Customization of Application Specific Heterogeneous Multi-Pipeline Processors

ABSTRACT
In this paper we propose Application Specific Instruction Set Processors with heterogeneous multiple pipelines to efficiently exploit the available parallelism at instruction level. We have developed a design system based on the Thumb processor architecture. Given an application specified in C language, the design system can generate a processor with a number of pipelines specifically suitable to the application, and the parallel code associated with the processor. Each pipeline in such a processor is customized, and implements its own special instruction set so that the instructions can be executed in parallel with low hardware overhead. Our simulations and experiments with a group of benchmarks, largely from Mibench suite, show that on average, 77% performance improvement can be achieved compared to a single pipeline ASIP, with the overheads of 49% on area, 51% on leakage power, 17% on switching activity, and 69% on code size.

1. INTRODUCTION
Increasingly pervasive, ubiquitous and large embedded systems demand designs which are high in performance while low in cost. Embedded systems differ from general purpose computing systems since such processors only execute a single application or a class of applications. Application Specific Instruction Set Processors (ASIPs) in particular, are suited for utilization in embedded systems where customization allows increased performance, yet reduces area cost and power consumption by not having unnecessary functional units.

Research and development on ASIPs has mainly focused on instruction set generation for processors with simple pipeline architectures. Systematic parallelism exploration in ASIP design, is still in its early stage.

Parallelism exploitation often comes with an area overhead due to the need to replicate resources. In this work we aim to explore the possibility of parallelizing applications with little replication, reducing area overhead as much as possible.

For a given application, it is possible to design an ASIP with customized multiple pipelines. Since the application is well understood, the number of pipelines and each of the individual pipes can be customized. We call this "customized VLIW ASIP" since its parallel processing scheme is similar to a VLIW processor (though some different in architecture) but it is strongly application oriented. The number of pipelines is determined specifically for the application and the functional units on each pipeline are based purely on the application itself.

1.1 Related Work
Research and development in the area of ASIPs has been flourishing for a couple of decades. Numerous tool suites have been developed [2, 14, 19].

To generate an ASIP, one needs first to create an instruction set specifically tailored to a given application. Given an application, there are a large number of design alternatives. Research on automating design space exploration and instruction set generation has been very active [18] [20] [4] [12] [7].

Apart from specific instruction set generation, customization of processor architectural features such as register file and function units, has been studied [11] [5] [3] for performance enhancement.

To further improve performance, researchers have considered parallel processing approaches. In [9], the authors presented a Very Large Instruction Word (VLIW) ASIP with distributed register structure. Jacome et al. in [10] proposed a design space exploration method for VLIW ASIP datapaths. In [13], Kathail et al. proposed a design flow named PICO (Program In Chip Out) for a specific SoC (System-on-Chip) design, where parallelism exploration is tackled at different design levels including at the instruction level with VLIWs. An example of optimization of VLIW architectures to a typical image processing application is presented in [6]. Sun et al. in [17] proposed a design for customized multi-processors. Recently Tensilica has developed a VLIW-like technology, with FLIX instructions [1], which allows flexible-length instruction extensions, with each instruction being similar to a VLIW instruction.

In [16], the author discussed a decoupled Access/Execute architecture, with two computation units each of which contained its own instruction streams. Using a similar architecture, in [15], the authors presented a design approach for dual-pipeline customized processor. In this paper, we expand the above work to a multiple pipeline structure, which is customizable to a given application. We enhance the pipeline structure presented in [15] by utilizing a forwarding scheme so that the data hazards in the pipelines can be reduced. Also, unlike in [15], where a single clock cycle penalty for memory accesses is used, we consider different wait cycles for memory access so that the effect of memory access penalty on performance can be observed. Moreover, instead of using small and non-standard benchmarks as in [15], we target the applications from Mibench benchmark suites (popular in embedded system design) in our study. Our approach is also somewhat similar to FLIX [1] and the configurable VLIW in PICO [13]. However, FLIX is limited by the instruction length. Its maximum length is 64 bits. Though more parallel operations can be squeezed into one instruction, the limited encode-bits restrict the parallelism exploitation (due to few available operation types for each parallel execution stream) and possibly reduce the opportunity of computing resource sharing (resulting in high chip area cost). Such a limitation, however, does not apply to our design approach. For the VLIW in PICO, the communication between parallel components is done through memory; while in our design, communication is performed through fast forwarding logics or the register file itself. In comparison to FLIX and PICO, our design uses dedicated control circuits for each pipeline, which reduces the complexity of the critical path, hence the critical path delay. Moreover, we present a different systematic design flow that allows a high degree of customization, from functional units, to individual pipelines, and to the number of pipes.

