Tradeoffs in the Design Space Exploration of Application-Specific Processors

N. Kavvadias and S. Nikolaidis

Electronics and Computers Division,
Department of Physics,
Aristotle University of Thessaloniki,
54124, Thessaloniki, Greece

E-mail: nkavv@skiathos.physics.auth.gr

Introduction

- Embedded processors are extensively used in wireless communications and multimedia consumer applications
- Such processors present interesting architectural refinements in order to achieve performance/cost effective support of power hungry algorithms e.g. for video compression and decompression
- Thus, the embedded systems industry has shown an increasing interest in Application-Specific Instruction-Set Processors (ASIPs) which are processors designed to exploit special characteristics of a particular application or set of applications
- In the process of designing an ASIP, obtaining best results requires proper decisions at the joint instruction set and micro-architectural level
- ASIPs are considered as a balance between two extremes: ASICs and general-purpose processors

Key issues in ASIP design

- Benefits and challenges in ASIP design
  - ✓ Maintain a level of flexibility/programmability through the use of an instruction set
  - ✓ Overcome the limitations of conventional RISC/DSP architectures (restricted parallelism, high energy consumption)
  - ✓ Definition of the design space of both instruction sets and micro-architectures to be explored
  - ✓ Development of *tools* (assembler, linker, compiler backend, cycle-accurate simulator, debugger) for efficient design space exploration
- Steps in an ASIP design flow
  - ✓ Application analysis, architecture design space exploration, instruction set generation, code synthesis and hardware synthesis

Establishing an architecture template for design space exploration

- Need for a good parameterized model of the architecture
- The size of the design space is defined by the parameter set and the range of values which can be assigned to these parameters
- Parameters of architecture models:
  - number, types and attributes (latency, throughput, I/O ports) of functional units and storage elements
  - interconnection resources
  - number of pipeline stages
  - instruction level parallelism, subword parallelism
  - addressing support
- In our case: 4-stage pipeline (IF-ID-EX-WB), Harvard bus architecture, 2-cycle branch penalty
- Multi-cycle functional units are supported

The proposed approach for ASIP design space exploration

- *Application analysis*: identify requirements of hardware modules and extract the CDFG of the application

- Each node in the CDFG corresponds to a basic block and contains an instruction list

  ✓ A pass was written in MachSUIF to produce the corresponding DAGs

- The DAGs are studied in order to derive schedules of primitive operations. Sequences or groups of operations that have data dependencies are amenable to replacement by complex instructions (possibly multi-cycle)

  ✓ Multi-cycle instructions reduce the number of instruction fetches which contribute significantly to the system energy consumption

- *Performance estimation*: the target application is simulated on the architecture model to evaluate the effect of architecture changes

Case study: Full-search motion estimation

- Case study: Full-search block matching motion estimation (FSME) application used in MPEG-compliant video coding
- Used to remove the temporal redundancy in video sequences which is determined by similarities amongst consecutive pictures
- *Input*: two neighboring frames
- *Output*: the displacements of pixel blocks between these pictures (termed as motion vectors)
- The computational complexity of the motion estimation algorithm ranges from 50% to 75% for both MPEG-2 and MPEG-4
- Example: call graph profiling on the *mpeg2enc* benchmark of the MediaBench suite, reveals that 63% of the overall processing time is spend in procedure *dist1*, which is the native implementation of FSME

A detailed view of the case study algorithm

```
for x=0 to H/B-1, step 1
  for y=0 to W/B-1, step 1
    min = 255*B*B;
    for i=-p to p, step 1
      for j=-p to p, step 1
        dist = 0;
        for k=0 to B-1, step 1
          for l=0 to B-1, step 1
            p1 = current[B*x+k,B*y+l];
            if  (p2 out of picture borders) then
              p2 = 0;
            else
              p2 = reference[B*x+i+k,B*y+j+l];
            endif;
            dist = dist + absolute_value(p1-p2);
          endfor;
        endfor;
        if (dist < min) then
          min = dist;
          MVx = i;
          MVy = j;
        endif;
      endfor;
    endfor;
  endfor;
endfor;
```

