

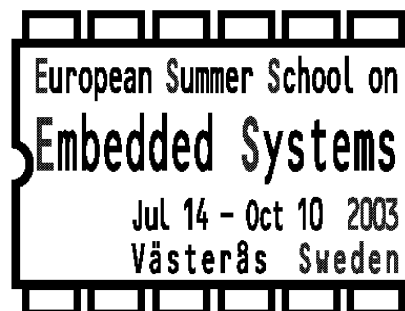
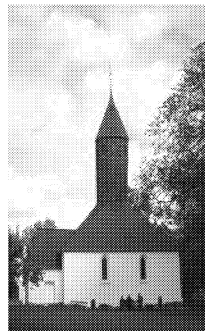
MÄLARDALENS HÖGSKOLA

ESSES 2003

European Summer School on Embedded Systems

Lecture Notes Part XIII

Embedded Systems: Embedded System Programming and Compiling



Editors: Ylva Boivie, Hans Hansson, Jane Kim, Sang Lyul Min

Västerås, August 25-29, 2003

ISSN 1404-3041

ISRN MDH-MRTC-108/2003-1-SE

MRTC

MÄLARDALEN REAL-TIME
RESEARCH CENTRE

www.mrtc.mdh.se

Flash Memory Basics

```
INFOR_HEADER_T  
  
IF (FM_Open() != FM_SUCCESS) {  
    return(FTL_MEDIAERR);  
}  
  
/* erase the whole device */  
for (current_block = 0; current_block < NO_OF_BLOCK; current_block++) {  
    FM_Erase(current_block);  
}  
...
```

Sang Lyul Min

School of Computer Science and Engineering
Seoul National University

2003. 8. 29

SNU Flash Team

Flash memory

Why flash memory ?

- Faster access
- Lower power
- Shock / Temperature resistance
- Smaller size
- Lighter weight
- Noiseless



SNU Flash Team

Test Framework

```
INFOR_HEADER_T *hp;  
  
if (FM_Open() != FM_SUCCESS) {  
    return(FTL_MEDIAERR);  
}  
  
/* erase the whole device */  
for (current_block = 0; current_block < NO_OF_BLOCK; current_block++) {  
    FM_Erase(current_block);  
}
```

Sang Lyul Min

School of Computer Science and Engineering
Seoul National University

2003. 8. 29

SNU Flash Team

Test Framework

Target: CompactFlash (CF)



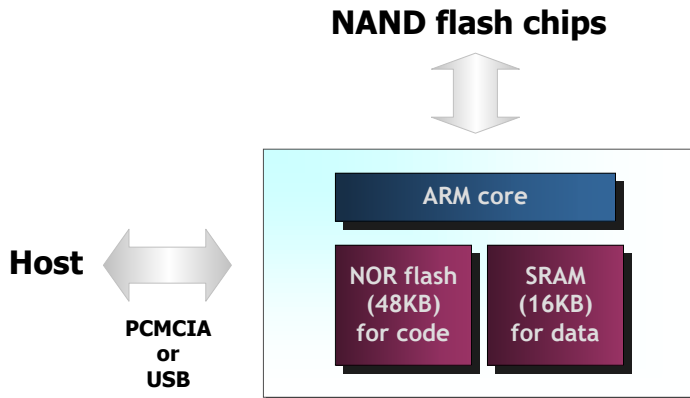
Front



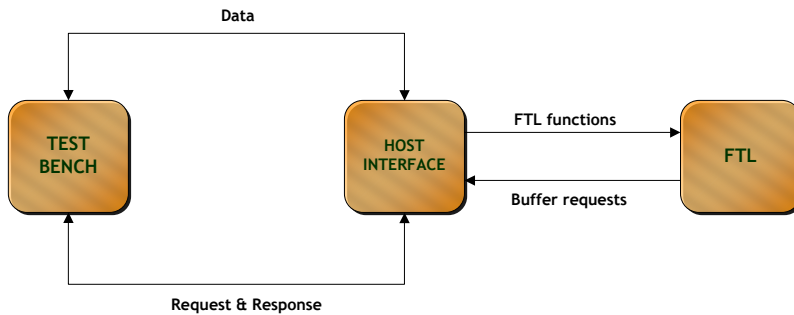
Back

SNU Flash Team

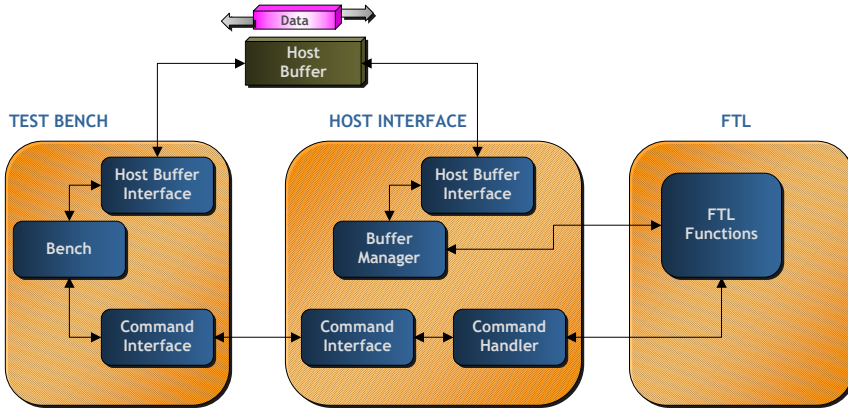
Controller (S3F49FAX)



Model

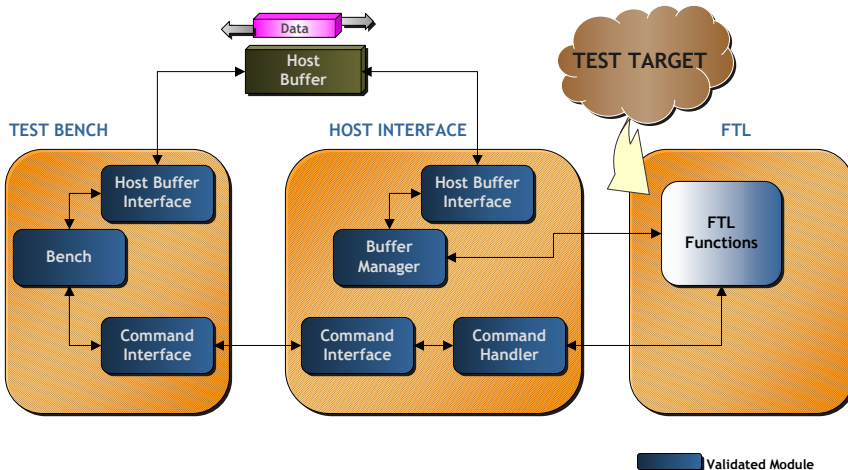


Detailed model



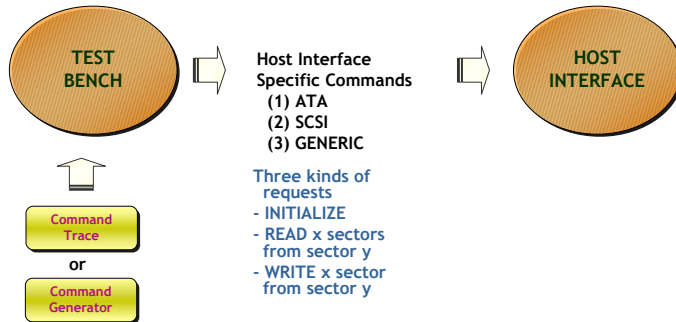
SNU Flash Team

Detailed model

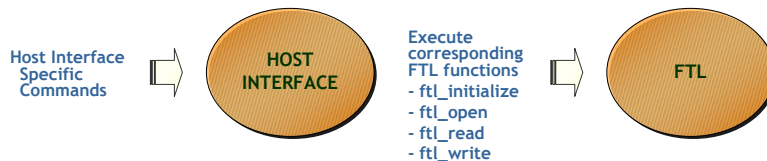


SNU Flash Team

Interaction between test bench and host interface



Interaction between host interface and FTL



Host interface module interprets host interface specific commands and invokes the corresponding FTL functions

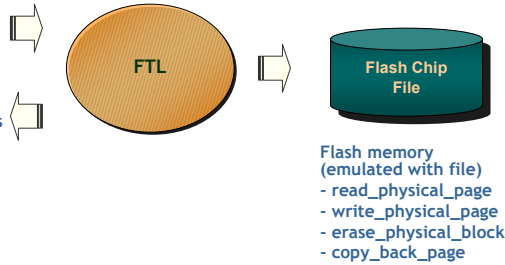
Interaction between FTL and flash chip emulation routine

