Customization of Application Specific Heterogeneous Multi-Pipeline Processors

Swarnalatha Radhakrishnan, Hui Guo, Sri Parameswaran
School of Computer Science & Engineering
University of New South Wales
Sydney, Australia
{*swarnar, huig, sridevan*}*@cse.unsw.edu.au*

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 functional 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 performance, area, code size, and energy consumption of processors created 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 implemented; 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 instruction set architecture (Thumb ISA), which is simple and small. Figure 1 illustrates the general architecture of our ASIP design. It consists of at least two pipelines, Pipe 1 and Pipe 2, which are necessary for primary functions of all applications. Pipe 1 is specifically designated for program flow control. This pipeline is primarily responsible for fetching instructions from the instruction memory and dispatching them to all other pipelines. When the program branches, Pipe 1 flushes all pipelines. Pipe 2 performs data memory access, transferring 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 instruction sets for the pipelines are enlarged.

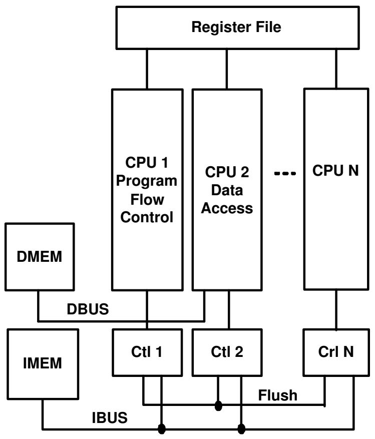

Figure 1: Architecture Template

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 methodology 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 compiled 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.

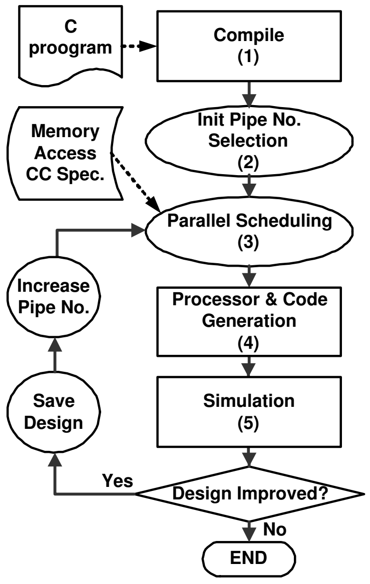

Figure 2: Design Flow

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 parallel 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 executed 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 software tool, to create a design for each pipeline.The tool takes as input the instruction set, functional unit specification, and instruction microcode, and produces a VHDL simulation model and a VHDL synthesis 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 using Modelsim for functional validation, and is then synthesized with Synopsys Design Compiler. The simulation and synthesis step provides the performance, area and power consumption of the design, which are used in the design evaluation. The iterative process, 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 switching 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 instructions 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.

Seq 1	Seq. 2	Seq. 3
.L68:	.L68:	.L68:
mov r2,sl	push {lr}	mov r1,#15
and r1,r1,r3	ldr r9,[r2]	0
bc:	bc:	bc:
mov r2,#0	0	0
cmp r0,#0	str r2,[r5,r9]	0
beq .L20	0	0
.L24:	.L24:	.L24:
sub r1,r0,#1	add r2,r2,#1	0
and r0,r0,r1	0	0
cmp r0,#0	0	0
bne .L24	0	0
.L20:	.L20:	.L20:
mov r2,sl	mov r5,r3	add r6,r6,#1
add r5,r5,#4	ldr r7,[r2,#96]	0
.L21:	.L21:	.L21:
sub r4,r4,#1	ldr r3,[r7,r2]	0
cmp r4,#0	str r3,[r2]	sub r2,r2,#4
bne .L21	0	0
.L54:	.L54:	.L54:
mov r0,r2	str r6, [sp, #12]	mov r1,r8
lsl r2,r0,#2	pop {pc}	0

(a) Parallel Program Sequences

ISA 1	ISA 2	ISA 3
mov rn,rm	push {rn}	mov rn,rm
and rn,rn,rm	ldr rm,[rn,immed]	mov rn,immed
mov rn,immed	mov rn,rm	add rn,rn,immed
cmp rn,immed	add rn,rn,immed	sub rn,rn,immed
beq label	str rd,[rn,rm]	
sub rn,rm,immed	ldr rd,[rn,rm]	
add rn,rn,immed	str rm,[sp,immed]	
sub rn,rn,immed	pop {rn}	
lsl rn,rm,immed	str rd,[rn,immed]	
bne		

(b) Individual Pipeline Instruction Sets

Figure 3: Parallel Program Sequences and Instruction Sets

Scheduling Techniques

Given the number of pipelines, our scheduling determines the time slot and the executing pipeline for all instructions such that the resulting processor area and power cost is small while the execution time for the block of instructions is minimal. Our scheduling algorithms are based upon the following assumptions and considerations.

