Embedded Software in Real-Time Signal Processing Systems:

Design Technologies

Minxi Gao
Xiaoling Xu

Outline

• Problem definition
• Classification of processor architectures
• Survey of compilation techniques
• Optimization algorithms examples on register allocation, memory allocation and scheduling (by sherry)
• Conclusion
Embedded Systems

- Telecommunications
  - Pagers
  - Cellular phones
- Multimedia
  - Digital video disks (DVD), videophone
- Consumer electronics
  - Microprocessor in a car, antilock brakes.

A paradigm shift from hardware to software

- Possible to include late specification changes in the design cycle
- becomes easier to add new features to it
- easier to reuse previously designed functions: independent from implementation platform (e.g., C code)
Embedded Systems
Processor Features

- large degree of architectural variations
- short market life time
- real time constraints
- code size limitation
- can afford compilation time

Problems

- Lack of suitable design technology
  - compiler
  - simulator
- Compiler tool
  - standard compiler techniques in 70’s, 80’s-> low code quality
  - Normally no compiler support for ASIP’s
- Machine code design
  - low designer productivity
  - hard to debug
  - low code reuse
Research Interests and Development

• Architectural retargetability
  – compiler targeted for a wide range of processor architectures - architectural variation and short life time

• Code optimization
  – take advantage of processor architectural features: faster speed, smaller code size, lower power dissipation
  – Sophisticated techniques - can afford compilation time

Core Processor Types

• Off-the-shelf processors (micro-controller and DSP)
  – quick and reliable
  – low production volumes

• Application-specific instruction-set processors
  – customize core architecture and instruction set
  – high-volume consumer products

• Parameterizable processors
  – basic architecture, several versions
Processor Architecture Classification

• Retargetable compiler
  – based on architectural model
  – compiler generate efficient code if it fits model
• Classification scheme
  – characterize a given compiler: classes of architectures it can handle successfully
  – characterize a given processor: whether or not suitable compiler support can be found

Architecture Classification (1)

• Arithmetic specialization
• Data type
• Code type
• Instruction format
• Memory structure
• Register structure
• Control-flow capabilities
Architecture Classification (2)

- Arithmetic specialization
  - DSP:
    - parallel multiplier/accumulator unit -- correlation-like algorithms sped up (digital filters, auto and cross correlation)
  - ASIP:
    - Arithmetic specialization carried further: Critical sections of algorithms (e.g., deeply nested loops) executed in a minimal number of machine cycles

Architecture Classification (3)

- Data type
  - Typically fixed-point arithmetic
  - floating-point requires additional silicon area and dissipate more power
- Code type
  - Data-stationary
  - Time-stationary
Architecture Classification (4)

• Instruction format
  – Orthogonal
    • Fixed control fields set independently from each other
  – Encoded format
    • Interpretation of instruction bits as control fields deduced from designated bits in instruction words

Architecture Classification (5)

• Memory structure
  – Memory access
    • Von Neumann architecture: single memory access
    • Harvard architecture: separate memories access
  – Addressing modes
    • Direct, immediate, indirect addressing, indirect addressing
    • Indirect address: address post-modify
  – Operand location
    • Register-register
    • Memory-memory, memory-register
Architecture Classification (6)

• Register structure
  – Homogeneous register set
    • Register interchangeable
  – Heterogeneous register set
    • Specific registers for specific instructions
  – Register class
    • Number of register classes – measure for heterogeneity

Architecture Classification (7)

• Control flow
  – Small branch penalties
  – Zero loop overhead
  – Conditionally executable arithmetic instructions
  – Interrupt controller supports context saving mechanisms - hardware support context switching
Software Compilation Issue (1)
- Retargetability and high quality code

Software Compilation Techniques (1)
- Processor Specification Languages
- Processor Models for compilation
- Code selection
- Register Allocation
- Memory Allocation and Address Generation
- Scheduling
Software Compilation
Techniques (2)

• Processor specification languages
  – Netlist based languages: hardware building blocks
    • Data path, memories, instruction decoder, controller
    • MSSQ compiler: Mimola languages
  – High-level languages
    • Structural skeleton of the processor: declaration storage elements, data types
    • Description of actual instruction set

Software Compilation
Techniques (3)