1.2 Contributions
We propose the design of ASIPs with varying number of pipelines. With such a design strategy, parallelism can be efficiently exploited. In particular, we introduce a novel architecture which tightly couples
multiple pipelines via the register file; propose a method to customize
the number of pipelines and instruction sets for each of the pipelines;
and develop an implementation system with such a design.

To show the efficiency and viability of our approach, we study perfor-
mance, area, code size, and energy consumption of processors cre-
ated by our design approach for several well known benchmarks.

1.3 Paper Organization

The rest of the paper is organized as follows: section 2 describes
the architecture template of the multi-pipeline processor to be imple-
mented; while section 3 describes the methodology taken to design
such a processor. Experimental setup and results are given in section
4; and the paper is concluded in section 5.

2. ARCHITECTURE

Our design approach is based upon the Thumb processor instruc-
tion set architecture (Thumb ISA), which is simple and small. Fig-
ure 1 illustrates the general architecture of our ASIP design. It con-
ists of at least two pipelines, Pipe 1 and Pipe 2, which are necessary
for primary functions of all applications. Pipe 1 is specifically design-
ated for program flow control. This pipeline is primarily responsible
for fetching instructions from the instruction memory and dispatch-
ning them to all other pipelines. When the program branches, Pipe 1
flushes all pipelines. Pipe 2 performs data memory access, transfer-
ing data between the register file and data memory. Pipe 1 contains
(at least) an ALU, while Pipe 2 contains (at least) a data memory
access unit (DMAU). This structure can be augmented when the in-
struction sets for the pipelines are enlarged.

Figure 1: Architecture Template

Extra pipelines are utilized based on the parallelism exhibited in
the application. All pipelines share one register file which is multi-
ported, so that all pipelines can access the register file simultaneously.
Each pipeline has a separate control unit that controls the operation
of the related functional unit on that pipe. Forwarding is enabled in all
pipelines so that the results from the execution unit can be forwarded
within a pipeline and between pipelines.

3. METHODOLOGY

In this section, we first give an overview of our design methodol-
ogy and then present the algorithms used in the design.

3.1 Approach Overview

The design flow described in this paper is illustrated in Figure 2. It
takes as input an application written in C. The program is first com-
piled into single-pipeline assembly code based on the Thumb ISA
(step 1).

In the next step (step 2), an initial pipeline number is chosen as the
starting search point of the design space exploration. We start from
the minimal 2-pipe structure and the number of pipelines is iteratively
increased as the exploration continues.

Figure 2: Design Flow

We use the number of cycles required for a memory access as an
input to the scheduling step. If more cycles are required to access
memory, then greater number of instructions can be scheduled in par-
allel with the memory access instruction.

The output of the scheduling step is illustrated in Figure 3(a) (More
details on the algorithm for step 3 are given in section 3.2).

The original one-pipe program is divided into several sequences
(shown in columns in Figure 3(a)). Instructions that are scheduled
in the same time slot (shown on the same row in the figure) are ex-
cuted simultaneously on different pipelines. Each of the sequences
forms an instruction set for the corresponding pipeline, as illustrated
in Figure 3(b), where instruction set, ISA i, is obtained from program
sequence i (Seq. i in the figure).

The parallel code is generated based on the object code of each of
the program sequences, which is obtained from the assembly output
of the GCC compiler.

In step 4 we use ASIPMeister, a single-pipe ASIP design soft-
ware tool, to create a design for each pipeline. The tool takes as input
the instruction set, functional unit specification, and instruction mi-
crocode, and produces a VHDL simulation model and a VHDL syn-
thesis model. All pipelines are then integrated into a multi-pipeline
processor with a parallel structure, as shown in Figure 1.

In the last step, the multi-pipe processor model is simulated us-
ing ModelSim for functional validation, and is then synthesized with
Synopsys Design Compiler. The simulation and synthesis step pro-
vides the performance, area and power consumption of the design,
which are used in the evaluation of the design. The iterative pro-
cess, formed by steps 3 to 5, is repeated for designs with increased
pipelines until no further improvement can be obtained.