- The functionality of a pipeline is determined by the instruction set, which is formed during the instruction scheduling phase. Scheduling an instruction to a pipeline may incur area overhead. If the pipeline already has the functional components used by the instruction, then the overhead is nil; otherwise, the overhead is the area of the extra functional component required for the instruction. For example, given a pipeline with an ALU function, scheduling an *add* instruction to this pipeline does not incur any additional area. Therefore, the overhead of the instruction is 0. If, however, this instruction is scheduled into a pipeline which only consists of a shift operation, then scheduling the *add* instruction will require at least an adder, and the area of the added functional unit is the incurred overhead. We call such an overhead as **scheduling overhead**.

- If an instruction can be executed by one of two pipelines, one that is complex and the other simple, executing the instruction in the complex pipe is assumed to be more costly as more logic gates are exercised, thus consuming more power. Since the complexity is closely related to the area, we use **area cost** to present the complexity of the pipeline. As such, one of our scheduling strategies is to schedule an instruction to a pipeline with a cheap area cost.

- Scheduling instructions to a pipeline of a lower workload is preferred, so that more instructions can be scheduled at the earliest time slot as possible. We define the **potential workload** of a pipeline as the percentage of instructions in the basic block which can be executed by the pipe. For example, in a pipe, assume we are able to execute any type of instruction other than memory and branch type instructions. If we have 10 instructions in the basic block, and three of them are memory and 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.

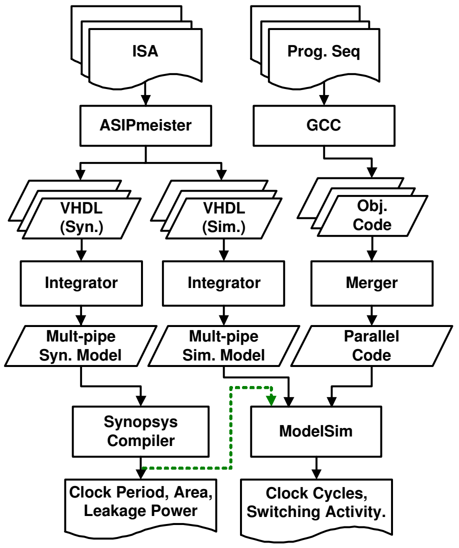

Figure 4: Processor/Code Generation and Testing System

We use **design area efficiency**, η, to evaluate the design quality. The efficiency is defined as the maximum possible execution frequency of an application per area unit and is given by the following formula,

$$\eta = 1/(T \times A),$$

where T is the execution time of the application and A the area of the processor that executes the application. Large η 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 instructions. 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)*

```
$//find\ a\ pipe\ for\ instruction,\ i,\ from\ a\ set\ of\ pipes,\ \mathbb{P}.$
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 scheduling 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, *Sched[timeslot][pipeline]*, to represent the scheduling result.

Algorithm 2 Basic Block Scheduling: *blockScheduling*$(B, \mathbb{P})$

```
//Initialize array, Sched, with nop instructions
Initialize(Sched);
//schedule instructions in Block, B, to a set of pipes, ℙ.
for all i ∈ 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, $\mathbb{P}_{avail}$, at t;
        find a suitable pipe, p, for instruction i using Algo. 1;
        if p exists then
            Sched[t][p] = i;
            scheduling_done(i) = TRUE;
        else
            //try scheduling the instruction to the next time slot
            t++;
        end if
    end while
end for
```

```
ntbl_bitcnt:
1:   push{lr}
2:   mov  r1, #15
3:   lsl  r2, r2, #2
4:   and  r1, r1, r0
5:   add  r0, r2, #4
6:   ldr  r3, .L42
7:   add  r4, r2, r3
8:   cmp  r0, #0
9:   beq  .L30
```

(a)