FTL functions

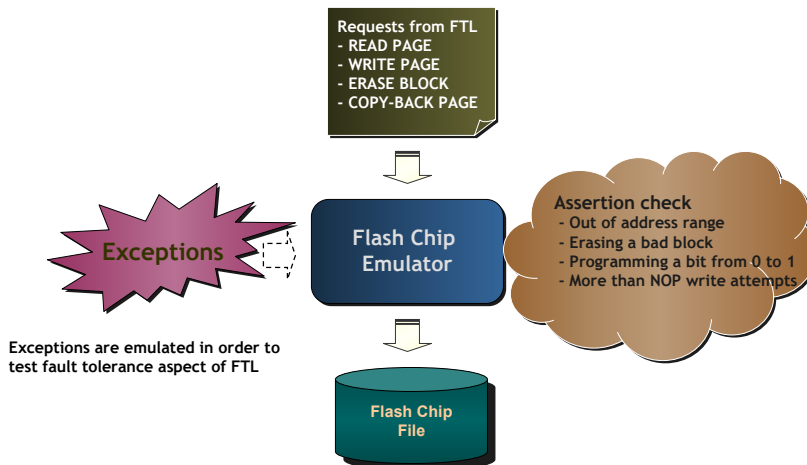
- ftl_initialize
- ftl_open
- ftl_read
- ftl_write

Buffer manager functions

- producer_get_buff,
- producer_unget_buff
- producer_advance,
- consumer_get_buff,
- consumer_unget_buff
- consumer_advance



Flash chip emulator



Emulated exceptions

- Exceptions in flash emulator
 - Power failure
 - Flash operation error (Read/Write/Erase/Copy-back)
- Exceptions in host interface
 - Corrupted data transfer
- Probabilities of different exceptions can be set by configuration file for each exception type

How to use

```

C:\WFLTestCore\Debug\WFLTest.exe
Command line start ...
1) help
Supported commands :
  INIT, OPEN,
  LREAD, LWRITE,
  FREAD, FWRITE, PERASE
  RANDRW, TRACE
  STREAM, CLS, EXIT
2) pread 0 0 0
000:01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :
010:11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00 :
020:19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00 :
030:21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00 :! " # $ % & ' (
040:29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00 :> = + , - . / 0
050:31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00 :1 2 3 4 5 6 7 8
060:39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00 :9 : ; < = > ? @
070:41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00 :a b c d e f g h
080:49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00 :i j k l m n o p
090:51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00 :q r s t u v w x
0a0:59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00 :y z { | } ^ _
0b0:61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00 :a b c d e f g h
0c0:69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00 :i j k l m n o p
0d0:71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00 :q r s t u v w x
  
```

- Available functions
 - Init Chip, Init FTL, Logical Read/Write, Physical Read/Write/Erase, Random Read/Write, Trace

Commands for initializing

- INITCIHP
 - Initialize flash chips
 - Parameter : num chips, num blocks per chip, bad block ratio
- INITFTL
 - Initialize FTL
 - Parameter : none

Commands mainly for primitive test purposes

- LREAD
 - Logical read
 - Parameter : start sector #, sector count
- LWRITE
 - Logical write
 - Parameter : start sector #, sector count, data

Commands for validation purposes

■ RANDRW

- Random Read/Write test
- Parameter : generated command count
- Related files
 - flash.exception_rate : specifies exception rates (read/write/erase error rates, power failure rates ...)
 - random_rw_test.config : specifies test features (backup period, read/write command ratio, debug mode setting...)

Config. files for random read/write test

■ flash.exception_rate

```
# read
read_error_rate          0.0
read_power_failure_rate  0.00005

# write
write_error_rate         0.0
write_power_failure_rate 0.00005
.....
```

■ random_rw_test.config

```
# number of backup files
retain_count            10

# read command generation ratio
read_command_ratio      0.5

# write command generation ratio
write_command_ratio     0.5
.....
```

Commands for debugging purposes

- PREAD
 - Physical read
 - Parameter : chip num, block num, page num
- PWRITE
 - Physical write
 - Parameter : chip num, block num, page num, data
- PERASE
 - Erase a block
 - Parameter : chip num, block num

Commands for performance evaluation

- TRACE
 - Trace-driven testing & performance evaluation
 - Parameter : trace file name
 - Results :
 - Number of flash memory operations
(Read, Write, Erase, Copy-Back)

Vanilla FTL

```
INFOR_HEADER_T *hp;  
  
if (FM_Open() != FM_SUCCESS) {  
    return(FTL_MEDIAERR);  
}  
  
/* erase the block groups */  
for (current_block = 0; current_block < NO_OF_BLOCK; current_block++) {  
    FM_Erase(current_block);  
}
```

Sang Lyul Min

School of Computer Science and Engineering
Seoul National University

2003. 8. 29

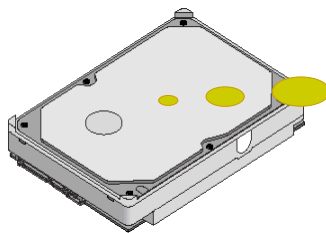
SNU Flash Team

Vanilla FTL

FTL (Flash Translation Layer)

■ Definition

- Software layer that makes flash memory appear to the system like a disk drive



Controller
(running FTL)
with NAND
flashes

SNU Flash Team

Logical interface for a disk drive



■ Operations

1. Identify drive(): returns N
2. Read sectors(start sector #, # of sectors)
3. Write sectors(start sector #, # of sectors)

Vanilla FTL

■ Barebone FTL implementation

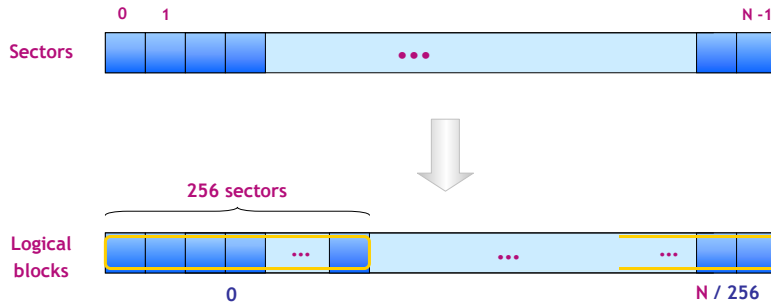
- Embarrassingly simple
- No error recovery
- No wear leveling
- No, nothing...

■ Supported flash memory chips

- 2nd generation flash memory chips
 - Page size : 2KB, block size : 64 pages (128KB)

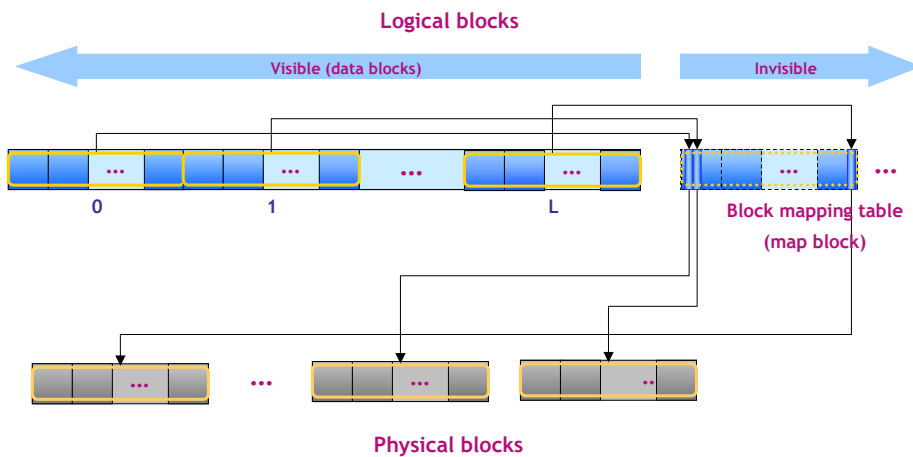
Block level mapping

- Logical blocks



Block level mapping

- Logical to physical block mapping

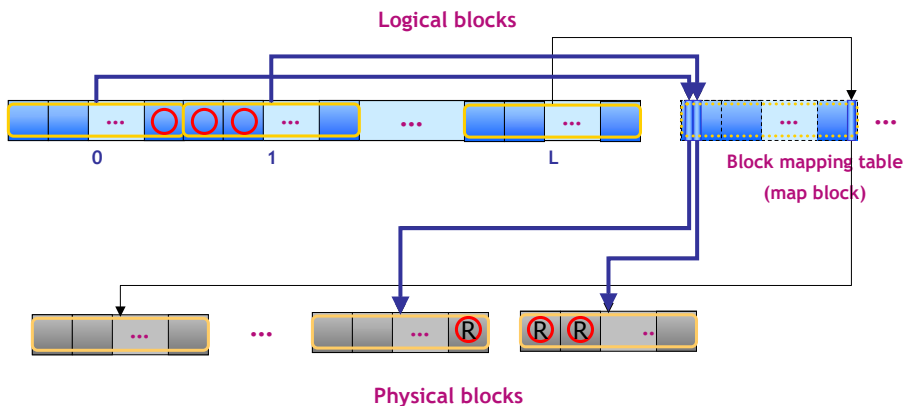


Block level mapping

- How many map blocks are needed?
 - Block addresses
 - 2 byte unsigned integer
 - 256 map entries in a sector (512B)
 - One map block
 - $256 \text{ (# of map entries/sector)} \times 256 \text{ (sectors/block)} \times 128\text{KB (block size)} = 8\text{GB}$
- One map block can cover 8GB of storage!

