CODE GENERATION FOR IRREGULAR ARCHITECTURES

- DSP Processor Features
- Constraints by Irregular Register Sets
- Memory layout for multi-banked memory
- Exploiting Address Generation Units
- Exploiting SIMD Instructions
- Energy Optimization

Domain-specific processors

- efficiency (MIPS/Watt)

  - ASICs
  - DSPs
  - μCs
  - NPUs
  - GPPs

  - flexibility
  - programmability

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 time/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)

Literature

Books:

Demands on code quality

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

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

Compilers for embedded processors must generate extremely efficient code:
- code size
  system-on-chip
  on-chip RAM / ROM
- performance
  real-time constraints
- power / energy consumption
  heat dissipation
  battery lifetime

Compilation problems in embedded processors

Not compiler-friendly:
- designed for efficiency, not for ease of programming
- irregular data paths
- special purpose registers
- constrained parallelism
- advanced addressing modes
- special instructions e.g. MAC (multiply-accumulate)

But compilers traditionally preferred regular architectures ...

More phase ordering problems: Code generation for DSPs

Clustered VLIW architectures, e.g. TI C6201:

Example: Hitachi SH3-DSP

Heuristic [Leupers’00] iterative optimization with simulated annealing

Residence constraints on concurrent execution (load + mul, add + mul, …)
Instruction scheduling and register allocation are not separable!
Phase-decoupled standard methods generate code of poor quality.
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

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

Constraint-logic programming [Bashford’99]

Memory layout for dual-banked memory

Some VLIW architectures have multiple (typically 2) memory banks to duplicate the memory–register bandwidth

Constraint-logic programming [Bashford’99]

Memory layout for dual-banked memory (2)

naive method: duplicate all data over all banks
- load always from closest bank
- stores must be duplicated as well for consistency
- if direct moves are possible: how to schedule them?
- space requirements...

optimal partitioning is NP-complete [Garey/Johnsson’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 heuristically

Memory layout for dual-banked memory (3)

Phase 1: construct bank interference graph
- start with empty bank interference graph
- extension of greedy list scheduling:
at each zero-indegree set \( z \)
... place instructions from \( z \) into LIW as long as units available ...
if load instruction \( v \in z \) could be scheduled
but one Load/Store unit was already assigned some \( u \in z \)
where \( u \) and \( v \) access different variables \( var(u) \), \( var(v) \)

then add edge \{ \( var(u) \), \( var(v) \) \} to bank interference graph
weighted e.g. by nesting depth

\[ \text{requires alias analysis!} \]
Example: Constructing the bank interference graph

\[
\begin{align*}
B[i] &= B[j] \times D[k] \\
C[i] &= B[j] / C[k] \\
\end{align*}
\]

for \( i=0; \ i<10; \ i++ \)

\[
\begin{align*}
C[i] &= A[i] + D[i] \\
\end{align*}
\]

Memory layout for dual-banked memory (3)

Phase 2: partition the bank interference graph

- greedy heuristic — here for 2 banks
- start with partition \( P = \{V_1, V_2 = \emptyset\} \), i.e., all nodes in bank 1
- cost of a partition \( P \):

\[
\sum_{i=1}^{\text{banks}} \sum_{u \in V_i, v \in V_j} w(u,v)
\]

- repeat

move a node \( v \) from \( V_1 \) to \( V_2 \)

that yields the maximum cost reduction

until cost cannot be decreased further

+ 13..43% 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

\[
\begin{align*}
B[i] &= B[j] \times D[k] \\
C[i] &= B[j] / C[k] \\
\end{align*}
\]

for \( i=0; \ i<10; \ i++ \)

\[
\begin{align*}
C[i] &= A[i] + D[i] \\
\end{align*}
\]

Cost: 15

Cost: 3

Cost: 2

Memory layout for dual-banked memory (4)

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

Integration of bank allocation and register allocation:

- common interference graph with different kinds of edges
- minimum-cost labelling of the graph: simulated annealing heuristic
Address generation units in DSPs (1)

