Digital Signal Processors (DSPs)

- trend towards more "general-purpose" DSPs: programmable
  - optimized for high throughput for special applications
- used as workhorse in high-performance embedded systems
  - execution throughput, code size, power consumption do matter
- typical features:
  - clustered VLIW architectures
  - non-homogeneous register sets
  - dual memory banks
  - address generation units
  - SIMD parallelism on subwords
  + MAC instruction (multiply-accumulate)

Domain-specific processors

- programmability
- efficiency (MIPS/Watt)
- flexibility

Ecehird systems and compiler requirements

- Code optimization for embedded processors
- digital signal processors
- multimedia processors
- application specific processors
- architecture exploration
- target machine description languages
- retargetable compilation

Books:


focus on optimizations

Hardware-software co-design

Architecture exploration

Target machine description languages

Retargetable compilation

Embedded systems and compiler requirements

- Code optimization for embedded processors
- digital signal processors
- multimedia processors
- application specific processors
- architecture exploration
- target machine description languages
- retargetable compilation

Digital Signal Processors (DSPs)

- trend towards more "general-purpose" DSPs: programmable
  - optimized for high throughput for special applications
- used as workhorse in high-performance embedded systems
  - execution throughput, code size, power consumption do matter
- typical features:
  - clustered VLIW architectures
  - non-homogeneous register sets
  - dual memory banks
  - address generation units
  - SIMD parallelism on subwords
  + MAC instruction (multiply-accumulate)
Demand on code quality

(Critical) code for DSPs was traditionally written in assembler.

More complex embedded software, shorter development time to market, and real-time constraints make assembler programming no longer feasible. The lingua franca is C(++)

Compilation problems in embedded processors

Not compiler-friendly:

But compilers traditional prefered regular architectures

...special instructions e.g. MAC (multiply-accumulate)

special addressing modes

Constrained parallelism

Regular data paths

Designed for efficiency, not for ease of programming

Not compiler-friendly:

Compilers for embedded processors must generate extremely efficient code:

- Instruction scheduling
- Mapping instructions to clusters
- Residence constraints on concurrent execution (load + add + mul, load + add + mul, ...)
- Code size
- System-on-chip
- Power/energy consumption
- Real-time constraints
- Code size

Compiler for embedded processors needs information about instruction scheduling and register allocation, whereas compiler for DSPs needs information about (concurrent) need of resources.

Heuristic [Leupers'00]: iterative optimization with simulated annealing

Heuristic [Deitrich]: iterative optimization with simulated annealing

Mapping instructions to clusters

More phase ordering problems: Code generation for DSPs

Example: Hitachi SH-3 DSP

Example: Hi68 SH-3 DSP

Heuristic [Leupers'00]: iterative optimization with simulated annealing

Mapping instructions to clusters

Heuristic [Deitrich]: iterative optimization with simulated annealing

Battery lifetime

Heat dissipation

Power / energy consumption

Real-time constraints

Code size

More complex embedded software: shorter time to market

Critical code for DSPs was traditionally written in assembler.

Demand on code quality

Compilation problems in embedded processors

More phase ordering problems: Code generation for DSPs

Compilation problems in embedded processors

More phase ordering problems: Code generation for DSPs
Conflicts in instruction selection for MAC instructions

Instruction selection for DAGs – NP-complete

Tree-pattern-matching algorithm (dynamic programming) works fine as a heuristic for most regular processor architectures

Some VLIW architectures have multiple (typically 2) memory banks

Heuristic: Saghir/Chow/Lee ASPLOS-VII 1996:

Optimal partitioning is NP-complete

Garey/Johnson '79

Heuristic: Saghir/Chow/Lee ASPLOS-VII 1996:

Exploiting dual data memory banks in digital signal processors

2 phases:

1. Build bank interference graph

2. Partition bank interference graph

Where n and d access different variables and

But one load/store unit was already assigned some n ∈ E.

If load instruction e ∈ E could be scheduled

Place instructions from into L1W as long as units available...

Each zero-indegree set:

Start with empty bank interference graph

Extension of greedy list scheduling:

Phase 1: Construct bank interference graph

Data layout problem: how to exploit parallel loads/stores?

Memory layout for dual-banked memory

In phase 1 the interference graph is constructed.