The approach used for processor/code generation and evaluation
is summarized in Figure 4. Note in order to obtain accurate switch-
ing activity, a second simulation with the gate-level VHDL model
produced by Synopsys Compiler, is performed (as indicated by the
dashed line in the figure).

3.2 Exploitation of Instruction Parallelism

Our parallel scheduling is performed within instruction basic blocks.
For a basic block, if one of its instructions is executed, then all in-
structions in the block are executed. Therefore, parallelism within
blocks is static and can be easily extracted at the initial stage of
the design. Another advantage of scheduling within basic blocks is
the avoiding of the complicated speculation/prediction issue which
would otherwise need to be addressed.
Integrator
GCC
ModelSim

ing processor area and power cost is small while the execution time
are based upon the following assumptions and considerations.

Scheduling Techniques

Figure 3: Parallel Program Sequences and Instruction Sets

(a) Parallel Program Sequences

<table>
<thead>
<tr>
<th>Seq 1</th>
<th>Seq 2</th>
<th>Seq 3</th>
</tr>
</thead>
</table>
| mov r2, s1
and r1, r1, r3
bc:
mov r2, #0
cmp r0, #0
bne .L20: |
| push (1r)
ldr r9, [r2]
bc:
cmp r0, #0
bne .L20: |
| mov r1, #15 |

(b) Individual Pipeline Instruction Sets

<table>
<thead>
<tr>
<th>ISA 1</th>
<th>ISA 2</th>
<th>ISA 3</th>
</tr>
</thead>
</table>
| mov rn, rm
and rn, rm, rm
cmp rn, immed
bne label |
| push (rn)
ldr rm, [rn, immed]
add rn, rm, immed
cmp rn, immed |
| mov rn, rm
mov rn, immed
add rn, immed |

Figure 4: Processor/Code Generation and Testing System

branch instructions, the potential workload of the pipe would be
0.7.

- For a single pipeline, a memory load instruction will cause a
pipeline stall. The pipeline is idle until the memory operation
is complete. Instead of stalling all pipelines in the processor
during a load operation in one pipeline, we allow instructions
to be scheduled in other pipes such that the effect of the hazard
is limited.

We use design area efficiency, \( \eta \), to evaluate the design quality. The
efficiency is defined as the maximum possible execution fre-
quency of an application per area unit and is given by the following
formula,

\[
\eta = \frac{1}{(T \times A)},
\]

where \( T \) is the execution time of the application and \( A \) the area of
the processor that executes the application. Large \( \eta \) means high performance
with small area cost.

Basic Block Scheduling Algorithm

Our scheduling method is presented in a bottom-up manner, where
the pipeline selection for an instruction (Algorithm 1) is presented
first, followed by the basic block instruction scheduling procedure
(Algorithm 2).

In Algorithm 1, we find a suitable pipeline for an instruction. Three
parameters: scheduling overhead, pipeline area cost, and pipeline
work load, are used here to guide the pipeline allocation for instruc-
tions. As can be seen from the algorithm, scheduling overhead takes
the highest priority among the three parameters, with the workload
coming the second and area cost the last.

Algorithm 1 Instruction pipeline selection: PipeSelection(i, P):

//find a pipe for instruction, i, from a set of pipes, P.

step 1: find the pipe where the scheduling overhead is minimal if
the instruction is placed in that pipe;

step 2: if more than one pipe is found in the previous step, find the
pipe with minimal load (i.e., one with fewer instructions);

step 3: If more than one pipe is found in the previous step, find the
pipe with the smallest area;

step 4: return the found pipe;

With the above pipeline selection algorithm, the basic block schedul-
ing method is given in Algorithm 2. The algorithm takes a basic block
and schedules its instructions to a set of pipes, \( P \). We use an array,
Algorithm 2 Basic Block Scheduling: blockScheduling(B, \mathcal{P})

// Initialize array, Sched, with Nop instructions
Initialize(Sched);