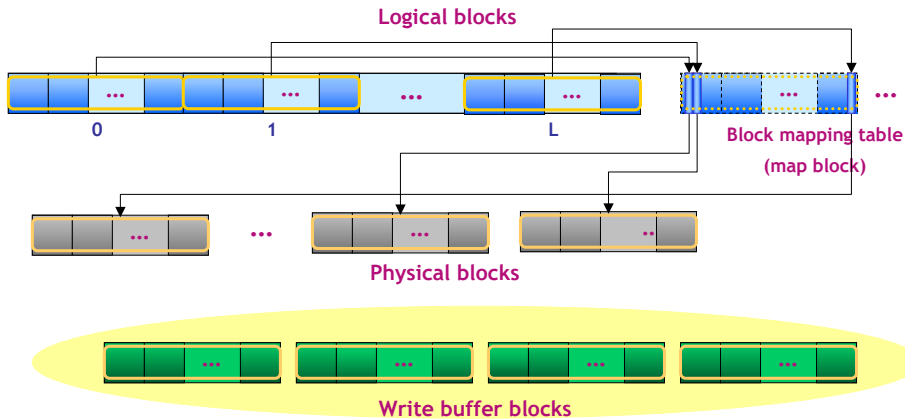
Read procedure

- Ex. read 3 sectors from 255



Write buffer blocks

- Reserved blocks for write processing

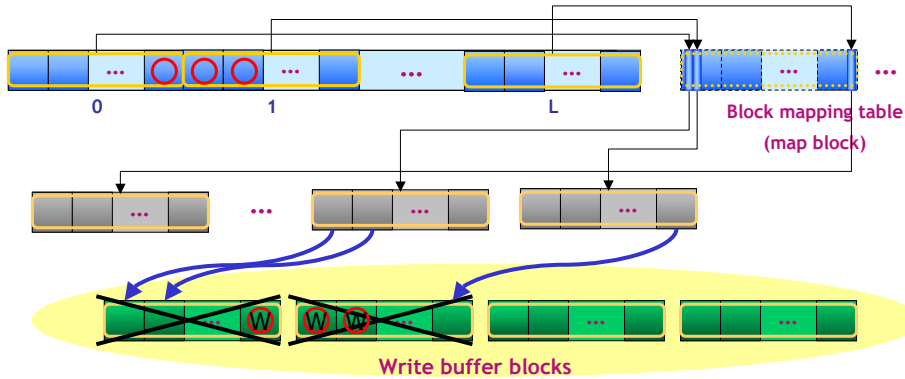


Write procedure

1. Erase write buffer blocks for data
2. Write new data pages to write buffer blocks
3. Fill in remaining data pages
4. Erase write buffer blocks for map
5. Read-modify-write map page (update map entry for data blocks)
6. Fill in remaining map pages

Write procedure

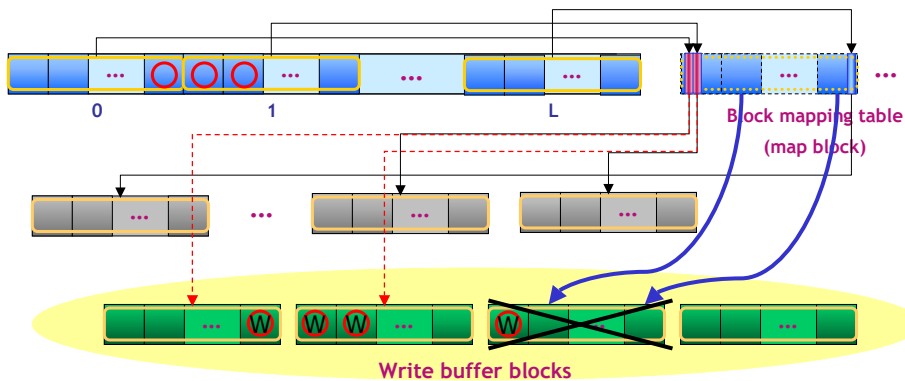
- Ex. write 3 sectors from 255



1. Fill remaining data pages at a

Write procedure

- Ex. write 3 sectors from 255



6. Fill remaining map pages

Checkpoint

- To make changes permanent
 - Written to flash after each write operation is completed
- Power down & recovery
 - Power down before checkpoint write
 - Recovered as if no write operation performed (no sectors are written)
 - Guarantee atomicity
 - Either all of sectors are written, or no sectors are written

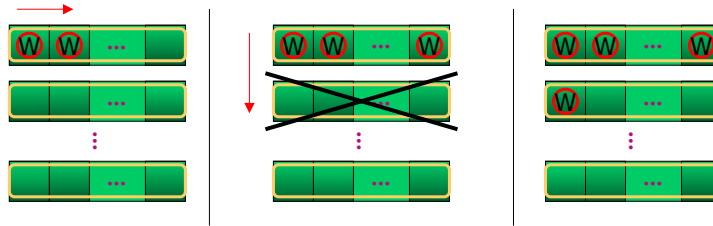
Checkpoint

- In-memory data structure changed with write operation
 - Write buffer block addresses
 - Map block addresses
- Use of pages in checkpoint blocks
 - Round-robin
 - Need timestamp to locate the page containing the most recent checkpoint at the recovery time

Checkpoint blocks

■ Checkpointing procedure

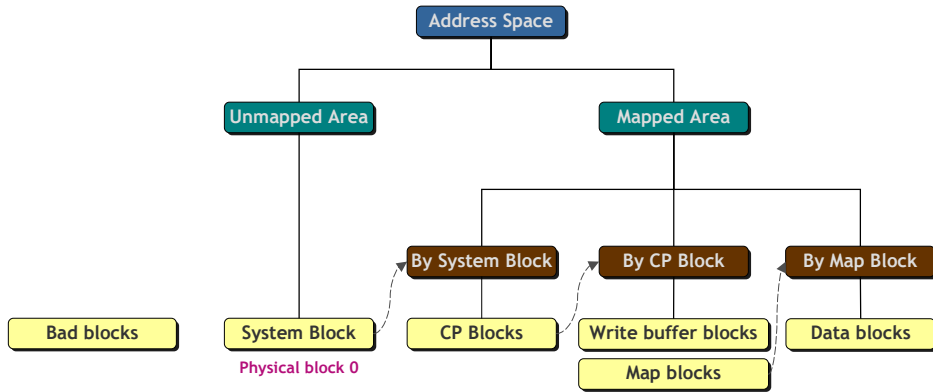
1. Increment timestamp
2. Write checkpoint data to the next page (in the round-robin order)
3. If the next page is in a new block, erase the block before write



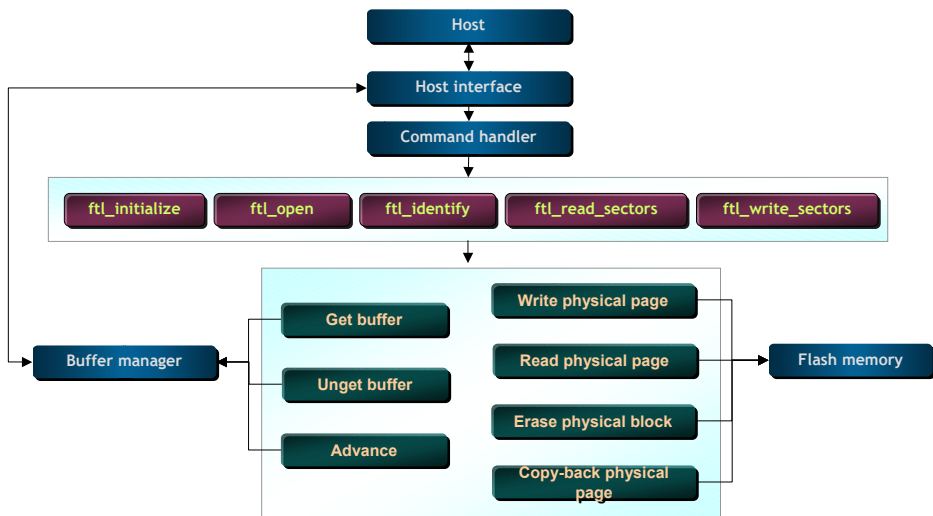
Permanent data

- Data that is never changed once FTL is initialized
 - Flash chip configuration info
 - Number of chips
 - Number of blocks in a chip
 - Checkpoint block info
 - Number of checkpoint blocks
 - Physical addresses of checkpoint blocks
 - Information about logical address space
- Stored in the system block