Address Generation Unit (AGU)

Address registers used for autoincrement / -decrement addressing of
- vector elements (small constant stride through array)
  — largest potential
- scalars on the stack or in global .data segment
  — to optimize scalar code if address register left

If the next address accessed differs from the previous one only by a small
constant, the AGU can be used in parallel to the ALU datapaths
→ more throughput, as ALU is not blocked by address calculations
Examples: TI C2x/5x, Motorola 56000, ADSP-210x, ...
Autoincrement load/store instructions exist on almost all processors
(sometimes called pop/push)

Single Offset Assignment problem (SOA)

Generation of optimal address code for computations on stack-allocated scalar variables, using 1 address register with autoincrement/decrement
- Given: Access sequence $S$ (linear schedule)
- Compute: Memory layout of the variables on the stack
  that minimizes the number of extra address instructions required

Example: $S = \{a, b, c, d, e, f, a, d, a, d, a, f, a, d\}$

Single Offset Assignment problem (SOA)

(one single address register + one constant offset value $r$)

Given:
- $V = \{v_1, \ldots, v_n\}$ local variables to be placed in the stack,
- $S = \{s_1, \ldots, s_l\}$ access sequence, $s_i \in V$, $1 \leq i \leq l$
  (known, as this is done after scheduling),

find a bijective offset mapping $M : V \to \{0, \ldots, n-1\}$ (stack addresses)
such that $Cost(M) = 1 + \sum_{i=1}^{l} z_i$ is minimized,
where $z_i = 1$ if $|M(s_{i+1}) - M(s_i)| > r$, and 0 otherwise.
(Usually, this offset for autoincrement/decrement is $r = 1$.)

Optimal solution: NP-complete!
Algorithms for SOA

\(O(n^3)\) heuristic [Bartley'92]

Branch-and-bound algorithm [Liao et al. PLDI'95]
- build a variable affinity graph:
  - For each pair \((s_i, s_{i+1})\) in \(S\) increase weight of edge \((s_i, s_{i+1})\) by 1
- find a maximum-weight Hamiltonian path in the affinity graph by Branch&Bound, using a modified Kruskal MST algorithm

Example: \(S = (a, b, c, e, f, d, a, d, e, c, d, f, a, d)\)

Heuristic [Leupers’00]: genetic algorithm.

General offset assignment (GOA) problem

GOA: Extension of SOA for multiple \((K > 1)\) address registers:
- variable (index) set \(V = \{1, \ldots, n\}\),
- access sequence \(S = \langle v_{s_1}, \ldots, v_{s_{nm}} \rangle \in V^m\), where \(n \leq m\), and
- range of possible autoinc/-decr. offsets,

find a \(K\)-coloring of the elements in the address sequence \(S\)
- = partitioning of \(S\) into \(K\) sub-access sequences \(S_k, k = 1 \ldots K\),
- each color defines a sub-access-stream \(S_k = \langle v_{s_k(1)}, v_{s_k(2)}, \ldots, v_{s_k(m)} \rangle\),

for each \(S_k\), all elements appear in the same relative order as in \(S\)

and find a data layout \(\pi: V \rightarrow V\) in memory

that minimize overall cost \(C_K = \sum_{k=1}^{K} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} \cdot t_{i,j} = (\langle i, j \rangle)\)

where \(c_{i,j} \in \{0, 1\} = \text{cost for changing address register from } i \in V \text{ to } j \in V\)

and \(t_{i,j} = \text{number of times where address } j \text{ occurs in } S \text{ directly after } i\).

Exploiting SIMD instructions

ADD2 performs two 16-bit integer additions on the same functional unit in one clock cycle.

Operands must reside in lower and upper 16 bits of the same registers.

