoder from the Mediabench [25] benchmark suite. We start off with the base processor which is generated using the approach in [30] and is based on the Portable ISA (PISA), making it possible to integrate new architectures due to the freely available source design. The JPEG encoder is profiled and the loop segment which has the highest optimization potential (Huffman encoder) is selected for acceleration and converted to a coprocessor. A novel coprocessor architecture is introduced. We then show how parallelism can be achieved when a coprocessor assists the base processor during the loop execution. Performance is mainly achieved through loop pipelining [11, 23] and memory latency hiding [20]. We create a coprocessor in two different ways and compare the performance, area and power improvements. The first method uses an off the shelf HLS system to create a coprocessor using a C-to-VHDL conversion and the second method uses a custom method to create a coprocessor. To compare with the above coprocessor ap-

*National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

proaches, we also accelerate the application by implementing the loop as customized instructions.

The rest of this paper is organized as follows: Section 2 gives a summary of the existing works on coprocessors and loop accelerations; Section 3 describes the JPEG benchmark program and how a loop is chosen to be implemented in a coprocessor; Section 4 provides a detailed approach of using the HLS framework in the coprocessor creation; Section 5 explores the custom architectural approach of implementing such a coprocessor; Section 6 provides a detail analysis of both of the coprocessor architectures used; Section 7 describes the experimental procedures and the tools used to obtain the results; Section 8 describes the flow for power, area and performance analysis, and presents the results; and Section 9 concludes the paper.

2. Related Work

Coprocessors have been used in applications to speed up computation, offloading much of the work performed by the main processor in the system. They come in many flavors (e.g. instruction based, functional based, SIMD based and vector processor based). Typical coprocessor examples are: graphic accelerators [24, 37]; numeric & floating point units (FPUs) [26]; Digital Signal Processors (DSPs) [27]; and Multiply & Accumulate (MAC) units [9].

Early approaches to hardware-software partitioning was demonstrated in [14, 16, 29] by the extraction of loop segments into coprocessors. However, most of these approaches often did not fully achieve maximum possible performance improvements.

In [33], the authors implemented a framework to profile a program dynamically. The executed loops are detected via an onboard hardware profiler, decompiled and synthesized onto an FPGA coprocessor in the SOC. A dynamic partitioning approach is used to extract the appropriate loops. However, the system has a limited amount of memory to cache the loops and thus deals with much smaller regions of code. The framework only executes a single cycle loop body and the number of iterations of the loop has to be determined before the loop executes.

CriticalBlue [13] provides a complete methodology with a toolset for converting functions to individual coprocessors. Being software programmable, the coprocessors generated by the system have some flexibility to accommodate changes to standards.

The authors in [28] proposed a processor-coprocessor architecture for high-end video applications. High level synthesis was used to map algorithms in the application to the coprocessor and to minimize the number of computing units on the chip. For example, the number of ALUs were increased until the ALUs were no longer the bottleneck. However, the work did not address how the overall program is actually executed on the system.

In contrast to the above approaches which utilizes dedicated buses, [18, 36, 31] explored closely coupled components with the host CPU using FPGAs as the reconfigurable components. Extended instructions could be created and performed by these blocks which share the register files and pipeline registers of the host CPU. However, performance is limited to the higher latency FPGAs with respect to the ASIC host processor. Similar to this category of tightly coupled components, our work explores a new architecture for accelerating loops via a coprocessor component in tandem with the host CPU. The coprocessor would run at the same clock rate as the host CPU as it is also implemented as an ASIC.

As far as JPEG and image processing are concerned, accelerations have been achieved through the use of hardware IP cores. These specialized chips [6, 21, 32] are ASIC implementations of their software counterparts [4, 25]. The hardware encoders are fast and efficient but highly inflexible if major changes are made to the system. A slightly flexible approach would be processors which use loop accelerators [24, 27, 37] for image processing purposes. Such approaches require existing programs to be modified extensively to work with coprocessors.

Our approach differs from the above codesign works in the following ways:

1. we introduce a hardware software codesign architecture with a tightly coupled coprocessor which shares the main processor's register file; and
2. we show a latency hiding approach (taking advantage of the tightly coupled architecture) for the first time in an ASIP environment to enhance application performance.