Block arrangement



Interfaces between modules



FTL interface functions

```
uint8 ftl_initialize(init_parameter_t *init_parameter_p);  
    □ called only once at format time
```

```
uint8 ftl_open(void);  
    □ called on power-on
```

```
ftl_info_t* ftl_identify(void);
```

```
uint8 ftl_read_sectors(uint32 start_sector_num,  
                      uint16 sector_count,  
                      uint16 *processed_sector_count,  
                      uint8 read_mode);
```

```
uint8 ftl_write_sectors(uint32 start_sector_num,  
                      uint16 sector_count,  
                      uint16 *processed_sector_count,  
                      uint8 write_mode);
```

FTL interface functions

```
struct init_parameter {  
    flash_chip_info_t flash_chip_info; // flash chip configuration  
};  
typedef struct init_parameter init_parameter_t;  
  
struct ftl_info {  
    uint32 num_total_sectors;  
};  
typedef struct ftl_info ftl_info_t;
```

Buffer manager

■ Structure of single buffer

```
struct buffer_info {  
    uint8 is_valid;  
    uint8 error_code;  
    uint32 l_page_addr;           // logical physical address of buffer data  
    uint8 *buffer_p;             // buffer pointer  
};  
typedef struct buffer_info buffer_info_t;
```

Buffer manager

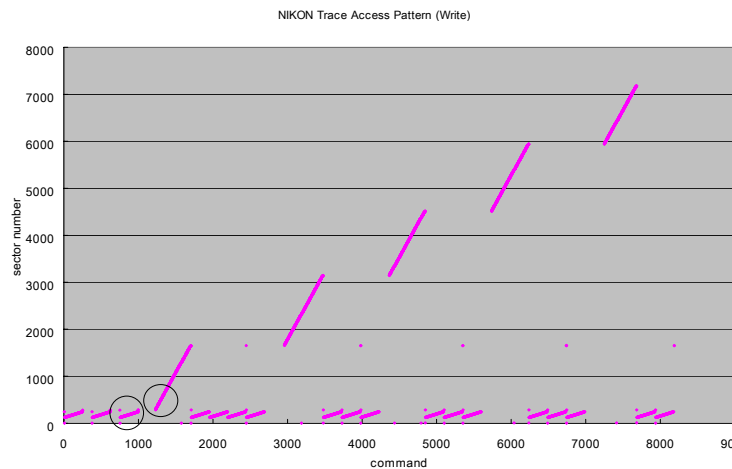
■ Buffer manager interface functions

```
buffer_info_t* buffer_manager_producer_get_buffer(void);  
void buffer_manager_producer_unget_buffers(uint8 num_buffers_to_unget);  
void buffer_manager_producer_advance(uint8 num_buffers_to_advance);  
  
buffer_info_t* buffer_manager_consumer_get_buffer(void);  
void buffer_manager_consumer_unget_buffers(uint8 num_buffers_to_unget);  
void buffer_manager_consumer_advance(uint8 num_buffers_to_advance);
```

Possible project topics

1. Wear-leveling for checkpoint blocks and possibly for data blocks and map blocks
2. Recovery from
 - Physical block erase errors
 - Physical page write errors
 - Physical page read errors
3. Advanced write buffering for
 - Large sequential writes (data writes)
 - Small random writes within a block (FAT writes)
4. Caching for map pages

Typical write access pattern



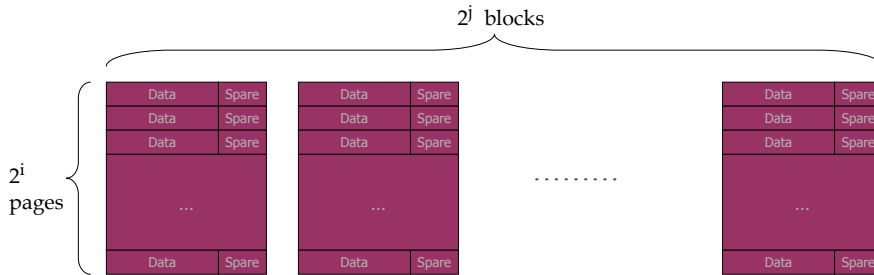
Then why still limited deployment ?

- Higher cost
 - Flash memory: 25 ¢/MB
 - Hard disk: 0.83 ¢/MB
 - CD-ROM: 0.07 ¢/MB
- Limited write performance
 - Flash memory: 4~5 MB/s
 - Hard disk: 20~30 MB/s

Different flash memory types

| Type | NOR | NAND | DINOR | AND |
|---------------------|---|--|--------------------------------------|-----------------------|
| Product | Intel 28F128J3A-150 28F320D18B110 AMD Am29LV641DU Am29DL322D | Samsung K9F5608UOM Toshiba TH58512FTI AMD Am30LV0064D | Mitsubishi M5M29GB/ T160BVP-80 | Hitachi HN29W25611 |
| Density | Low | High | Low | High |
| Erase Block size | Large (64 ~ 128KB) | Large (16 ~ 128KB) | Large (32 ~ 64KB) | Small (2KB) |
| Page size | 1, 2, 8, 32B | 512B, 2KB | 1, 2, 256B | 2KB |
| Capacity | Low | High | Low | High |

NAND flash memory chip (big picture)



☞ **Note 1**

$i = 5$, Data = 512B, Spare = 16B, NOP = 1 for first-generation NAND flash

$i = 6$, Data = 2KB, Spare = 64B, NOP = 4 for second-generation NAND flash

j depends on chip capacity

☞ **Note 2**

Some blocks are bad at the manufacturing time that are marked in the spare area, but block 0 is guaranteed to be good

☞ **Note 3**

There is an upper limit on the maximum number of erases allowed for each block

Basic flash memory operations

- Read physical page
 - (chip #, block #, page #)
 - ~45 us
- Write physical page
 - (chip #, block #, page #)
 - ~325 us
- Erase block
 - (block #)
 - ~2 ms
- Copy-back
 - (chip #, target block #, target page#, source block #, source page #)
 - ~320 us

Flash memory interface routines

- read_physical_page
- write_physical_page
- erase_physical_block
- copy_back_physical_page

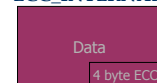
Read physical page

```
read_physical_page(chip_num, p_block_addr_in_chip, p_page_num,
                  offset, size, *buff_p, ecc_mode, *ecc_p);
```

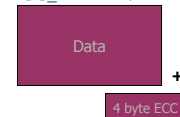
PARAMETERS

- chip_num : chip number
- p_block_addr : physical block address
- p_page_num : page number in block (0 ~ 63)
- offset : offset in page (0 ~ 2111, most cases 0, 512, 1024, or 1536)
- size : data size to be read from offset (most cases 512)
- buff_p : buffer pointer for read data
- ecc_mode : ECC_NONE, ECC_INTERNAL, ECC_EXTERNAL
(Use ECC_NONE for now)
- ecc_p : ecc for ECC_EXTERNAL (4B)

ECC_INTERNAL



ECC_EXTERNAL



Write physical page

```
write_physical_page(chip_num, p_block_addr, p_page_num,  
                    offset, unit_size, *buff_p[], ecc_mode,  
                    ecc_p[][ECC_SIZE]);
```

PARAMETERS

- chip_num : chip number
- p_block_addr : physical block address
- p_page_num : page number in block (0 - 63)
- offset : offset in page (0 - 2111)
- unit_size : data size in each buffer to be written (most cases 512)
- buff_p : array of buffer pointers (unused buffer = NULL)
- ecc_mode : ECC_NONE, ECC_INTERNAL, ECC_EXTERNAL
(Use ECC_NONE for now)
- ecc_p : ecc for ECC_EXTERNAL (4B)

Erase physical block

```
erase_physical_block(chip_num, p_block_addr);
```

PARAMETERS

- chip_num : chip number
- p_block_addr : physical block address

Copy-back physical page

```
copy_back_physical_page(chip_num,  
                        p_target_block_addr, p_target_page_num,  
                        p_source_block_addr,  
                        p_source_page_num);
```

PARAMETERS

- chip_num : chip number
- p_target_block_addr : physical address of target block
- p_target_page_num : page number in target block (0 ~ 63)
- p_source_block_addr : physical address of source block
- p_source_page_num : page number in source block (0 ~ 63)