```
     |      Pipe 1       |      Pipe 2       |      Pipe 3
     |                   |                   |
x1:  | mov r1, #15       | push{lr}          | lsl  r2, r2, #2
x2:  | and r1, r1, r0    | ldr  r3, .L42     | add  r0, r2, #4
x3:  | cmp r0, #0        | nop               | nop
x4:  | beq .L30          | nop               | add  r4, r2, r3
```

(b)

Figure 5: Basic Block Scheduling

The notation, $Sched[i][j] = k$, means instruction k is scheduled to time slot i on pipe j.

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 types), 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 *lsl* 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 the instructions and the scheduling result is shown in Figure 5(b).

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, η. The design loop stops when new design cannot bring further improvement.

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

4. SIMULATIONS AND RESULTS

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

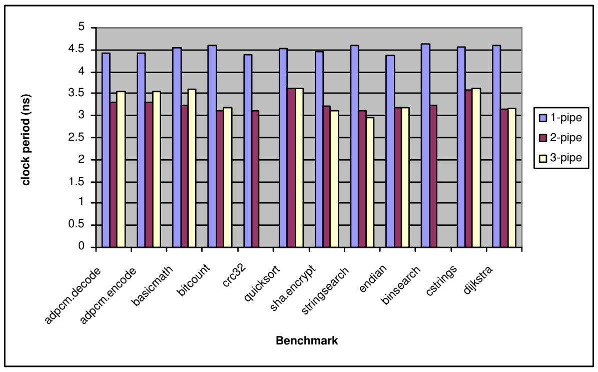

Figure 6: Clock Period

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 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 Com-

Table 1: Performance Improvements and Overheads

	Benchmarks	adp.dec	adp.enc	bas.math	b.count	crc32	q.sort	sha.encr.	str.search	endian	b.search	cstr.	dijkstra	Average
Performance	2 Pipe	69.1	60.5	90.9	97.0	79.2	63.8	84.5	97.1	78.4	69.5	50.5	84.8	77.1
%	3 Pipe	60.0	49.9	74.1	93.6	0.0	69.0	89.2	107.9	85.2	N/A	52.6	83.6	63.7
Area	2 Pipe	37.8	35.2	56.6	69.9	38.0	62.2	45.2	56.4	59.2	52.4	20.0	58.3	49.3
%	3 Pipe	140.5	134.0	132.5	126.8	N/A	117.4	128.0	102.1	114.9	N/A	101.1	118.2	101.3
Leak. Pow.	2 Pipe	39.1	39.2	61.5	65.5	40.7	61.3	46.6	57.8	61.2	53.2	28.6	64.0	51.5
%	3 Pipe	145.7	145.3	147.4	127.3	N/A	124.4	139.2	109.6	44.9	N/A	105.1	130.2	101.6
Switch. Activ.	2 Pipe	23.6	7.1	8.7	30.5	1.8	37.0	27.7	17.5	13.2	10.5	24.3	0.9	16.9
%	3 Pipe	41.5	15.5	11.7	39.7	N/A	47.0	46.4	23.0	13.8	N/A	49.2	5.4	24.4
Code Size	2 Pipe	66.4	60.6	59.1	73.1	72.7	70.2	45.3	84.9	69.8	86.0	69.2	71.7	69.1
%	3 Pipe	124.0	136.5	122.0	159.6	N/A	151.0	109.8	173.7	144.4	N/A	141.3	155.6	118.1

Table 2: 2-pipe Designs with Different Memory Access Time

	mem.latency	perf.metrics	adp.dec	adp.enc	bas.math	b.count	crc32	q.sort	sha.encr.	str.search	endian	b.search	cstr.	dijkstra
1-pipe	1CC	CC ('000s)	2624	8626	3130	5810	1029	4470	2441	1118	1060	3660	1300	1940
		Exec. Time (ms)	116	382	142	266	45	202	108	51	46	169	59	88
	2CC	CC ('000s)	2856	9401	3255	6719	1403	5299	3050	1214	1420	4395	1605	2354