In order to show the efficacy of our approach, we performed a case study using customized instructions, high-level synthesis approach and a custom coprocessor approach to analyze performance, power and area utilization.

3. The JPEG Encoder

The JPEG compression algorithm which is being used in this case study is a lossy compression scheme which removes redundant information unseen to the human eye. This scheme has its advantages for naturally occurring images which have a variety of shades. This algorithm is ubiquitous in most digital imaging products.

The benchmark program accepts a Portable PixMap (ppm) image (raw file). It reads the file, together with other parameters and outputs the corresponding JPEG file. The application in general has two main sections: lossy compression stage (DCT transformation + quantization) and the lossless compression stage (Huffman encoding). It cannot be easily determined from the source of the program about how the program is structured due to the complex nature of function calls. Thus, a designer would need to entirely understand the program and algorithm in order to produce a fully customized coprocessor for such an application.

3.1 Loop Identification

The benchmark program is profiled using tools we developed to support the current ISA as well as to extract the neccessary information not provided by other tools. We created a tool based on a loop detection scheme proposed in [35]. We then performed a detailed profile of the inner most loops in the program against a set of RAW images of different sizes.

Ideally, the theoretical maximum speedup gained from optimizing the loop would be the percentage of program runtime with respect to the new program runtime after removal of the loop runtime. This assumes that the loop execution can be eliminated completely. However, we choose a more realistic definition. We assume that the theoretical maximum improvement is achieved when all non-memory operations are eliminated completely. This is when all computations (non-memory operations) are moved to the coprocessor (see Section 4.1). The profiling stage detects the loop instructions executed (LIE), the memory operations executed in the loop ($Memory\ LIE$) and the total number of instructions executed (TIE). TIE consists of LIE and instructions in the rest of the code.

The theoretical maximum improvement (TMI) is:

$$TMI = \left(\frac{TIE}{TIE - (LIE - Memory\ LIE)} - 1\right) \times 100\% \quad (1)$$

where $TIE = LIE + non\text{-}LIE$

The TMI determined will be used to select the loop which has the greatest potential to achieve the highest speedup in our loop acceleration case study. For the image *rose.ppm* image provided with the benchmark application, which has a resolution of 227×149

(33,823) pixels, Table 1 shows the seven most critical loops in the JPEG encoder. The last three columns in the table shows the percentage runtime of the loop with respect to the whole program, the percentage of non-memory operations in the loop and the theoretical maximum improvement which can be obtained. The loop starting at line 643 of jcphuff.c has the highest TMI value and is thus selected for our case study.

Loops	*Cycles*	*RT%*	*COMP%*	*TMI%*
jcphuff.c:643	**4213978**	**19.77**	**78.82**	**18.46**
jcdctmgr.c:232	1329496	6.24	87.71	5.79
jfdctint.c:220	1101194	5.17	90.11	4.88
jfdctint.c:155	1087578	5.10	89.98	4.81
jchuff.c:766	982554	4.61	91.65	4.41
jchuff.c:684	509413	2.39	91.73	2.24
jchuff.c:673	508646	2.39	91.71	2.24

Table 1: Loop runtimes

4. High-level Synthesis Approach

A high-level synthesis approach is used to convert the loop written in ANSI-C code to synthesizable VHDL code. Slight modifications are performed to the loop code before the VHDL component can be generated. Each I/O pin of the generated component corresponds to the inputs and outputs defined as arguments to the loop function definition (see Figure 2). These functional blocks can be used as functional units which provide extra processing power to a System-on-Chip (SOC) Architecture.

The VHDL component has a *start* signal to control the execution of the component and a *done* signal to indicate end of execution. The HLS component has no memory location awareness and thus expects input values to be available at its pins at time of execution.

A high-level synthesis approach was initially used as it provides an option to unroll loops. Loop unrolling dramatically improves parallelism in the loop while reducing the number of conditional branches being executed which is found at the end of every loop segment. However, loop unrolling would only be beneficial if multiple resources can be utilized in parallel. In a single pipeline processor system like ours, parallelism through loop unrolling was found to be overly ambitious, as the bottleneck is the serially executed memory operation by the base processor.