Jakob Engblom

Uppsala University & Virtutech
jakob.engblom@it.uu.se
jakob@virtutech.com

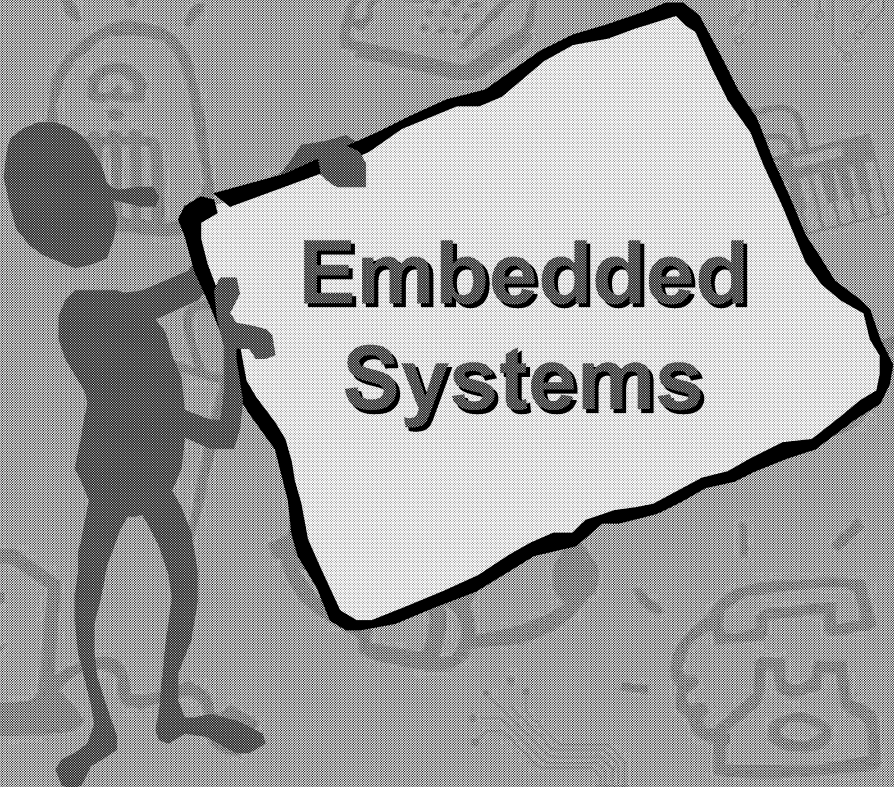
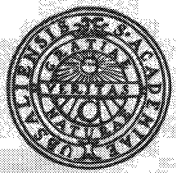
Embedded Systems Computer Architecture

Jakob Engblom, PhD

Uppsala University & Virtutech Inc.

jakob.engblom@it.uu.se
jakob@virtutech.com

virtutech

A black stick figure is shown from the side, holding a large, irregular white sign with a thick black border. The sign contains the text 'Embedded Systems'. The background is a light gray with faint, stylized icons of various electronic components like a calculator, a clock, a computer monitor, and a circuit board.

**Embedded
Systems**

Embedded Systems

Now what is this elephant thing?

It is a snake!

No, a wall!

You're all wrong, it is a fan!

No, it is a tree trunk!

No, a pillar!

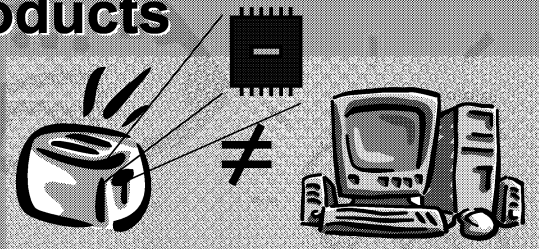
29 November 2002

Embedded Computer Architecture

3

Embedded Systems

- ✱ "A computer that doesn't look like a computer"
- ✱ Interacts with world
- ✱ Primitive or no user interface
- ✱ Part of other products



29 November 2002

Embedded Computer Architecture

4

Embedded Systems

*Single purpose products

- ◆ Not *general purpose* like desktop PCs
- ◆ Do one thing very efficiently

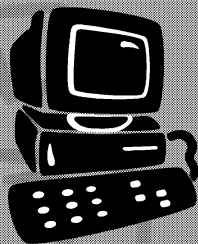
*Software very important:

- ◆ Gives character to product
 - Used to differentiate inside a “platform”
- ◆ Can be changed late
- ◆ Processor cheaper than special HW
- ◆ Today, dominates dev cost

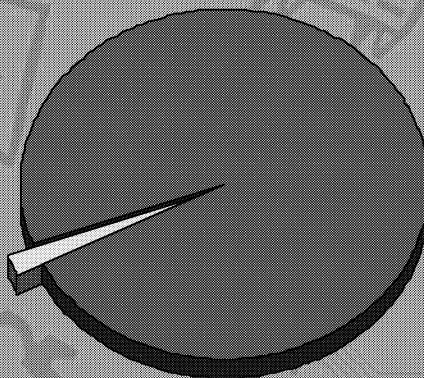
Processor Market

*Embedded = most processors!

- ◆ 200 million PC and server
- ◆ 8000 million embedded



"Desktop"
2%



"Embedded"
98%

Processor Market

* Processors:

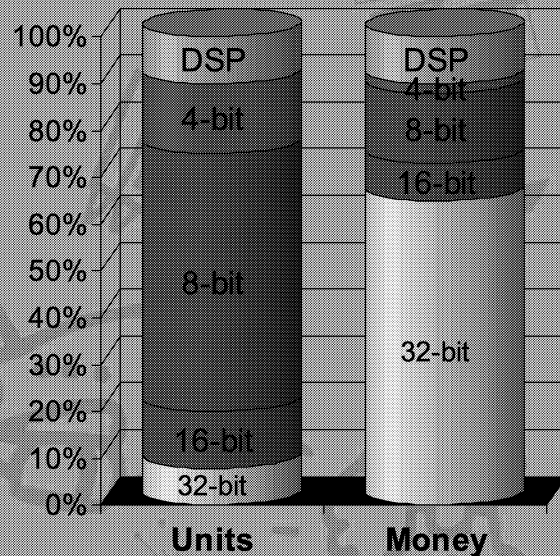
- ◆ 50% of all semiconductor revenue
- ◆ Explains why everyone wants to do processors

* 32-bit dominant

- ◆ 30% of total semiconductors

* PC processors:

- ◆ 50% of CPU revenue
- ◆ 15% of total semiconductors
- ◆ AMD and Intel share it



Real-Time System

* Timing as important as result

* Hard real-time:

- ◆ Hard deadlines
- ◆ Dead if missed deadline
- ◆ Worst-case

* Soft real-time:

- ◆ Fuzzier deadlines
- ◆ Can miss some deadlines
- ◆ Average-case



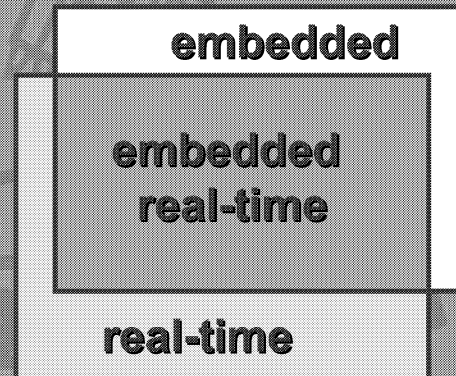
Real-Time Systems

*Embedded and Real-Time

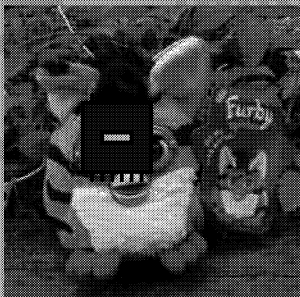
◆Synonymous?