// schedule instructions in Block, B, to a set of pipes, \mathcal{P}.
for all \ i \in B \ (in \ program \ sequence) \ do
    scheduling_done(i) = FALSE;

    while scheduling_done(i) is FALSE do
        find the earliest time slot, t, for instruction i;
        find all available pipes, \mathcal{P} \_available, at t;
        find a suitable pipe, p, for instruction i using Algo. 1;
        if p exists then
            Sched[i][p] = i;
            scheduling_done(i) = TRUE;
        else
            try scheduling the instruction to the next time slot
            t++;
        end if
    end while
end for

We demonstrate the scheduling algorithm with a basic block as shown in Figure 5(a). The block contains nine instructions. Assume the instructions are to be scheduled into 3 pipes: Pipe 1, Pipe 2 and Pipe 3. Pipe 1 contains an ALU and can perform all but memory access instructions. The memory access instructions are exclusively performed by Pipe 2. The functions in Pipe 3 are initially undefined but can be any functions (except memory and branch type), as requested by the scheduling algorithm.

The scheduling starts with the first instruction, push, since it is not dependent on any other instruction, its earliest scheduling time slot is 1 (i.e. t=1). At this moment, all three pipes are available. Amongst them, Pipe 2 is the most suitable pipe (the only pipe with 0 scheduling overhead). Next instruction is mov, and it can be scheduled in the first time slot. Among the two available pipes: Pipe 1 and Pipe 3, Pipe 1 is selected (since Pipe 1’s scheduling overhead is 0). The following non-dependent instruction isl is then scheduled to the last pipe: Pipe 3, with the scheduling overhead of a Shifter. The next instruction and is dependent on the second instruction, and therefore its earliest scheduling time slot is 2. With zero scheduling overhead to Pipe 1, it is assigned to the second time slot in Pipe 1. The remaining two pipes are available to the following add instruction. Both pipes do not have functional unit for add instruction. The related scheduling overheads are therefore same. Since the potential load for pipe 2 (7/9) is greater than pipe 3 (4/9), pipe 3 is selected for the add instruction, which leaves pipe 2 for the next instruction ldr to be scheduled in the time slot. This process is repeated for the rest of instructions and the scheduling result is shown in Figure 5(b).

Algorithm 3 Multi-pipe Processor Design:

// best design and its related design efficiency are initialized.
best_design = NULL;
\eta_{best} = 0;

// the design iteration starts from 2 pipe structure.
N_p = 2;
design_done = FALSE;

while design_done is FALSE do
    for all B \in G do
        blockScheduling(B, N_p);
    end for

    processor_generation(\gamma);
    processor_simulation(\epsilon);
    \eta = calculate_design_efficiency(\sigma);
    // if design is improved
    if \eta > \eta_{best} then
        best_design = current_design;
        \eta_{best} = \eta;
        N_p++;
    else
        design_done = TRUE;
    end if
end while

output best_design;

Design Algorithm

Based on the above basic block scheduling algorithm, an application program can be scheduled into multiple pipes. The overall design is summarized in Algorithm 3. Given the basic block graph, G, for an application, the algorithm gives an efficient design with a suitable number of pipelines and special instruction sets for each of the pipelines. Each design iteration is evaluated with performance/area ratio, \eta. The design loop stops when new design cannot bring further improvement.

4. SIMULATIONS AND RESULTS

With the above methodology we designed multi-pipeline processors for a set of applications mainly from MiBench [8]. These benchmarks represent a variety of application fields such as network, security, telecommunication and automotive, which are frequently encountered in embedded systems.

Figure 6: Clock Period (ns)

As described in section 3, our base instruction set architecture was based on the Arm-Thumb processor. We generated VHDL models and the associated executable code for the multi-pipeline processor for each of the applications, with memory access latencies. The designs were then synthesized using Synopsys Design Compiler based on the TSMC 90nm core library, and simulated with the Modelsim simulator.