• Processor Models
  – Template pattern bases
    • Patterns expressed in a grammar
      – terminals: operations
      – non-terminals: storage locations
  – Graph models
    • More readily represent structural information
    • MSSQ: connection-operation graph
    • RL: place-time graph
    • Chess: instruction-set graph (ISG)
Software Compilation Techniques (4)

• Code selection
  – Dynamic programming
    • tree structure
    • tree pattern matching
    • Tree covering
  – LR parsing
    • parse subject tree using specified grammar
  – Graph matching
    • DAG structure
    • DAG mapping

Software Compilation Techniques (5)

• Code selection (continued)
  – Bundling:
    • construct patterns on the fly
    • checked against processor model
Software Compilation Techniques (6)

- Code selection (continued)
  - Rule-Driven Code Selection:
    - virtual machine
    - set of rules to map operation patterns onto virtual machine

\[ *p++ = b + 1; \]

\[ \text{asgn RULE} \]
\[ \text{? matches($left, ind_postinc)} \]
\[ \text{STR_PINC $left, $right;} \]
\[ \text{...} \]
\[ \text{Binary_ADD_RULE} \]
\[ \text{? Matches($right, const)} \]
\[ \text{ADDI $dest, $left, $right;} \]
\[ \text{...} \]

\[ \text{LOAD R1, } _b \]
\[ \text{ADDI R1, R1, 1} \]
\[ \text{STR_PINC AX1, R1} \]

Software Compilation Techniques (7)

- Register Allocation
  - Graph coloring

<table>
<thead>
<tr>
<th>Value</th>
<th>Live range</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td></td>
</tr>
<tr>
<td>v3</td>
<td></td>
</tr>
<tr>
<td>v4</td>
<td></td>
</tr>
<tr>
<td>v5</td>
<td></td>
</tr>
<tr>
<td>v6</td>
<td></td>
</tr>
</tbody>
</table>

- Not homogenous: take register classes into account during graph coloring
- live range of values not known: interaction between register allocation and scheduling
Software Compilation Techniques (8)

- Register Allocation (continued)
  - Data routing
    - Strong heterogeneous register structure
    - More specialized register allocation techniques -- data routing
    - Selection of most appropriate route

Software Compilation Techniques (9)

- Memory Allocation and Address Generation
  - Allocation of data memory locations during memory spills in register allocation
  - passing argument values in function calls
  - Often stored in a stack
    - pointer post-modification in parallel with arithmetic operations
      - advantageous to have consecutively ordered memory accesses use adjacent memory locations
Software Compilation Techniques (10)

• Scheduling
  – High code quality: efficient use of architectural resources
  – Local scheduling
    • level of basic blocks: linear sequences of code without branching
  – Global scheduling
    • instructions moved across basic block boundaries (code motion)

Optimization for DSP/ASIP Compilers

• Two challenges to the DSP/ASIP compilers
  – Efficient Use Special Features of DSP/ASIP
    • irregular datapath
    • small number of registers
    • multiple data-memory banks
    • modes
  – Dense Code
• Optimization algorithm for the special DSP/ASIP architecture
Case 1: TMS320C25

  - Heuristic algorithm.
  - Simultaneously deal with instruction scheduling and register allocation.
  - Optimizing accumulator spilling and mode selection.

Basic Architecture of TMS320C25 (1)
Basic Architecture for TMS320C25 (2)

- ALU + Multiplier enables the machine to execute one-cycle multiply accumulate operations.
  - Good for digital signal processing
- No general-purpose registers other than the accumulator
  - T and P are not general purpose registers.

Basic Architecture for TMS320C25 (3)

- AGU can auto-increment or auto-decrement the content in the address registers.
- Some arithmetic operations are controlled by the sign-extension mode bit and product-shift mode bit.
  - Extra mode setting instruction may be needed in some cases.
Problem Formalization

- Mode optimization Problem for Scheduling
  - The goal is to schedule the instructions so that the number of mode-setting instructions is minimized.
- Register Allocation and Accumulator Spilling
  - Minimize memory spilling.