*Most embedded systems are real-time

*Most real-time systems are embedded



Simple Embedded Systems



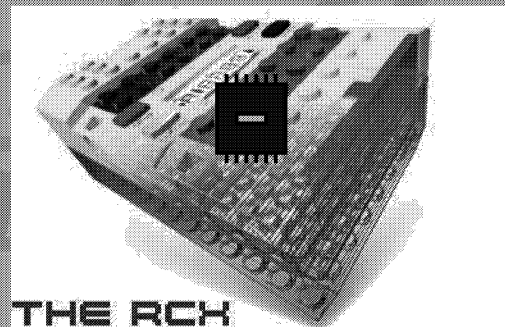
8-bit Intel 8051,
standard microcontroller

Behavior, talk,
IR communications

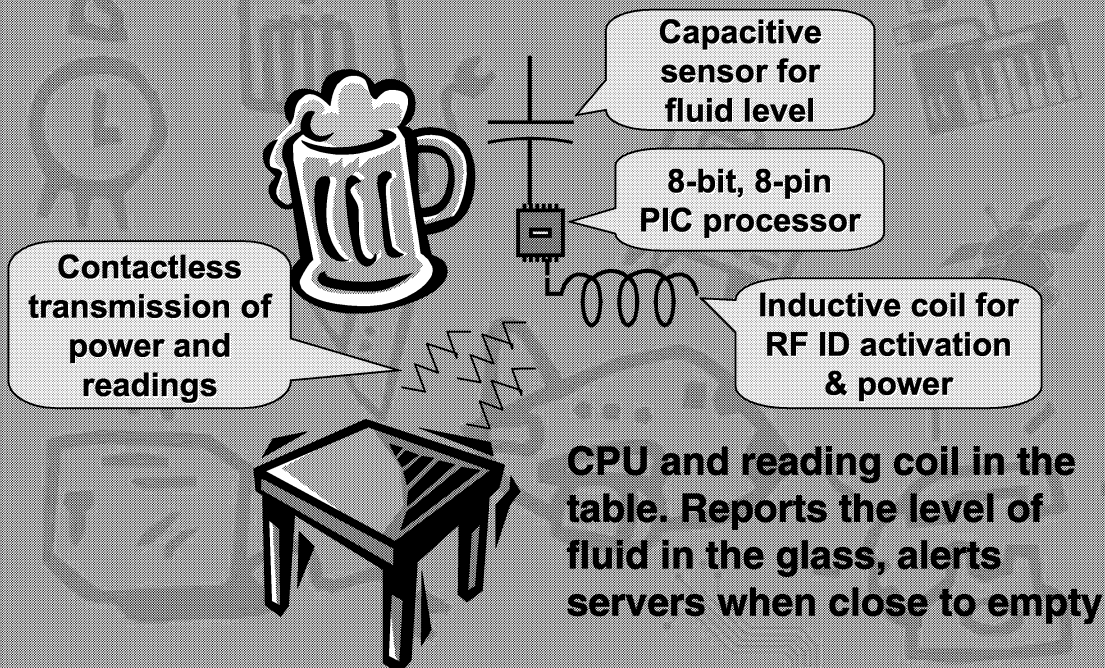
8-bit Hitachi H8/300
32 kB ROM, 32 kB RAM

Standard microcontroller chip

Byte-code machine,
sensor drivers, ...



Fun App: Smart Beer Glass



29 November 2002

Embedded Computer Architecture

11

No Upgrades Possible

- *Once a product ships...
- *...it often cannot be serviced
 - ◆No download ability
 - ◆No writable persistent storage
 - ◆No disks
 - ◆No loader
- *Software is write-once
- *(There are exceptions)

29 November 2002

Embedded Computer Architecture

12

Consumer Electronics



* Heterogeneous multiprocessor

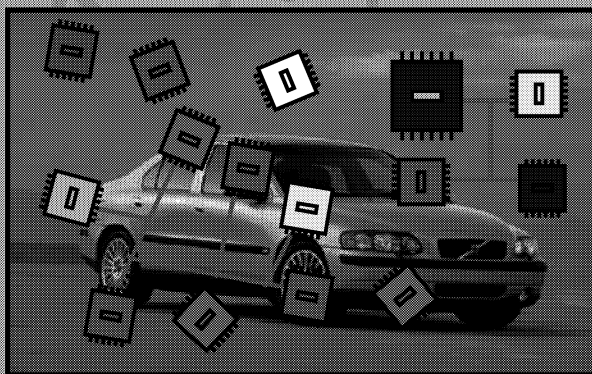
- ◆ 8-bit Atmel AVR for UI, games, ...
- ◆ 16-bit fixed-point TI C54 DSP for GSM coding, radio interface, ...
- ◆ 32-bit ARM7 in Bluetooth module
- ◆ + maybe ARM7 in IRDA interface

* All in custom chips

* Software is large:

- ◆ 16 MB of code in control part
- ◆ Plus signal processing code

Automotive



* Multiple networks

- ◆ CAN for body electronics: 30+ nodes
- ◆ CAN for engine control: few nodes
- ◆ LIN for instruments

* Many processors

- ◆ Up to 100

* Large diversity in processor types:

- ◆ 8-bit CPUs (PIC, HC08) for door locks, lights, etc.
- ◆ 16-bit CPUs (C167, HC11, HC12) for most functions
- ◆ 32-bit CPUs (PPC, V850) for engine control, airbags

* Total amount of code: 40-50 MB

Automotive

*Form follows function

- ◆ Processing where the action is
- ◆ Architecture given by application
- ◆ Sensors and actuators distributed

*Heterogeneous systems

- ◆ Many different makes of CPUs
- ◆ Standardized at the network/bus

Timing Aspects

*Interrupt latency

- ◆ Important criterion for embedded
- ◆ A few clock cycles at most
- ◆ Measure of RTOS performance

*Real-Time = predictability

- ◆ In-order pipelines
- ◆ SRAM instead of caches
- ◆ Lockable caches
- ◆ Several small CPUs instead of one big

Military Shipboard



**Standard multiprocessor
UltraSparc servers for
radar, target tracking,
combat control, ...**

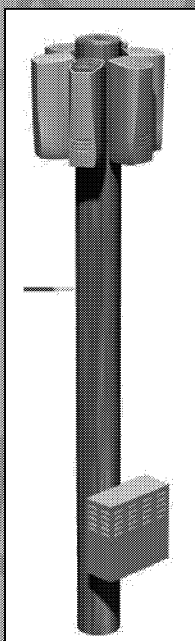
**Many CPUs in missiles,
gun controls, engines, ...**

29 November 2002

Embedded Computer Architecture

17

Mobile Phone Base Station



★Handle signals

- ◆Data streams to and from phones
- ◆Massively parallel system
- ◆Thousands of DSP tasks
- ◆Perfect parallel scalability

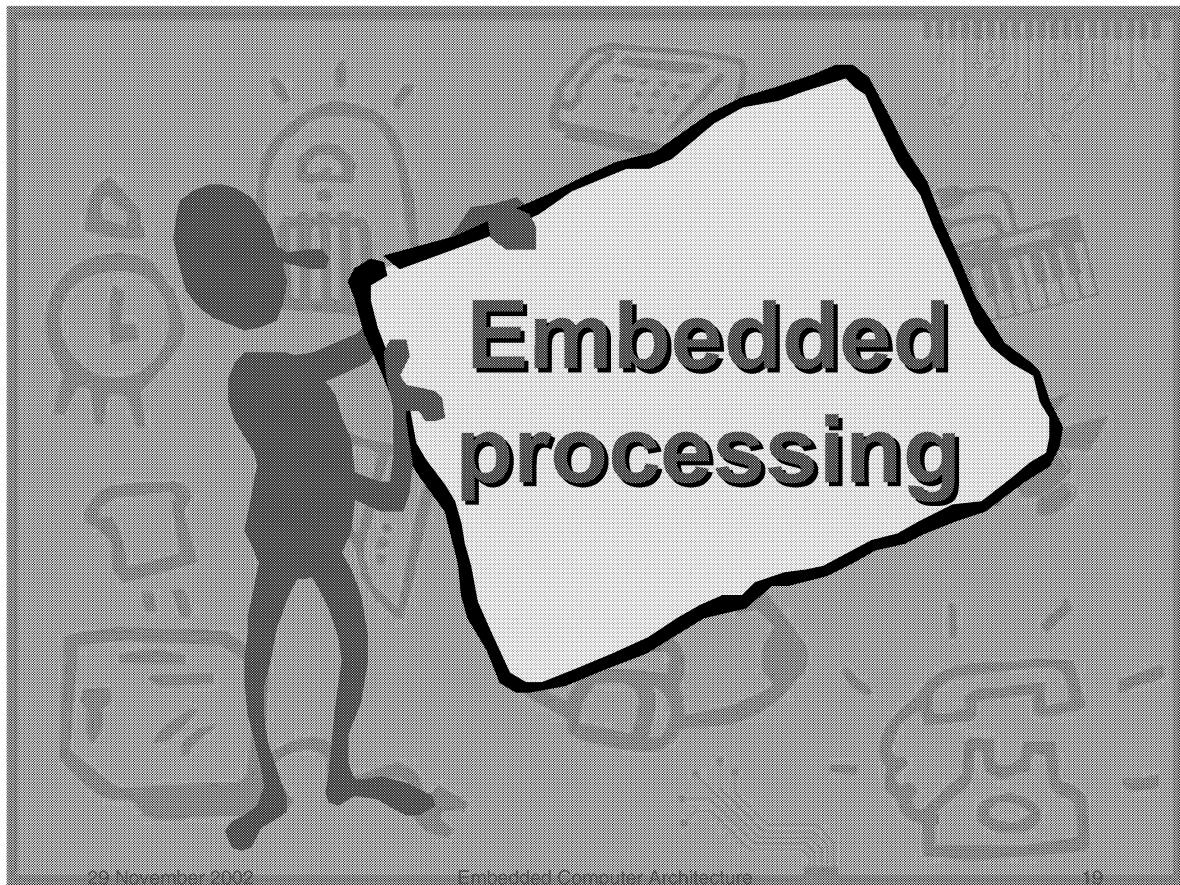
★Custom or standard DSPs

- ◆Up to 8 DSPs on a single chip

29 November 2002

Embedded Computer Architecture

18



Integration

- ✱ **A single chip:**
 - ◆ CPU Core
 - ◆ Integrated memory
 - ◆ Integrated peripherals
 - ◆ Integrated services
- ✱ **Goal:**
 - ◆ System on one chip
 - ◆ No external HW
 - ◆ Fit application “perfectly”

| | | | |
|-------------|-----|-----------|-------|
| RAM (small) | | ROM (big) | |
| CPU Core | | | |
| UART | A/D | LCD D | Timer |