4.1 Architecture

```
EOB = 0;
for (k = cinfo->Ss; k <= Se; k++) {
    temp = (*block)[jpeg_natural_order[k]];
    if (temp < 0)
        temp = -temp;
    temp >>= Al;
    absvalues[k] = temp;
    if (temp == 1)
        EOB = k;
}
```

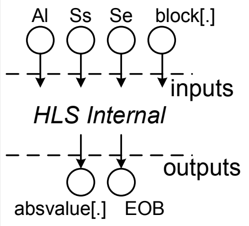


Figure 1: Example code segment & coprocessor interface

The loop segment which we have selected in Section 3.1 is shown in Figure 1. Variables are identified to be either inputs or outputs to the loop. The loop is then wrapped up as a function with input variables being defined as inputs to the function and output variables being defined as pointers passed to the function. The modified function designed for the HLS framework is shown in Figure 2. In order to fetch values, a wrapper was made to fetch the right variables (i.e. the wrapper sends the proper address signal to the address line of the read port). We have chosen to connect the inputs of the HLS component to the read register ports of the general purpose register (GPR) and output ports of the component to the write register ports of the GPR.

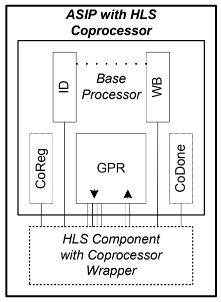
```
void prepass(int Al, int Ss, int Se, int blocks, int *absvalues, int *EOB) {
    int k;
    register int temp;
    *EOB = 0;
        for (k = Ss; k <= Se; k++) {
            temp = blocks;
            if (temp < 0) temp = −temp;
            temp >>= Al;
            *absvalues = temp;
            if (temp == 1) *EOB = k;
        }
}
```

Figure 2: Modified loop as an ANSI-C function

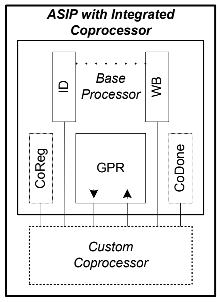
In this HLS approach, the number of register read / write ports depends upon the number of parameters the loop function has. A function with many parameters, requires a large number of GPR ports.

Both the coprocessor and base processor share the same register file in the system. Our selected loop segment has 4 inputs and 2 outputs, thus requiring us to connect the necessary lines directly to the register file. Originally, the base processor has a register file with 4 read ports and 2 write ports. As these ports are used by the existing components in the pipeline, additional ports have to be assigned to the coprocessor. This results in an "8-read, 4-write" register file to be created for the integration of the HLS coprocessor into our design.

Figure 3(a) shows how the HLS based coprocessor is integrated into the existing design. We have introduced two registers which can be accessed by the base processor. One bit from the COREG register is connected to the *start* signal of the coprocessor wrapper. The remaining bits can be connected to additional coprocessors of the same architecture. The CODONE register *set* signal is connected to the *done* signal of the coprocessor. This register would be used to signal the base processor when the coprocessor has finished executing the loop.


ASIP with HLS Coprocessor
ID
Base Processor
WB
CoReg
GPR
CoDone
HLS Component with Coprocessor Wrapper

(a) HLS Coprocessor has many lines connected to the GPR



(b) The Custom Coprocessor has only one read port and one write port connected to the GPR

Figure 3: Coprocessor Integration

Two new instructions are added into the existing instruction set: SCPR (set coprocessor) is used to set the specific bits in the COREG register, which in turn asserts the start signal of the coprocessor; BCPR (branch coprocessor) behaves like a normal branch instruction except that it only branches whenever the CODONE register indicates that the coprocessor has yet to complete the loop execution.

The premise of our approach is to capitalize on the latency hiding approach[20] in the loop execution itself. The base processor would perform all memory operations (fetch / store) while the coprocessor computes the values obtained from those operations. The whole body of the loop is pipelined into different stages (see Figure 5). The load segment of the loop is repeated before the loop body to facilitate loop pipelining. While the base processor performs the fetching for the second iteration of the loop, the coprocesor could perform calculation for the data fetched during the first iteration. However, the result from the coprocessor should be ready before the end of the second iteration where the value will be used and stored back to memory. The values in the Instruction Decode (ID) and Writeback (WB) pipeline registers synchronizes the execution of the coprocessor with the base processor (i.e. informing when the values from memory are available).