Most other instructions (Load, Store, Copy, ...) work on 2x16bit pairs in the same way as for 32bit words, thus same opcode.
- Requirement for load/store of 2x16bit, 4x8bit etc.:
  Consecutive layout in memory, possibly alignment constraints

Remark: SIMD data types such as 2x16bit pairs, 4x8bit quadruples etc. are sometimes referred to as "vectors"
Instruction Selection for SIMD Instructions

Previously:

Pattern matching rule \texttt{reg: ADDI ( reg, reg )}

Now add 2 new nonterminals \texttt{reg_hi}, \texttt{reg_lo} (denote upper/lower register halves) and 2 new rules for ADD2:

\texttt{reg_hi: ADD2 ( reg_hi, reg_hi )}
\texttt{reg_lo: ADD2 ( reg_lo, reg_lo )}

Beware of creating artificial dependence cycles when covering nodes with complex patterns! (see lecture on Instruction Selection)

Preparing for Selecting SIMD Instructions

May require loop unrolling to find candidates for ADD2 matching:

```c
void vector_add ( short a[], b[], c[], unsigned int N )
{
    unsigned int i;
    for ( i = 0; i < N; i += 2 ) {
        a[i] = b[i] + c[i];
        a[i+1] = b[i+1] + c[i+1];
    }  
}
```

Example (cont.) — Effect of code for 1 loop iteration

Energy optimization (1)

Hardware-support for energy saving:

- Voltage scaling: reduce voltage and frequency for non-critical program regions
- Clock gating: switch off parts of a processor (e.g. float unit) if not used for a while
- Pipeline gating switch off speculation \(\rightarrow\) reduces unit utilization
- Memory accesses reduce spill code by space-aware code selection / scheduling
Energy optimization (2)

Other factors that can be exploited by software:

- Switching activities on buses at the bit level
  CMOS circuits dissipate power if a gate output changes 0→1 or 1→0
  Number of ones (“weight”) on buses may also influence power
- Instruction decoding/execution: varying base costs
- Energy consumption = ∫₀^T power(t) dt
  but shorter time may not necessarily yield less energy

Energy optimization (3)

Example: Register pipelining for ARM7 Thumb

[Steinke’01, ’02]

C code:

```c
int a[1000];
c = a;
for (i=1; i<100; i++){
  b += *c;
  b += *(c+7);
  c += 1;
}
```

optimized for time:

```assembly
Loop:
  LDR r3, [r2,#0]
  ADD r3, r0, r3
  MOV r0,#28
  LDR r0, [r2,r0]
  ADD r0, r3, r0
  ADD r2, r2, #4
  ADD r1, r1, #1
  CMP r1, #100
  B LT Loop
```

2096cc, 19.92µW s

optimized for energy:

```assembly
Loop:
  ADD r3, r0, r2
  MOV r0,#28
  MOV r2, r12
  MOV r0, #28
  MOV r2, r12
  MOV r0, #28
  MOV r2, r12
  ADD r2, r2, #4
  ADD r1, r1, #1
  CMP r1, #100
  B LT Loop
```

2231cc, 16.47µW s

Energy optimization (4)

Simulation-based power models

- input: detailed description of the target processor
- simulate the architecture cycle by cycle with a given program
- Ex.: SimplePower [Ye et al. DAC 2000], Wattch [Brooks et al. ISCA-2000]

Measurement-based models

- know the set of most influential factors for power consumption
  (for a family of processors)
- assume \( \text{power}(t) \) = linear combination of these, weighted by coefficients
  coefficients found by measuring current drawn for simple test sequences
  with an amperemeter, by regression analysis
- Examples: [Lee et al. LCTES’01], [Steinke et al. PATMOS’01]

Summary – Challenges for DSP code optimization

- Instruction-Level Parallelism, Scheduling, Data layout
  SIMD instructions, MAC, VLIW, clustered VLIW
  Address code generation and stack data layout
  Banked memory
- Power consumption
  power models
  instruction selection and scheduling for low-power
  minimize memory accesses – register allocation
- Code size reduction
  selective function inlining / tail merging, selective loop unrolling
  instruction selection (compact instruction formats)
- Retargetability — Generate or parameterize optimizer from target description