Outside World

29 November 2002 Embedded Computer Architecture 20

Processors: Wide Span

- * **Bitwidths: 4 to 64 bits**
 - ◆ Most common: 8 bit (4G units)
 - ◆ 32-bit growing fastest
 - ◆ 32/64-bit outnumbers desktop
- * **Frequency: DC to 2 Ghz**
- * **Memory: From 0.5 kB to 5 MB**
- * **Power: mW (and up)**
- * **1/30 to 10 instructions per cycle**

Devices on the Chip

- * **Interface with the world**
 - ◆ Digital I/O
 - ◆ Analog/Digital conversion
 - ◆ Digital/Analog conversion
- * **Communications**
 - ◆ CAN networks
 - ◆ Ethernet networks
 - ◆ Radio
 - ◆ Serial ports (UART, USART)
 - ◆ USB, FireWire, ...

Devices on the Chip

*Timers

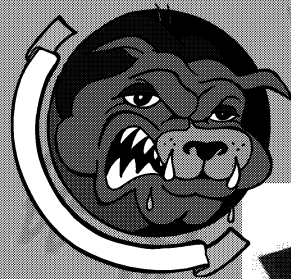
- ◆ Trigger interrupts
- ◆ Watchdogs

*Graphics

- ◆ LCD drivers
- ◆ 2D/3D graphics acceleration

*Buses

- ◆ On-chip: between devices: AMBA, ...
- ◆ Off-chip: PCI, HyperTransport, RapidIO ...



Trends

*Market

- ◆ 32-bit market is growing fast
- ◆ DSPs are growing fast

*Technology

- ◆ Configurable processors
- ◆ Configurable logic as help
- ◆ Fusion of DSP and microcontrollers
- ◆ More complex architectures
- ◆ Higher integration on each chip
- ◆ Multiprocessors on-a-chip

Trends

*Hardware to software

- ◆ Increase flexibility, lower cost
- ◆ Software on fast processor can equal HW

*Software to hardware

- ◆ Better power consumption & performance
- ◆ Design custom hardware for application

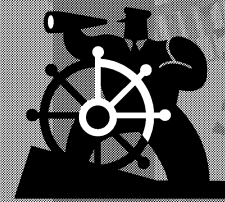
*Hardware-software codesign

- ◆ Delay division HW/SW to late in project
- ◆ Obtain “optimal” HW/SW division

Control vs Data

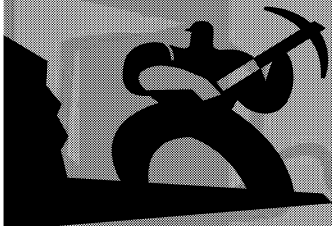
*Control plane:

- ◆ Microcontrollers
- ◆ Decision-making
- ◆ “Integer applications”
- ◆ UI of a phone, packet routing, ...



*Data plane:

- ◆ Move or process data
- ◆ Performance is key
- ◆ Signal processing, multimedia, ...
- ◆ Floating/fixed point



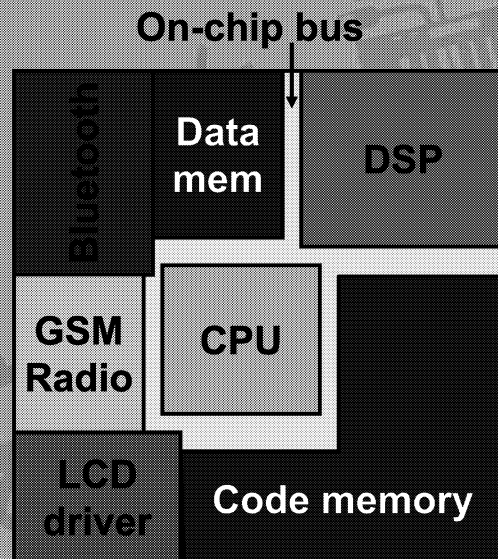
System-on-a-chip

*Integration extreme

- ◆ Thanks to modern semiconductors

*Entire product on a chip

*One or more processors, accelerators, ...



Packaging

Packaging of Processing

- ✱ **Microprocessor**

- ◆ Standard stand-alone processor chip

- ✱ **Microcontroller**

- ◆ Processor plus devices

- ✱ **ASIP**

- ◆ Application-Specific Integrated Processor

- ✱ **ASIC**

- ◆ Application-Specific Integrated Circuit

- ✱ **FPGA**

- ◆ Field-Programmable Gate Array

Microcontrollers

- ✱ **Classic embedded hardware**

- ✱ **Standard parts**

- ◆ Quite broad application domains
- ◆ Sold in large series
- ◆ Defined by hardware vendors
- ◆ As cheap as a single dollar

- ✱ **Single processor + devices**

- ✱ **Huge number of variants**

- ✱ **Usually intended for control plane**

Microcontrollers

Example: PIC 12CE674

- * **Memory arch:** Harvard
- * **Program memory:** 2048 x 14 (OTP/Flash)
- * **EEPROM:** 16 bytes
- * **RAM:** 128 bytes
- * **ADC channels:** 4 (8 bits)
- * **I/O ports:** 6
- * **Timers:** One 8-bit, One WDT
- * **Clock:** onchip crystal, 10MHz
- * **Package:** 8 pins (Pentium 4:700 pins)
- * **Cost:** <\$1.00 (Pentium 4:>\$200.00)

Example: AT91M42800A

- * **ARM7TDMI 32-bit core**
 - ◆ Static design: 0 to 33 Mhz
- * **Memory**
 - ◆ 8 kB SRAM on chip
 - ◆ External memory interface, 8/16 bit interface
- * **Devices**
 - ◆ 6 timers
 - ◆ 2 serial ports
- * **JTAG debug interface**
- * **About 0.5 W power**
- * **About 18 USD**
- * **144 Pin package**
- * **One of 13 AT91 variants**



ASIPs / ASSPs

* Application-specific integrated/standard processor

- ◆ Targeting a particular niche market
- ◆ More targeted than microcontroller
- ◆ Domain-specific accelerators

* Usually more upscale

- ◆ 32-bit processors
- ◆ Multiprocessors
- ◆ Expensive peripherals
- ◆ External memory assumed
- ◆ Higher performance, includes data-plane

ASIP / ASSP

Example: PowerQUICC III

* Motorola

* Target market

- ◆ Communications

* Processing

- ◆ PowerPC e500
- ◆ 666-1000 Mhz
- ◆ 256 kB L2 cache

* Networking

- ◆ CPM module, RISC-based microcode

* About 160 USD

Features

| | |
|--|---|
| Serial Communications Controller (SCC) | 4 |
| Fast Communications Controller (FCC) | 3 |
| Multi-Channel Controller (MCC2) | 2 |
| Serial Management Controller (SMC) | 2 |
| Serial Peripheral Interface (SPI) | 1 |
| I2C controller | 1 |
| DDR Memory controller | 1 |
| PCI-X/PCI controller | 1 |
| RapidIO controller | 1 |
| Ethernet 10/100/1000 controller | 2 |

Capabilities

| | |
|-------------------------------|-----|
| Ethernet, 10 (from SCC) | 4 |
| Ethernet, 10/100 (from FCC) | 3 |
| Ethernet 10/100/1000 | 2 |
| Utopia II ATM (from FCC) | 2 |
| Multichannel HDLC (from MCC2) | 256 |