Performance is evaluated in the processor clock speed, which is given by Design Compiler, and the clock cycles given by Modelsim. Power consumption is divided into dynamic power and leakage.
Table 1: Performance Improvements and Overheads

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>adpcm.decode</th>
<th>adpcm.encode</th>
<th>bas.math</th>
<th>bitcount</th>
<th>crc32</th>
<th>q.sort</th>
<th>sha.encr.</th>
<th>strs.search</th>
<th>endian</th>
<th>b.search</th>
<th>cstr.</th>
<th>dijkstra</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance 2 Pipe</td>
<td>200%</td>
<td>250%</td>
<td>225%</td>
<td>187.5%</td>
<td>225%</td>
<td>187.5%</td>
<td>225%</td>
<td>225%</td>
<td>225%</td>
<td>225%</td>
<td>225%</td>
<td>225%</td>
<td>225%</td>
</tr>
<tr>
<td>Area % 2 Pipe</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
</tr>
<tr>
<td>Leak. Pow. % 2 Pipe</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
</tr>
<tr>
<td>Switch. Activ. % 2 Pipe</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
</tr>
<tr>
<td>Code Size 2 Pipe</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
</tr>
<tr>
<td>3 Pipe</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
</tr>
</tbody>
</table>

Figure 7: Area (cells)

Figure 8: Performance/Area (1/(second \times million cells))

high execution capacity per million cells for all of the tested applications. This is because the average instruction parallelism for the basic blocks is below 3. Unrolling loops would have improved parallelism, but unrolling of loops was not considered in the experiments.

It is worth noting that leakage power closely follows the area cost as can be seen from Figures 7 and 9, where both figures show a near identical trend. However, this similarity is not obvious for switching activity in Figure 10. It also can be seen from Figure 11 that the code size increases as the number of pipelines increases because it is unlikely that all pipelines can be fully utilized during application execution.

Since memory is typically slower than the processor, we examined the effect of slower memory on performance in terms of both clock cycles and execution time for the 2-pipe case, along with the 1-pipe designs for comparison. The clock cycles and execution time for each of the applications (columns 4–15) under different memory access latencies (ranging from 1 clock cycle to 3 clock cycles) for 1-pipe and 2-pipe processors are tabulated in Table 2, where the rows with the metrics labeled as CC('000s) give the clock cycles (in thousand) taken by each of the application programs, while rows with the label Exec. Time (ms) provide the execution times. The performance improvement of the 2-pipe designs over 1-pipe designs with different memory access latencies are also given in the table and it is graphically represented in Figure 12. As can be seen from Figure 12, there is little difference in performance improvement between different memory access latency schemes. Longer memory latency rarely

power. The leakage power is estimated by the Synopsys Design Compiler; and the dynamic power is represented by the switching activity of the gate level model generated by Synopsys Designer Compiler, for a sample set of data, and is obtained from Modelsim.

The simulation results for clock period, area, design efficiency, leakage power, switching activity, and code size are given in Figures 6, 7, 8, 9, 10 and 11, respectively. The percentages of performance improvement of the multiple-pipe processors over the single-pipe processors, and the related overheads in area, power and code size are summarized in Table 1. Note for application crc32 and binsearch (under the name b.search in the table), there is no 3-pipe design due to their low parallelism and intensive memory access nature.

As can be seen from Figure 6, 1-pipe processors demonstrate higher clock period than multiple-pipe processors. It is because the 1-pipe processors have a large control unit that needs to control the execution of all instructions in the instruction set while in multiple-pipe processors, the control unit for each pipeline only implements a small subset of instructions, resulting in shorter critical paths.

When the design changes from 1-pipe to 2-pipe, substantial performance improvements can be obtained. In contrast, there is little or no performance gain when going from 2-pipe designs to 3-pipe designs, but the design area overheads become significant, as illustrated in Figure 8, where 2-pipe processors give the best designs, with
affects the performance improvement. This is due to the efficiency of our parallel scheduling algorithm.

5. CONCLUSIONS

We presented an approach to customize a multiple pipe processor. The approach relates application instruction level parallelism with the multiple pipeline architecture. An effective parallel instruction scheduling algorithm is used to determine the number of pipelines and the instruction sets to be implemented by each of the pipelines such that the high performance improvement can be achieved with small area overhead. The performance improvement is achieved by specific instruction sets, improved pipeline (with forwarding logics) structure, and parallel instructions executing on the multiple pipelines. The small area overhead is retained by utilizing a distributed controller, minimized instruction set overlap between pipelines, and appropriate number of pipelines.

Our designs for a given set of benchmarks show that on average 77% performance improvement can be achieved with some overheads: 49% on area; 51% on power; 17% on switching activity; and 69% on code size. The parallel scheduling algorithm proposed in this paper can also efficiently utilize the memory access latency so that the effect of slow memory on the overall execution performance is reduced, with average standard deviation below 6% in our simulation experiments.

6. REFERENCES