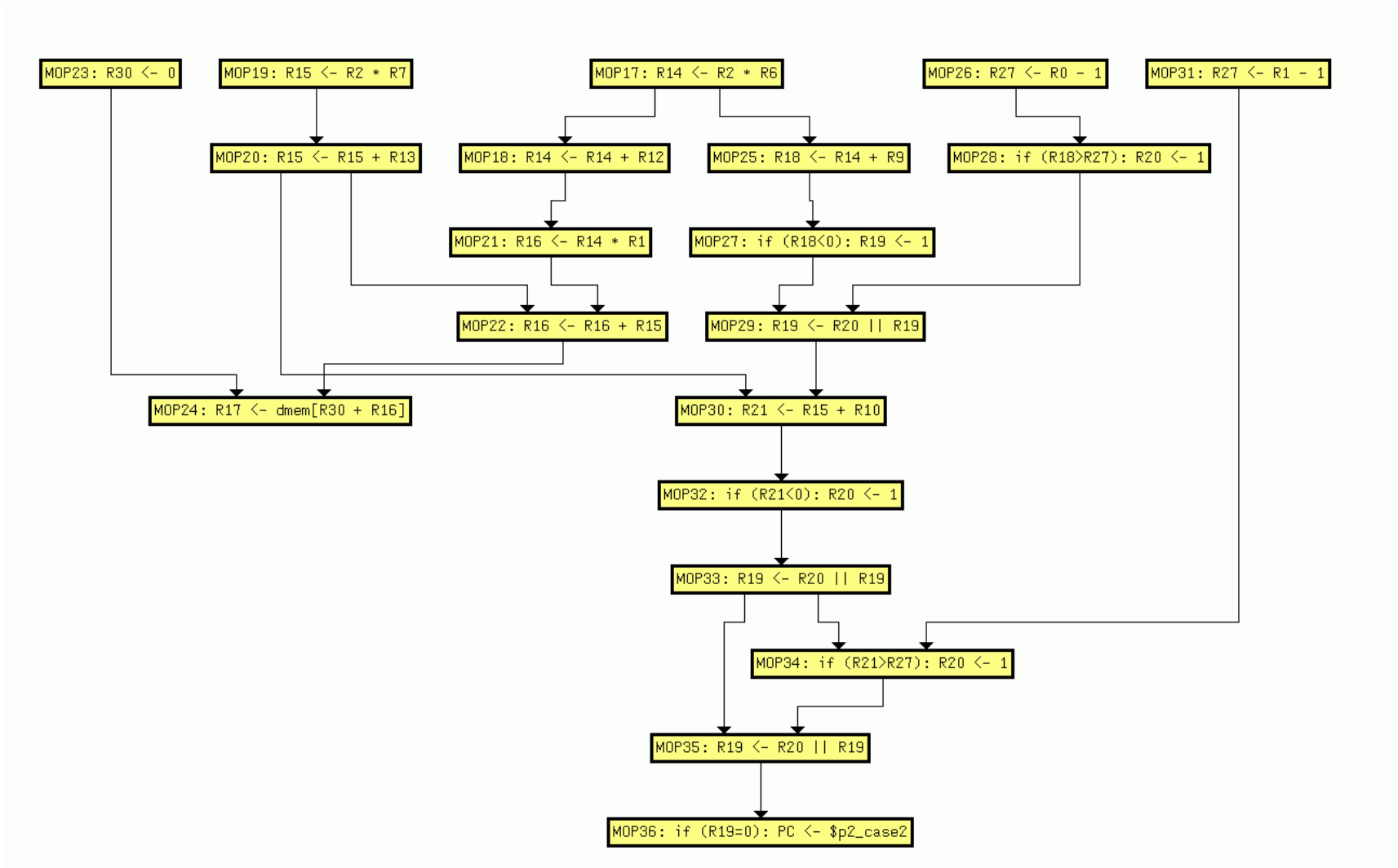
- The algorithm consists of 3 double nested loops:
- (x,y): address a block in the “current” picture
- (i,j): select a block in the “reference” picture
- (k,l): calculate the sum of the absolute differences between the pixels in the current and reference block