In phase 2 the interference graph is partitioned.

Problem: Common subexpressions could be part of multiple possibilities for covers by complex instructions – at most one can be realized.

As a heuristic for most regular processor architectures

Conflict in Instruction Selection for MAC Instructions

Constraint-logic programming

[Bashford '99]
Example: Constructing the bank interference graph

\[ C[i] = A[i] + D[i] \]

... 

C[i] = B[j] / C[k]
for (i=0; i<10; i++)
C[i] = B[j] / C[k]

Memory layout for dual-banked memory

- Minimum-cost labeling of the graph: simulated annealing heuristic
- Common interference graph with different kinds of edges
- Integration of bank allocation and register allocation

**Extension:** [Sudarsanam/Malik ICCAD'95, TODAES'2000]

Memory layout for dual-banked memory (3)

Phase 2: Partitioning the bank interference graph

1. Remove a node \( v \) from \( P \)
2. Cost of a partition \( \sum \{\text{edge } u' \in \text{partition } P \} \)
3. Start with partition \( P \) as all nodes in bank \( 1 \)
4. Greedy heuristics - here for 2 banks

**Integration of bank allocation and register allocation:**

- Assumption of a large, general-purpose register file is unrealistic
- Requires alias analysis, especially for array elements

+ 13.4% improvement in practice

13.4% improvement in practice – requires alias analysis, especially for array elements – assumption of a large, general-purpose register file is unrealistic

**Example (cont.): Partitioning the bank interference graph**

...
Address generation units in DSPs

If the next stack address differs from the previous one only by a small constant, the AGU can be used in parallel to the CPU datapaths.

**Address generation units in DSPs**

**AGU**

Examples: TI C25/5, Motorola 56000, ADSP-210x...

Branch-and-bound algorithm: [Liao et al. PLDI’95]

Find a maximum-weight Hamiltonian path in the affinity graph.

1. Build a variable affinity graph.
2. Find a maximum-weight Hamiltonian path in the affinity graph.

Heuristic [Leupers’00]: genetic algorithm.

**Heuristic**

Optimal solution: NP-complete

(unsatisfactory)

where $\nu > I = 3^{I/2}$ and $d$ otherwise.

and $\nu > I = 3^{I/2}$ is minimized.

such that $C_{\text{total}} = I$.

An initial offset mapping

$\nu = I$ is done after scheduling.

An initial offset mapping $\nu = I$ is done after scheduling.

$\nu = I$ is done after scheduling.

$\nu = I$ is done after scheduling.

One single address register, containing value $r$.  

General offset assignment (GOA) problem

Extension of SOA for multiple ($k < 1$) address registers.

**Algorithm for SOA**

1. Build a variable affinity graph.
2. Find a maximum-weight Hamiltonian path in the affinity graph.

Heuristic [Bartley’92]: branch-and-bound algorithm.

**Heuristic**

Branch-and-bound algorithm: [Liao et al. PLDI’95]

Heuristic [Leupers’00]: genetic algorithm.

**Heuristic**

Optimal solution: NP-complete

(unsatisfactory)

where $\nu > I = 3^{I/2}$ and $d$ otherwise.

and $\nu > I = 3^{I/2}$ is minimized.

such that $C_{\text{total}} = I$.

An initial offset mapping

$\nu = I$ is done after scheduling.

An initial offset mapping $\nu = I$ is done after scheduling.

An initial offset mapping $\nu = I$ is done after scheduling.

An initial offset mapping $\nu = I$ is done after scheduling.

One single address register, containing value $r$.  

**Heuristic**

Branch-and-bound algorithm: [Liao et al. PLDI’95]

Heuristic [Bartley’92]: branch-and-bound algorithm.

**Heuristic**

Branch-and-bound algorithm: [Liao et al. PLDI’95]

Heuristic [Bartley’92]: branch-and-bound algorithm.

**Heuristic**

Branch-and-bound algorithm: [Liao et al. PLDI’95]

Heuristic [Bartley’92]: branch-and-bound algorithm.

**Heuristic**

Branch-and-bound algorithm: [Liao et al. PLDI’95]

Heuristic [Bartley’92]: branch-and-bound algorithm.