The following simplifications are made to the design: We assume that there would be no preemption and interrupt request during loop operation. In a multitasking system, a restriction has to be imposed such that swapping would not occur during loop execution. Secondly, the coprocessor does not stall the CPU. This raises a question about what happens when data is not ready for the base processor to use. Such operation latency would already be known to the designer at creation time. If such a situation were to occur, NOP instructions would be inserted into the source code. This would simplify the circuitry as well as logic for such an implementation. On the other hand, additional signals could be inserted to handle such a situation.

4.2 Advantages & Limitations

1. The high-level synthesis approach provides a fast method for creation of the coprocessor.
2. Since we do not know the schedule of the HLS based coprocessor, data to the coprocessor has to be available before the coprocessor starts executing. This requires a large number of register ports. Additionally, we need a special wrapper to stall the coprocessor until the values are ready.
3. Addresses are not known by the automatic C-to-VHDL converter, as such addresses have to be computed in the base processor. This does not allow us to hide the latency between the address calculations of data required for the coprocessor, and memory accesses.

5. Custom Coprocessor Approach

To overcome the limitations of the HLS based coprocessor, we created a custom coprocessor in which we scheduled the operations carefully, such that they only wait for the dependent data to be ready. In addition, since only one piece of data can be read / written into the base processor at any one time, whenever such data is available, we schedule the coprocessor to fetch it from the register file, and store it within the coprocessor. This reduces the amount of interconnect between the processor and coprocessor.

5.1 Architecture

Similar to the HLS architecture, the SCPR and BCPR coprocessor instructions are added to the existing instruction set. The coprocessor connects to and shares the register file of the base processor.

One of the differences in comparison to the HLS approach is that the coprocessor developed now has only one read port and one write port connected to the GPR. The coprocessor would read the required values from the GPR and store it to intermediate registers within the coprocessor. Figure 3(b) shows the similarities with the HLS approach. However, register file size is now fixed to 5 read ports and 3 write ports.

Our integrated coprocessor approach (as opposed to the HLS approach) takes assembly code and converts to macroblocks. Figure 4 shows a code segment with the corresponding graphs, where *lh* and *lw* are load instructions and *sw* is a store instruction. A macroblock is detected when a group of interdependent instructions are 'sandwiched' between load instructions and one store instruction. These macroblocks are implemented as components within the coprocessor (manually converted to VHDL and synthesized). Instructions which calculate memory addresses can be grouped together as macroblocks and executed as a coprocessor component (as opposed to the HLS approach where such execution is only performed in the base processor). Thus, the memory operations occur in the base processor (such as *lh* and *lw*) while the macroblock is executed in the coprocessor. By loop pipelining, while data is fetched from iteration 2 (Figure 5), data from iteration 1 is processed by the coprocessor. The combined calculation of addresses in the coprocessor improves the overall performance of the application. A coprocessor consists of a number of macroblocks.

```
       lh $20,0($2)
       lw $22,304($7)
       bgez $20,skip
       subu $20,$0,$20
skip:  srav $20,$20,$8
       sll $2,$22,0x2
       addu $2,$2,$3
       sw $20,0($2)
```

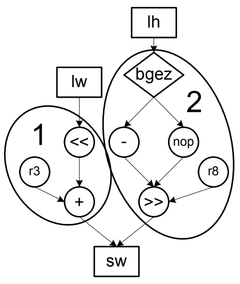


Figure 4: Example code segment & corresponding graph

The main premise of the integrated coprocessor approach is that memory operations can only be performed one after another in a single-pipeline processor. Thus it is redundant to read in all values of the registers at a single time. Reading from the register file one at a time and storing into internal registers within the coprocessor when the data is available would be sufficient, removing the need to increase the size of the register file.