		Exec. Time (ms)	127	417	148	308	62	240	135	55	61	203	73	107
	3CC	CC ('000s)	3057	10177	3380	7628	1710	6221	3660	1423	1780	5129	1910	2768
		Exec. Time (ms)	135	451	154	349	75	281	163	65	77	237	87	126
2 Pipes	1CC	CC (1000s)	2083	7215	2310	4343	810	3399	1830	832	810	3085	1100	1531
		Exec. Time (ms)	68	238	74	135	25	123	58	26	25	99	39	48
	2CC	CC ('000s)	2332	7884	2400	5042	1278	3826	2135	892	1120	3744	1275	1880
		Exec. Time (ms)	77	260	74	163	39	139	68	27	35	121	45	59
	3CC	CC ('000s)	2519	8607	2495	5997	1590	4641	2441	952	1415	4401	1453	2291
		Exec. Time (ms)	83	284	78	194	49	168	78	30	45	142	52	71
improvement	1CC	Exec.Time	69	60	91	97	79	64	84	97	78	70	51	85
	2CC	(%)	64	60	91	96	55	72	97	99	73	68	60	83
	3CC		63	59	91	87	52	67	108	119	72	67	67	76

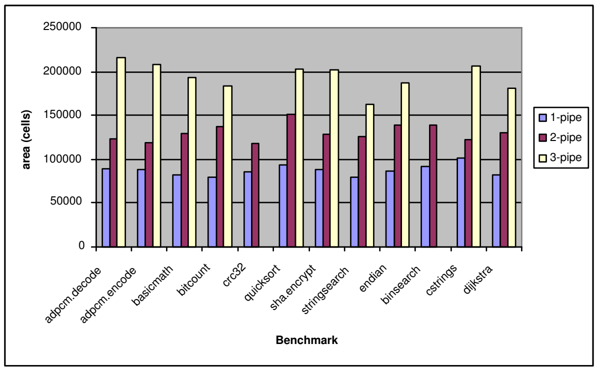

Figure 7: Area

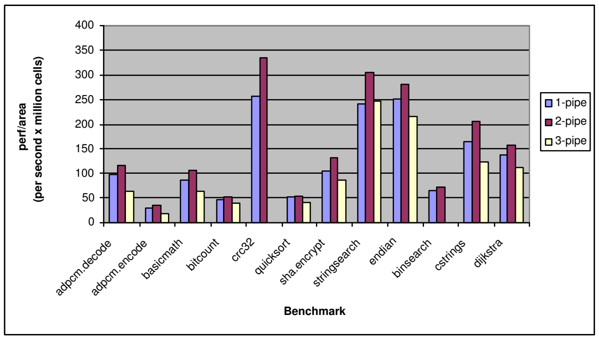

Figure 8: Performance/Area

piler, 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. Note, the area is measured in cells. A cell is approximately equivalent to 0.4 2-input NAND gates. 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 short 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 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 shown 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. Here we assume the memory is on-chip, with a small access time as compared to off-chip memory. 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 affects the performance im-

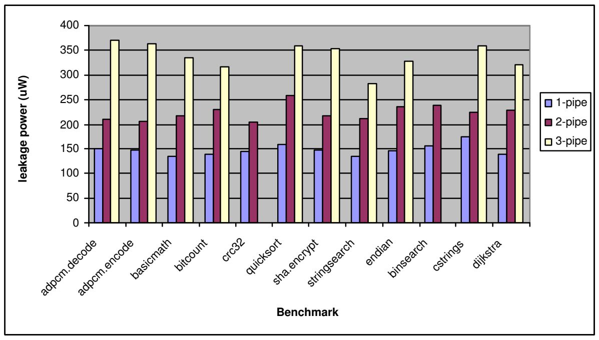

Figure 9: Leakage Power

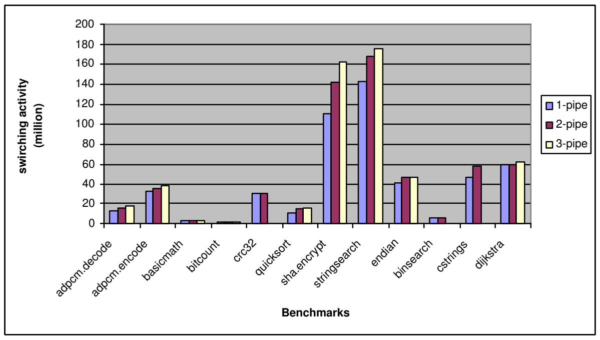

Figure 10: Switching Activity

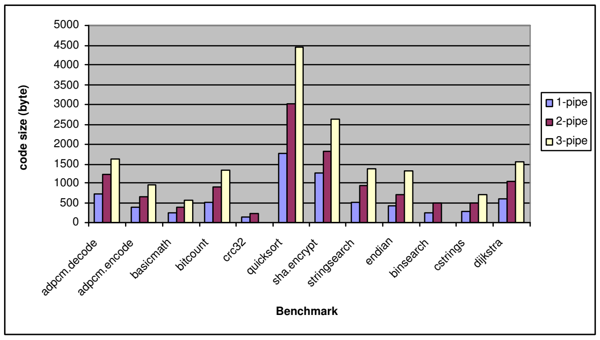

Figure 11: Code Size

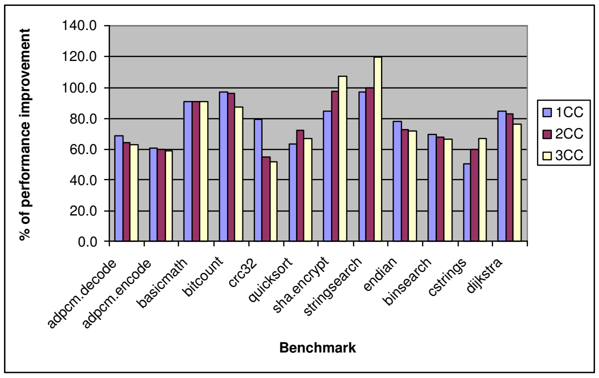

Figure 12: Performance of Varying Memory Access Latency

provement. 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 instructions (the related work and approach can be found in [15]), 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

[1] Xtensa processor. Tensilica Inc. (http://www.tensilica.com).

[2] N. Binh, M. Imai, and Y. Takeuchi. A performance maximization algorithm to design asips under the constraint of chip area including ram and rom size. In *ASP-DAC*, 1998.

[3] N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi. A hardware/software partitioning algorithm for designing pipelined asips with least gate counts. In *Proc. of the 33rd DAC*, pages 527–532. ACM Press, 1996.

[4] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, 2002.