Mode Optimization (1)

- Represent this problem with a DAG $G = <V, E>$.
  - a vertex represents one instruction
  - mode value $l(v)$ associate with each vertex
  - min. $\#_{of\_mode\_setting}(S)$
- Don’t-Care Reduction
  - Not all instructions affected by the mode value. If not, assign don’t care - to that node.
  - For each path $v_1, v_2, ...v_{k-1}, v_k$ in $G$ where $l(v_i) = -$, for $1 < i < k$, and $l(v_i), l(v_k) \neq -$ add an edge $(v_1, v_k)$.
  - Remove each node that $l(v) = -$, and all edges attached to $v$. 
Mode Optimization (2)

- Two classes of mode: sign extension and product shift => two optimal schedule
- Merge the two schedules to one single optimal schedule for G if the two schedules are compatible.
  - Compatible means the generated schedule is valid.
  - If the two schedules are not compatible, then non-optimal reduced schedules will have to be chosen in order to achieve compatible.

Mode Optimization (3)

- Example

  sign-extension: s(ign) or u(nsing)

  product-shift: 0,1,2
Mode Optimization (4)

For sign extension optimization, best schedule is
S1 = \{ACBEF\} or
S2 = \{BACFE\}.
the two schedule both has
\#_of_mode_setting = 2

Mode Optimization (5)

For product shift optimization,
best schedule for this DAG
is S3 = \{ADFHEG\}, with
\#_of_mode_setting = 2
Mode Optimization (6)

- \(S_1 = \{ACBEF\} \& S_3 = \{ADFHEG\}\) are not compatible.
- \(S_2 = \{BACFE\} \& S_3 = \{ADFHEG\}\) are compatible. Merge these two schedules, we get an optimal schedule for the whole picture. \(S = \{BACDFHEG\}\)

Accumulator Spilling

- Only one accumulator for TMS320C25
- Minimum spilling-cost.
- \(\#_{\text{of spill}}(S) = \text{Number of accumulator spills in } S\)
Optimal Technique (1)

• Cost Function
  – \( C(S) = W_s \times \#_{of\_spill}(S) + W_m \times \#_{of\_mode\_setting}(S) \)
  – \( W_s \) and \( W_m \) depend on the relative cost of instructions required to spill the accumulator and instructions accomplish a mode switch

• Branch-and-Bound Algorithm
  – Lower Bound for Mode Setting
  – Lower Bound for Accumulator Spilling

Optimal Technique (2)

• Lower Bound for Spill Cost on a Partial Schedule
  – Nodes whose outputs are outputs of the basic block and which write into the accumulator will spill their contents.
  – If a node \( v \) has an output which has more than two fanouts, which means \( o(v) \) is used as an input in two other instructions, \( o(v) \) needs to be spilled.
  – If a node \( v \) receives inputs from nodes \( x \) and \( y \) which correspond to instructions that write into the accumulator, one input has to be spilled.
Optimal Technique (3)

• Lower Bound for Mode Cost
  – Simply compute the cost for mode switching along each path of $G'$.
  – Every schedule needs to somehow cover that path.
  – Choose the maximum cost.

Experiments and Results
Conclusion

• Present a scheduling algorithms that are able to exploit the features of the TMS320C25 microprocessor.
• Results indicate that these algorithms obtain substantial improvements in code size and performance over conventional code generation techniques.

Related Publication

• Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures; Guido Araujo and Sharad Malik.
  – Optimal Solution using tree covering.
Case 2: Multiple Data-Memory Banks

- *Memory Bank and Register Allocation in Software Synthesis for ASIPs*; by Ashok Sudarsanam and Sharad Malik
  - Post-pass optimization, after the instruction selection phase
  - Motorola 56000

Motorola 56000 Architecture (1)
Motorola 56000 Architecture (2)

- Dual Data-Memory Banks
  - 24-bit input registers: X0, X1, Y0, Y1
  - 56-bit accumulators: A, B
  - The dual-memory banks permits two memory accesses to occur in parallel.

Motorola 56000 Architecture (3)

- Address Generation Unit
  - two files of 16-bit address registers
  - {R0, R1, R2, R3} & {R4, R5, R6, R7}
  - one ALU for each file
  - one address generated from each file.
  - The Address from different file must point to different memory banks.
Motorola 56000 Architecture (4)

• It encodes parallel operations in a single 24-bit word.
• Up to two move operations can be specified in one instruction word.
  – 2 memory accesses
  – memory access + register transfer
  – register transfer + immediate load
• Most data ALU operations can be encoded in a single word with 1 or 2 move operations.