Instruction SCPR takes the integrated coprocessor from Idle State to Ready State. To start a macroblock component within the coprocessor (say macroblock 2 in Figure 4), special signals are implemented which indicate the availability of the values from load instructions.

6. Discussion of the Architecture

The execution schedule in Figure 5 shows how memory operations and computation can be performed at the same time using this architectural approach. Figure 5 shows that the computation stage in iteration one (marked with '1's) can be hidden during the loading stage of iteration two (marked with '2's) and so on. Execution latency is hidden during memory operations (i.e. loads / stores) performed by the base processor. The graph shows the execution stages in a single-pipeline processor and in our current approach. Our approach is best used when there are lots of computation cycles which can be hidden via the loop pipelining technique [11, 23].

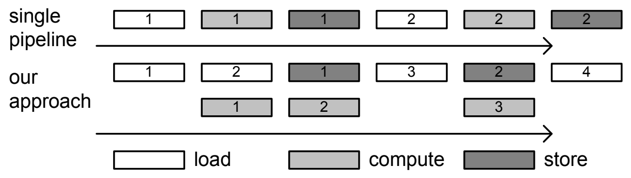


Figure 5: Parallel Execution

A high pipeline depth of any processor design has always been associated with high penalty costs for branch mispredictions, jumps and data dependencies on prior executed instructions [19]. Our coprocessor can read the value from a register only after the value is written back to the register file (i.e. after the writeback stage). If the coprocessor requires the value from the register file (which has been updated / written by a prior instruction), then the coprocessor has to wait six clock cycles from the time the "prior instruction" was decoded. Thus, this coprocessor architecture would not be feasible for short loops and if the loop pipelining technique imposed does not provide enough time to hide this latency. The custom coprocessor design approach is best used when there is a huge number of computations 'sandwiched' between memory operations.

If macroblock 2 in Figure 4 is converted to a coprocessor component, then the loaded value in instruction *lh* can only be read and processed by the component after the writeback stage. The component then writes back the value to the register file before decode stage of the store instruction *sw*.

The component created by the HLS framework can be seen as a function: accepting inputs during an iteration and outputs the results at the end of the iteration. As explained in Section 4.2, addresses of values needed by the HLS component need to be calculated by the main processor before the coprocessor requires them. These calculations are required due to the fact that it is not possible to break the loop functionality of the coprocessor using the HLS framework. Such tasks would be more complicated in situations where indirect memory accesses are required. Note that this is only a limitation when the core generated by the HLS framework is being used as a coprocessor in this architecture.

The number of register ports used by the integrated coprocessor approach remains constant (as opposed to the HLS approach - see Section 4.1) and would not affect the size of the base processor when the coprocessor increases in complexity. However, Figure 8 shows that the size of the integrated coprocessor is actually larger than the HLS-based coprocessor. This is due to the intermediate registers used in the integrated coprocessor architecture (see Section 5). However, when combined with the base processor, the integrated coprocessor approach achieves a smaller size compared to the HLS approach.

7. Experimental Setup & Tools

We used the SPARK [17] framework in our high-level synthesis approach. SPARK is a C-to-VHDL high-level synthesis framework that employs a set of compiler, parallelizing compiler, and synthesis transformations. The SPARK methodology is ideal for creating functional ASIC or FPGA blocks / modules from ANSI-C functions which can be used by ASIPs or general purpose microprocessors. Slight modifications are performed to the function code before the VHDL component can be generated.

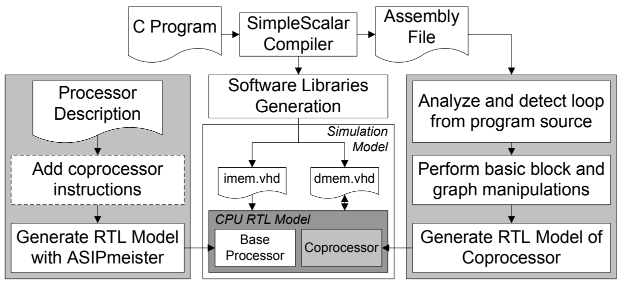

C Program
SimpleScalar Compiler
Assembly File
Processor Description
Add coprocessor instructions
Generate RTL Model with ASIPmeister
Software Libraries Generation
Simulation Model
imem.vhd
dmem.vhd
CPU RTL Model
Base Processor
Coprocessor
Analyze and detect loop from program source
Perform basic block and graph manipulations
Generate RTL Model of Coprocessor