[5] H. Choi, J. H. Yi, J.-Y. Lee, I.-C. Park, and C.-M. Kyung. Exploiting intellectual properties in asip designs for embedded dsp software. In *Proceedings of the 36th DAC*, pages 939–944. ACM Press, 1999.

[6] A. Ferrante, G. Piscopo, and S. Scaldaferri. Application driven optimization of vliw architectures: A hardware-software approach. In *Proceedings of Real Time and Embedded Technology and Applications Symposium*, pages 128 – 137. IEEE Computer Society, 2005.

[7] D. Goodwin and D. Petkov. International conference on compilers, architecture and synthesis for embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137 – 147. ACM Press New York, NY, USA, 2003ISBN:1-58113-676-5.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *In IEEE 4th Annual Workshop on Workload Characterization, Austin, TX*, pages 83–94, December 2001.

[9] M. Jacome, G. de Veciana, and C. Akturan. Resource constrained dataflow retiming heuristics for vliw asips. In *Proceedings of the seventh CODES*, pages 12–16. ACM Press, 1999.

[10] M. F. Jacome, G. de Veciana, and V. Lapinskii. Exploring performance tradeoffs for clustered vliw asips. In *Proceedings of the 2000 ICCAD*, pages 504–510. IEEE Press, 2000.

[11] M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Bal-akrishnan. Evaluating register file size in asip design. In *CODES*, 2001.

[12] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sar-rafzadeh. Instruction generation for hybrid reconfigurable systems. In *ICCAD*, 2001.

[13] V. Kathail, shail Aditya, R. Schreiber, B. R. Rau, D. C. Cron-quist, and M. Sivaraman. Pico: Automatically designing custom computers. In *Computer*, 2002.

[14] S. Kobayashi, H. Mita, Y. Takeuchi, and M. Imai. Design space exploration for dsp applications using the asip development system peas-iii. In *ASSP*, 2002.

[15] S. Radhakrishnan, H. Guo, and S. Parameswaran. Dual-pipeline heterogeneous asip design. In *Proceedings of CODES + ISSS*, pages 12 – 17. IEEE Computer Society, 2004.

[16] J. E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

[17] F. Sun, N. Jha, S. Ravi, and A. Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *Proceedings of Real Time and Embedded Technology and Applications Symposium*, pages 551 – 556. IEEE Computer Society, 2005.

[18] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Synthesis of custom processors based on extensible platforms. In *ICCAD*, 2002.

[19] J.-H. Yang, B.-W. Kim, et al. Metacore: an application specific dsp development system. In *DAC*, 1998.

[20] Q. Zhao, B. Mesman, and T. Basten. Practical instruction set design and compiler retargetability using static resource models. In *DATE*, 2002.