**Heuristic**

Branch-and-bound algorithm: [Liao et al. PLDI’95]
Exploiting SIMD instructions

[Example]

Energy optimization (1)

Hardware support for energy saving:

- Memory access
  - Memory access
  - Switch off speculation
  - Pipelining

- Clock gating

- Reduce voltage and frequency for non-critical program regions

- Volatile scheduling

Example: Register pipelining for ARM Thumb

```
code:  ...  
loop:  ...  
```

Example: Register pipelining for ARM Thumb

```
code:  ...  
loop:  ...  
```

but shorter time may not necessarily yield less energy

\[
\text{Energy consumption} = \int_0^T \text{power} \, dt
\]

Instruction decoding / execution: varying base costs

Number of ones (‘weight’) on buses may also influence power

Switching activities on buses at the bit level

Other factors that can be exploited by software:

Energy optimization (2)

Reduce spill code by space-aware code selection / scheduling

Example: C code:

```
c int a[100];
c = a;
for(i=1;i<100;i++)
{b+=*c;
b+=*(c+7);
c+=1;}
```

Optimized for time:

```
LDR r3,[r2,#0]
ADDr3,r0,r3
MOV r0,#28
LDR r0,[r2,r0]
ADDr0,r3,r0
ADDr2,r2,#4
ADDr1,r1,#1
CMPr1,#100
BLT Loop
```

2096 cc, 19.92 µWs

Optimized for energy:

```
ADDr3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,r10
MOV r10,r9
MOV r9,r8
LDR r1,[r4,r0]
ADDr0,r3,r1
ADDr4,r4,#4
ADDr5,r5,#1
CMPr5,#100
BLT Loop
```

2231 cc, 16.47 µWs
Energy optimization (4)

Simulation-based power models
- Input: detailed description of the target processor
- Simulate the architecture cycle by cycle with a given program
- Output: detailed function inlining / tail merging
- Code size reduction
- Retargetability

Examples: [Lee et al. LCTES’01], [Steinke et al. PATMOS’01]

Measurement-based models
- Assumed power = linear combination of these, weighted by coefficients
- for a family of processors
- Know the set of most influential factors for power consumption

Hardwired description languages for reconfigurable code generation
- Instruction semantics
- netlist of hardware blocks
- units, latencies, restrictions
- Complete description
- Precise simulation

Structural models
- Instructions: structural semantics
- netlist of hardware blocks
- Units, latencies, restrictions
- Complete description
- Precise simulation

Behavioral models
- High abstraction from hardware
- Procedure-level, grammar-based
- Semantics
- Assembly representation
- For code generation in compilers

Mixed models
- Procedure-level, grammar-based
- Assumption structural constraints
- high abstraction from hardware
- retargetable code generation

Simulation-based power models
- Input: detailed description of the target processor
- Simulate the architecture cycle by cycle with a given program
- Output: detailed function inlining / tail merging
- Code size reduction
- Retargetability

Outlook – Requirements for code optimization
- Retargetability!
- Codesize reduction
- Selective function inlining / tail merging
- Instruction selection / compaction
- Parallelism
- Address generation
- Minimize memory accesses – register allocation
- Power consumption
- Instruction selection / compaction
- Selective function inlining / tail merging
- Code size reduction
- Retargetability

Methodology:
- Hardware description language
- Behavioral language
- Retargetable code generation
- Power models
- Hardware
- Software
- Simulators
- Measurement models
- Compilation strategies
- Retargetability
- Simulation-based power models
- Measurement-based power models
- Code size reduction
- Selective function inlining / tail merging
- Instruction selection / compaction
- Parallelism
- Address generation
- Minimize memory accesses – register allocation
- Power consumption
- Instruction selection / compaction
- Selective function inlining / tail merging
- Code size reduction
- Retargetability

Simulation-based power models
- Input: detailed description of the target processor
- Simulate the architecture cycle by cycle with a given program
- Output: detailed function inlining / tail merging
- Code size reduction
- Retargetability

Examples:
- [Lee et al. LCTES’01], [Steinke et al. PATMOS’01]

Measurement-based models
- Assumed power = linear combination of these, weighted by coefficients
- for a family of processors
- Know the set of most influential factors for power consumption

Hardwired description languages for reconfigurable code generation
- Instruction semantics
- netlist of hardware blocks
- units, latencies, restrictions
- Complete description
- Precise simulation