Motorola 56000 Architecture (5)

• Two parallel move operations are legal only when:
  – they access data in different bank
  – X data-memory => X0, X1, A, or B
  – Y data-memory => Y0, Y1, A or B
  – address registers belong to different AGU files.
Technique Overview

• Goal
  – symbolic reg => physical reg
  – variables => memory banks

• Step
  – Compaction
  – Constraint Graph Creation
  – Constraint Graph Labelling

Compaction

• Two move operations are compacted into a single instruction; one ALU operation is compacted with up to two move operations.
  – no data dependency
  – legal combination
Constraint Graph Creation (1)

- Vertex
  - Local Vertex
    - symbolic register => register-vertex
    - variable => variable-vertex
  - Global Vertex
    - X0, X1, Y0, Y1, A and B
  - label register-vertex with X0, X1, Y0, Y1, A or B
  - label variable-vertex with X-MEM or Y-MEM

Constraint Graph Creation (2)

- Edge
  - Each edge represents the constraints that need to be satisfied when labelling the vertices.
    - Red Edge
    - Green/Brown/Blue/Yellow Edge
    - Black Edge
    - Address Register Allocation
  - A cost/penalty associates with each edge.
Constraint Graph Creation (3)

• Red Edge: added between simultaneously alive register-vertices.
  – $\text{reg}_i$ and $\text{reg}_j$ should be labelled differently
  – cost: extra spill code

Constraint Graph Creation (4)

• Green Edge: added for each parallel move corresponding to a dual memory access.
  – $\text{var}_i$ and $\text{var}_j$ must be labelled differently
  – if $\text{var}_i$ is labelled X-Mem, $\text{reg}_i$ must be labelled X0, X1, A or B
  – if $\text{var}_j$ is labelled Y-Mem, $\text{reg}_j$ must be labelled Y0, Y1, A or B
Constraint Graph Creation (5)

- Blue Edge: added for each parallel move corresponding to a memory load and register transfer.
- Brown Edge: added for each parallel move corresponding to an immediate load and register transfer.
- Yellow Edge: added for each parallel move corresponding to a memory store and register transfer.
- Cost for green/blue/brown/yellow edges = 1

Constraint Graph Creation (6)

- Black Edge: added between a local register-vertex x and a global-vertex y, implies that x cannot be labelled with y.
- Hard Constraint: Labelling must satisfy this constraint, otherwise, hardware will not support it.
Constraint Graph Creation (7)

• MPY reg_i, reg_j, reg_k
  – regi and regj => input registers
  – regk => accumulator

Constraint Graph Creation (8)

• Address Register Allocation
  – Offset Assignment Problem
    • Goal: efficiently using the address registers
    • Approach: assign the two variable to adjacent memory location if they are frequently accessed together.
    • Offset Assignment Problem
Constraint Graph Creation (9)

- **Offset Assignment Problem**
  - generate the access sequence for a given code sequence
  - generate the access graph from the access sequence

- vertex x => variable x
- edge(x, y) with weight w => variable x and y adjacent to each other w times

```
c = a + b;
f = d + e;
a = a + d;
c = d + a;
d = d + f + a;
```

---

Constraint Graph Creation (10)

- **Offset Assignment Problem**
  - cost of an assignment = the sum of the weights of all edges connecting variables which are not assigned to adjacent memory locations.
  - To minimize the cost is equivalent to finding a maximum-weighted path covering of the access graph.
  - NP-hard problem
Constraint Graph Labelling

- Cost of a labelling = unsatisfied edge costs + offset assignment cost.
- Goal: To minimize the cost of a special labelling
  - complex cost function
  - large solution space
- Using simulated annealing to find a low-cost labelling.
  - easy to determine solution cost
  - easy to generate new solution

Results

- A program which accepts uncompacted, symbolic assembly code and generates high-quality compacted code.
- Be able to significantly reduce the code size of some DSP benchmark.
- Highlighted obstacles for embedded systems design
- Outlined main architectural features of contemporary DSP’s and ASIP’s
- Presented existing software compilation technologies for embedded processors
- Specific examples of optimization algorithms for some popular DSP.