Figure 6: Experimental Setup

The experimental setup consists of the CPU RTL Model (see Figure 6) which is generated by the ASIPmeister CPU generation tool. We used the CPU generation methodology proposed in [30] to provide a basic infrastructure and framework, making use of the existing SimpleScalar [10] toolset. The framework provides rapid generation of an ASIP given a set of CPU specifications. As explained in Section 4, two coprocessor instructions are added (i.e. SCPR and BCPR) to the existing PISA ISA in the generated CPU. The RTL Model is connected to instruction and data memory models. These VHDL memory models are generated by compiling C programs into SimpleScalar binaries and then translating them into VHDL code. The coprocessors (HLS and integrated coprocessor) are added and connected to the overall system after the CPU is generated. Software simulation is performed via ModelSim SE 6.0c using the Simulation Model represented in Figure 6.

The CPU RTL Model is synthesized to gate-level using Synopsys Design Compiler [3] W-2004.12 with TSMC 90nm (tcbn90g_110a) standard cell libraries. All registers with a minimum bank size of 4 bits are clock gated by Power Compiler. The synthesized VHDL files are then simulated in ModelSim SE 6.0c together with the simulation model of data memory and instruction memory. The switching activities obtained are used by Synopsys PrimePower 2003.12 for power calculations (see Figure 7).

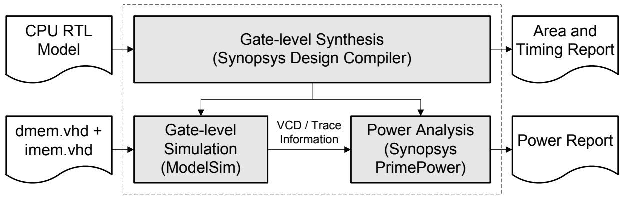

CPU RTL Model
Gate-level Synthesis (Synopsys Design Compiler)
Area and Timing Report
dmem.vhd + imem.vhd
Gate-level Simulation (ModelSim)
VCD / Trace Information
Power Analysis (Synopsys PrimePower)
Power Report

Figure 7: Synthesis and Power Calculation Flow

We also created customized instructions to compare against the coprocessor approaches. The customized instructions used in the extended processor were generated using the approaches developed in [12]. Their framework rapidly generates customized instructions with their corresponding customized components which are included in the pipeline of the base processor to accelerate the critical code segments. This approach is only applied to the selected loop segment of the Huffman encoder for comparison with our customized coprocessor approach and the HLS approach.

7.1 Verification

The SimpleScalar [10] simulator is modified to provide a framework for rapid prototyping of new extended and coprocessor instructions without implementing the VHDL model (synthesizable VHDL models for all three systems were created - each usually takes approximately 150 minutes for complete simulation of JPEG using ModelSim). The simulator reduces this time to a few seconds and verifies that the change in code does not adversely affect the functionality of the program. When the VHDL model is developed, the data memory dump from both VHDL and SimpleScalar simulations are compared using the *diff* unix command. Both memory dumps should be identical.

For verification of new (coprocessor/extended) designs, the memory dumps of the old and new designs cannot be compared as the program code would have been changed. A program (i.e. *hexdump* is developed to extract the output files from the memory dump. Both output files from the original and new designs are then compared.

8. Results

Table 2 shows the power and performance improvement of the four designs used in our case study. Area and power figures are measurements of the on-chip components only and do not include the external memory.