- Inner loop computations:

1. Load pixel p1 from current picture into local register
2. Evaluate a memory access check and load pixel p2
3. Compute the SAD

Micro-operation description of the application

- Directed-acyclic graph (DAG) for an inner loop basic block (*loop6*)



Baseline architecture for the FSME ASIP

- In the first step, we derive a baseline architecture for the ASIP
- Architecture parameters:
  - ✓ single 24b ALU
  - ✓ single-cycle 8bx8b multiplier
  - ✓ shifter with left/right shift capability
  - ✓ single register file with 3 read ports and 1 write port
  - ✓ dedicated adder for PC update
  - ✓ local (on-chip) instruction and data memory
- The initial instruction set consists of 19 instructions (following table)
- A retargetable compiler would cover the entire set of C-level operators. In this case, 30 to 40 primitive operations and 2 or 3 addressing modes can be identified

Initial instruction set

Instruction	Instruction definition	MOP description
Add	ADD Rd, Rs1, Rs2	Rd ← Rs1 + Rs2
Multiply	MUL Rd, Rs1, Rs2	Rd ← Rs1 * Rs2
Logical OR	ORR Rd, Rs1, Rs2	Rd ← Rs1 || Rs2
Compare	CMP Rs1, Rs2	flags: Rs1 - Rs2
Set on greater than	SGT Rd, Rs1, Rs2	if (Rs1>Rs2) : Rd ← 1
Load register from mem	LD Rd, Rb, Ro	Rd ← m[Rb + Ro]
Store register to mem	ST Rs, Rb, Ro	m[Rb + Ro] ← Rs
Add immediate	ADDI Rd, Rs, #imm	Rd ← Rs + imm
Subtract	SUBI Rd, Rs, #imm	Rd ← Rs - imm
Set on less than immediate	SLTI Rd, Rs, #imm	if (Rs<imm): Rd ← 1
Shift left logical	SHL Rd, Rs, #imm	Rd ← Rs >> imm
Move immediate	MOVI Rd, #imm	Rd ← imm
Move register	MOV Rd, Rs	Rd ← Rs
Move register negated	MVN Rd, Rs	Rd ← -Rs
Branch unconditional	B $target	PC ← target
Branch if less than	BLT $target	if (N=1): PC ← target
Branch if greater than	BGT $target	if (N=0): PC ← target
Branch if register clear	BRC Rd, $target	if (Rd=0): PC ← target
Branch if register set	BRS Rd, $target	if (Rd=1): PC ← target

Architectural modifications to the basic instruction set/ micro-architecture

- Modifications are applied to the initial specification of the ASIP and 3 different configurations are produced
  - ✓ *Version 1*: Effect of register file topology (additional register files are introduced to store the index values for the loops in the algorithm)
  - ✓ *Version 2*: Effect of implementing the multiply-add operation
  - ✓ *Version 3*: Effect of replacing chainable operations by complex instructions
    - *Variation A*: a constraint has been set on the maximum number of consecutive MOPs accounted when forming a new single cycle instruction
    - *Variation B*: no constraint is applied on the maximum number of consecutive MOPs and multi-cycle instructions can be identified

Version 1: Effect of register file topology

- The registers in the initial register file are reduced from 32 to 16 and 3 additional register files are used to store the running index, final and step values
- The index register files consist of 8 12-bit registers with 2 read and 1 write ports
- Tradeoffs in version 1 of the ASIP:
  - ✓ An additional address decoder for the index register files is required, and this overhead has to be compensated by the reduction in energy consumption due to fewer storage bits
  - ✓ The use of different data widths in the datapath requires the use of size-extension circuitry, which negatively affects propagation delay
- Results:
  - ✓ The number of execution cycles is not affected
  - ✓ Energy consumption is reduced by 35%