Column 1 shows the different processor design approaches used in our case study. The energy (column 2) in the loop segment is calculated using the equation below:

We define the energy in loop as :

$$E_{loop} = C_{loop} \times T_{clk} \times P_{loop} \qquad (2)$$

where clock period : $T_{clk} = 10ns$

Design Approaches	*Energy in Loop, E_{loop}*	*Loop Energy Savings*	*Loop, C_{loop} (cycles)*	*Loop Improvement*	*Program, C_{prog} (cycles)*	*Program Improvement*	*Idle Power Usage, P_{idle}*	*Loop Power Usage, P_{loop}*
Original Processor	176.903 μJ	NA	4,213,978	NA	21,317,212	NA	3.963 mW	4.198 mW
Extended Processor	142.776 μJ	19.29%	3,165,066	1.33x	20,265,496	5.19%	4.232 mW	4.511 mW
HLS + Processor	126.917 μJ	28.26%	2,667,332	1.58x	19,772,662	7.81%	4.133 mW	4.758 mW
Integrated + Processor	75.964 μJ	57.06%	1,640,340	2.57x	18,721,162	13.87%	4.125 mW	4.631 mW

Table 2: Power, Energy and Performance Table

All synthesized designs have a 10ns clock period. The total execution cycles (including all iterations) of our loop segment (identified in Section 3.1) is shown in column 4. The loop improvements are shown in column 5. Our approach accelerated program runtime by up to 13.87%, which is close to the theoretical maximum improvement of 18.46% shown in Table 1.

Column 6 gives the total execution cycles of the JPEG encoder benchmark program for encoding the image chosen in Section 3. The percentage speedup of the new designs compared to the original processor design is shown in column 7.

Column 8 and 9 in Table 2 respectively show the power used in the processor when no executions are performed (idle stage) and when executing the loop. When not in use, power consumption in our integrated coprocessor is 109.76 μW, whereas the HLS coprocessor utilizes 53.758 μW (not shown in Table 2). The graph in Figure 8 shows that the total energy used in the loop execution of the integrated version is halved compared to the execution in the original processor.

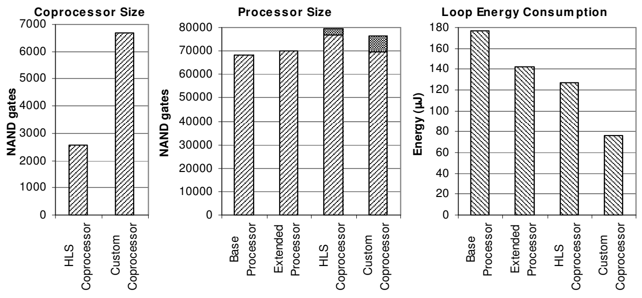


Figure 8: Area and Loop Energy Usage

Figure 8 shows the area usage of the synthesized processor designs and the individual coprocessors. The total processor size includes the size of the coprocessors as well, which is shaded in black. Although the integrated coprocessor is more than twice the size of the HLS coprocessor, the savings in the size of the register file offsets this and results in a smaller and efficient processor design compared to the HLS approach.

9. Conclusions

We have performed an interesting case study by exploring a novel and tightly coupled architecture to accelerate a computationally intensive loop in a JPEG encoder. Loop pipelining and latency hiding is used to achieve near maximum speedup and parallelism between the base processor and the coprocessor. We also found that the coprocessor approaches achieve much better speedup and lower energy consumption compared to the customized instruction approach. Additionally, using our integrated coprocessor approach, we notice that more computations can be offloaded from the base processor to the coprocessor compared to the high-level synthesis approach to achieve better performance.

10. References

[1] Altera Nios Processor. Altera Corp. (http://www.altera.com).

[2] ARCtangent. ARC International (http://www.arc.com).

[3] Design Compiler. Synopsys, Inc. (http://www.synopsys.com).

[4] Independent JPEG Group. IJG (http://www.ijg.org).

[5] Jazz DSP. Improv Inc. (http://www.improvsys.com).

[6] JPEG Encoder Core. Alma Technologies (http://www.alma-tech.com).

[7] SP-5flex. 3DSP Corp. (http://www.3dsp.com).

[8] Xtensa Processor. Tensilica Inc. (http://www.tensilica.com).

[9] *Intel XScale Core : Developer's Manual*. Intel Corporation, 2000.

[10] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[11] K. S. Chatha and R. Vemuri. A Tool for Partitioning and Pipelined Scheduling of Hardware-Software Systems. In *ISSS '98*, pages 145 – 151, Hsinchu, 1998.

[12] N. Cheung, S. Parameswaran, and J. Henkel. INSIDE: INstruction Selection/Identification & Design Exploration for Extensible Processors. In *ICCAD 2003*, pages 291–297, 2003.

[13] CriticalBlue. Coprocessor Synthesis – Increassing System on Chip Platform ROI. Technical report, CriticalBlue, June 2004.

[14] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. In *IEEE Design & Test*, volume 10, pages 64 – 75, 1993.

[15] T. Glökler and H. Meyr. Power Reduction for ASIPS: A Case Study. In *IEEE Workshop on Signal Processing Systems*, pages 235–246, Antwerp, Belgium, 2001.

[16] R. K. Gupta and G. De Micheli. Specification and Analysis of Timing Constraints for Embedded Systems. *IEEE Trans. of Computer-Aided Design of Integrated Circuits and Systems*, 16(3):241 – 256, 1997.

[17] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *VLSID 2003*, 2003.

[18] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Trans. on Very Large Scale Integration Systems*, 12(2):206 – 217, 2004.

[19] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[20] S. Hiroyuki and Y. Teruhiko. Characteristics of Loop Unrolling Effect: Software Pipelining and Memory Latency Hiding. In *IWIA 2001*, pages 63 – 72, Maui, HI USA, 2001.

[21] J. K. Hunter, J. V. McCanny, A. Simpson, Y. Hu, and J. G. Doherty. JPEG Encoder System-On-a-Chip Demonstrator. In *Asilomar Conf. on Signals, Systems, and Computers*, volume 1, pages 762 – 766, 1999.

[22] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: An ASIP Design Environment. In *ICCD 2000*, pages 430–436, Austin, TX, USA, 2000.

[23] J. Jeon and K. Choi. Loop Pipelining in Hardware-Software Partitioning. In *ASP-DAC '98*, pages 361 – 366, Yokohama, Japan, 1998. February 10 - 13.

[24] A. Langi and W. Kinsner. An Architectural Design of a Wavelet Coprocessor. In *CCECE 1994*, volume 2, pages 497 – 500, Halifax, NS, 1994.

[25] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *IEEE/ACM Int. Symp. on Microarchitecture*, pages 330 – 335, 1997.

[26] H.-Y. Lin, T.-J. Lin, C.-M. Chao, Y.-C. Liao, C.-W. Liu, and C.-W. Jen. Static floating-point unit with implicit exponent tracking for embedded DSP. In *ISCAS 2004*, volume 2, pages 821 – 824, 2004.

[27] T.-J. Lin and C.-W. Jen. CASCADE - configurable and scalable DSP environment. In *ISCAS 2002*, volume 4, pages 870 – 873, 2002.

[28] E. Maas, D. Herrmann, R. Ernst, P. Rüffer, S. Hasenzahl, and M. Seitz. A Processor-coprocessor Architecture for High End Video Applications. In *ICASSP 1997*, volume 1, pages 595 – 598, Munich, 1997.

[29] S. Parameswaran, M. F. Parkinson, and P. Bartlett. Profiling in the ASP codesign environment. *Journal of Systems Architecture*, 46(14):1263 – 1274, 2000.

[30] J. M. D. Peddersen, S. L. Shee, A. Janapsatya, and S. Parameswaran. Rapid Embedded Hardware/Software System Generation. In *VLSID 2005*, pages 111 – 116, 2005.

[31] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *MICRO-27*, pages 172 – 180, 1994.

[32] S. L. Shee. *VLSI Chip Implementation for Communication Protocols : JSCHIP Project*. Undergraduate thesis, The University of New South Wales, 2003.

[33] G. Stitt, R. Lysecky, and F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. In *DAC 2003*, pages 250 – 255, 2003.

[34] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):216–228, 2004.

[35] J. Tubella and A. González. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *HPCA 1998*, pages 14 – 23, 1998.

[36] R. D. Wittig and P. Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *FCCM 1996*, pages 126 – 135, Napa Valley, CA, 1996.

[37] B.-F. Wu and C.-F. Lin. An Efficient Architecture for JPEG2000 Coprocessor. *IEEE Trans. on Consumer Electronics*, 50(4):1183 – 1189, 2004.