Version 2: Effect of the multiply-add operation

- The resulting architecture (version 1) is used as input to subsequent design space exploration steps
- Application analysis has identified a frequent pattern consisting of the MUL and ADD micro-operations
- A new instruction, MADD, is accepted that incorporates the MUL and ADD micro-operations in the same control step
- Only minimal changes to the interconnection of the ALU and multiplier of the baseline architecture are required
- In case the processor cycle time is affected, a solution is to pipeline the MADD instruction to the WB stage
- Results:
  - ✓ Reduction of 9% in the clock cycles and 14% in the energy consumption are observed (against version 1)

Version 3: Effect of replacing chainable operations by single instructions (1)

- **Variation A**: Four new instructions have been identified and added to the initial instruction set
- Hardware modifications (FOR instruction): Dedicated adder, address decoder for selecting the running loop index

Instruction	Instruction definition	Operation pattern
Load register from mem (immediate offset)	LDI Rd, Rb, #imm	movi Ro, #imm
		ld Rd, Rb, Ro
Move immediate and shift left	MVISL Rd, #shamt, #imm	movi Rd, #imm
		shl Rd, Rs, #shamt
Loop increment and branch	FOR Rix, $loop_start	addi Rix, Rix, #1
		cmp Rix, Rfin
		blt $loop_start
Absolute difference	ABS Rd, Rs1, Rs2	sub Rd, Rs1, Rs2
		cmp Rd, (Rtemp=0)
		bgt $abs_oper2
		mvn Rd, Rd
		abs_oper2: …

- The number of machine cycles is reduced by 35% compared to the initial configuration, and energy consumption is reduced by a factor of 2.2

Version 3: Effect of replacing chainable operations by single instructions (2)

- **Variation B**: Two additional instructions have been identified and incorporated in Version 3 (variation A) instruction set
- DMC (Data Memory Check) instruction undertakes the task of a memory sizer/controller. Replaces the conditional in computation 2 of the innermost loop in the FSME algorithm and completes in 4 cycles

Instruction	Instruction definition	Operation pattern
Data memory check	DMC Rs1, Rs2, Rs3, Rs4	SLTI Rtmp1, Rs1, #imm
		SGT Rtmp2, Rs1, Rs3
		ORR Rtmp1, Rtmp2, Rtmp1
		SLTI Rtmp2, Rs2, #imm
		ORR Rtmp1, Rtmp2, Rtmp1
		SGT Rtmp2, Rs2, Rs4
		ORR Rtmp1, Rtmp2, Rtmp1

- Reduction in clock cycles of 25% compared to version 3A. The number of instruction fetches is also significantly reduced

Comparison

- Performance estimation for the examined architectural configurations

Architecture configuration	Static instruction count	Dynamic instruction count	Cycles	% diff	Power (W)	% diff	Energy (J)	% diff
Original	79	233523086	246370022	0.00	2.087	0.00	6.016	0.00
version 1	81	235230088	246370024	0.00	1.627	-22.02	3.930	-34.67
version 2	77	213040691	224179827	-9.01	1.631	-21.86	3.589	-40.43
version 3A	56	155616986	159406177	-35.30	1.752	-16.05	2.738	-54.49
version 3B	48	109596592	132406192	-46.26	1.652	-20.82	2.145	-64.35

- To summarize our results, the final configuration of the ASIP presents a reduction of 40% on code size, 50% on the dynamic instruction count and number of cycles, 20% in power consumption and nearly 65% on the energy consumption compared to the baseline architecture

Conclusions and future work

- In this paper we identified tradeoffs inferred in ASIP design by different register file topologies and complex instruction generation
- The effect of different architectural configurations on execution performance and energy consumption of the processor was examined
- The performance simulator is being re-coded in SystemC
- A power-characterized hardware component library should be built, since the models used are rather obsolete (0.5um CMOS process)
- The primitive instructions can be obtained in an architecture-neutral manner if C source code is compiled to the SUIF virtual instruction set
- In order to run larger benchmarks (e.g. from MediaBench) a compiler backend is required. In our case, the FSME algorithm was handcoded in the ASIP assembly