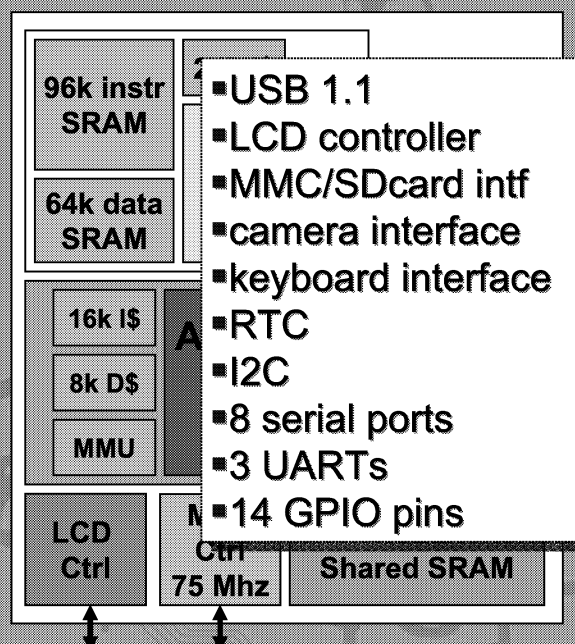
Example: C167CS

- * Infineon
- * Target Market
 - ◆ Automotive control
- * Processing
 - ◆ 16-bit C16x core
 - ◆ 4-stage simple pipeline
 - ◆ 40 Mhz operation
 - ◆ 16 MB memory space, including ROM, RAM, devices
- * 144 pin package
 - ◆ Tolerates -40 C to +125 C
- * About 25 USD

| Devices | |
|-----------------------------------|-------|
| CAN 2.0b controllers | 2 |
| General-Purpose Timers (GPT) | 5 |
| Watch-Dog Timer (WDT) | 1 |
| Pulse-Width Modulator (PWM) | 1 |
| Analog-Digital Converter Channels | 24+8 |
| USART | 1 |
| Synchronous Serial Comms (SSC) | 1 |
| Capture/Compare Channels | 2x16 |
| External Ports | |
| CAN interfaces | 2 |
| 8-bit ports from devices | 8 |
| 16-bit ports from devices | 1 |
| Memory | |
| ROM | 32 kB |
| Fast General Internal RAM (IRAM) | 3 kB |
| Extension Internal RAM (XRAM) | 8 kB |

Example: TI OMAP 5910

- * Texas Instruments
- * Target market
 - ◆ Data-intensive real-time
 - ◆ Audio, biometrics, etc.
- * Processing
 - ◆ Dual-core chip
 - ◆ ARM925T 150 Mhz
 - ◆ TI C55 DSP 150 Mhz
- * Power 230 mW
- * Price 32 USD



ASICs

*Application-specific integrated circuit

- ◆ Fully custom hardware
- ◆ Custom for your application
- ◆ As small or large as necessary

*Characteristics

- ◆ Expensive to develop
 - 10s of engineers, often 100s
- ◆ Large series necessary to pay off
 - At least 100 000 units necessary on average
- ◆ Mostly for large companies
- ◆ Typically, they become SoCs

ASIC

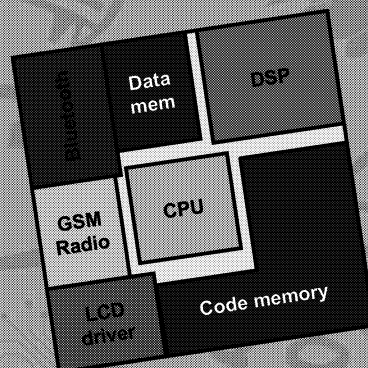
ASIC Components: "IP"

*IP Blocks

- ◆ *Intellectual Property*
- ◆ Companies sell pieces of hardware

*Examples:

- ◆ CPU Cores
- ◆ Memory
- ◆ Buses
- ◆ Network interfaces
- ◆ Accelerator circuits



CPU Cores

- ✱ **The biggest “IP” business**
- ✱ **Biggest players:**
 - ◆ ARM (best-selling 32-bit architecture)
 - ◆ MIPS (and its licensees)
- ✱ **Crowded field**
 - ◆ New companies appear monthly
 - ◆ “Fabless” semiconductor companies
 - ◆ Tuned for a particular application

Hard vs Soft IP

- ✱ **Hard IP:**
 - ◆ Customer buys a core as black box
 - ◆ Examples: ARM & MIPS
 - ◆ Gives good performance
 - ◆ Hides trade secrets
- ✱ **Soft IP:**
 - ◆ Get HDL code for the component
 - ◆ Examples: ARC & Tensilica
 - ◆ Integrate with own or other logic
 - ◆ Loses some performance

IP Core: ARM 926EJ-S

* Core "macrocell"

- ◆ CPU core, caches, bus interface, MMU as a package

* Instruction sets:

- ◆ Von Neumann architecture
- ◆ 32-bit ARM v5TE ISA
- ◆ 16-bit THUMB ISA
- ◆ Java bytecodes via Jazelle

* Processing power:

- ◆ Five stage pipeline, scaling to 180-270 Mhz
- ◆ 8 kB icache and 8 kB dcache

* Power: 0.2 to 0.9 mW/Mhz (P4: >35 mW/Mhz)

* MMU: for Symbian, Windows CE, Linux

IP Core: MIPS 24k

* Macrocell like the ARM926

- ◆ Processor, cache, memory interface, MMU, TLBs

* Instruction sets:

- ◆ MIPS16e
- ◆ MIPS32
- ◆ User extensions possible, via "CorExtend"

* Performance:

- ◆ 8-stage scalar pipeline, up to 550 Mhz
- ◆ Configurable cache, up to 64kB L1 I\$ & D\$
- ◆ Dynamic branch prediction

* Aimed at multiprocessor SoCs

- ◆ Cache coherency protocol standard
- ◆ Almost equivalent to a 1990's server processor

Example: Ericsson Bluetooth

- *ARM for protocol stack**
- *Memory for the code**
- *Special hardware for RF parts**
- *USB and serial connections**
- *Market**
 - ◆Aiming at huge volumes
 - ◆Component in mobile phones etc.

Producing Your ASIC

- *Old way: "Inhouse"**
 - ◆Build your own fab (everyone did!)
- *New way: "Silicon Foundries"**
 - ◆Fabs are getting very expensive
 - ◆Specialized fab companies
 - ◆Sell manufacturing capacity
 - ◆Examples: TSMC, UMC, IBM, TI
 - ◆Customers: Nvidia, ATI, Sun, Cisco
- *=Rise of "fabless" companies**

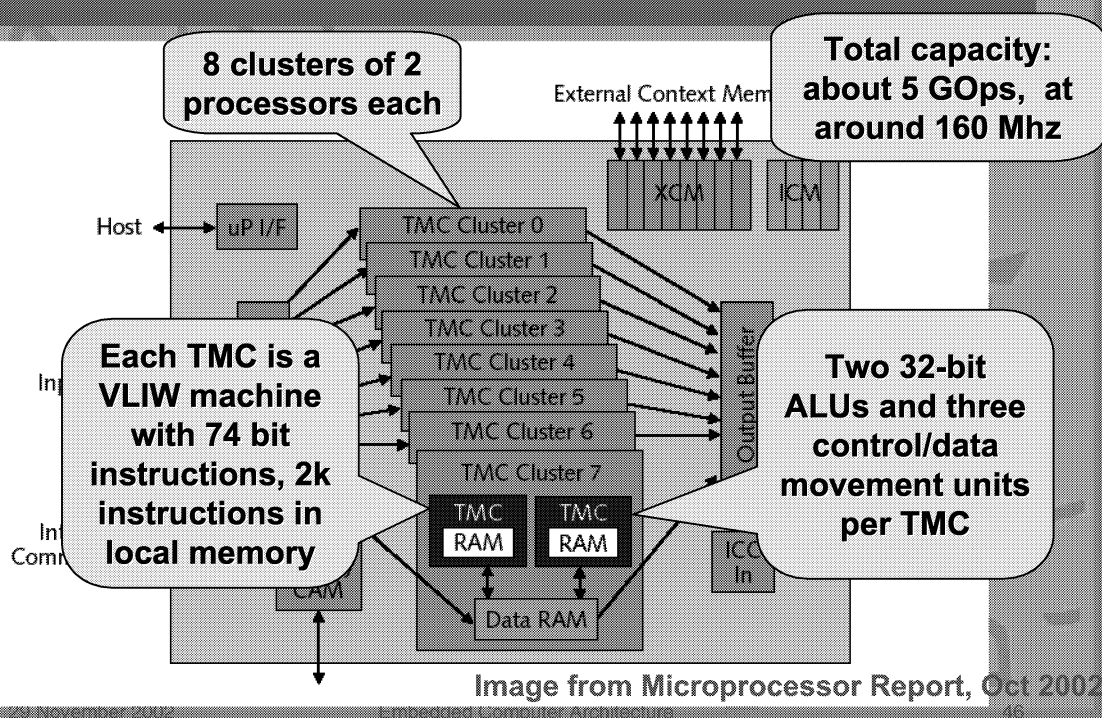
Full-Custom Systems

- ★ Volumes are high enough
- ★ Needs are special enough
- ★ In-house processor design

★ Examples:

- ◆ Ericsson APZ (now defunct)
- ◆ Cisco Toaster3 network proc (NPU)
- ◆ Ericsson FlexASIC DSP

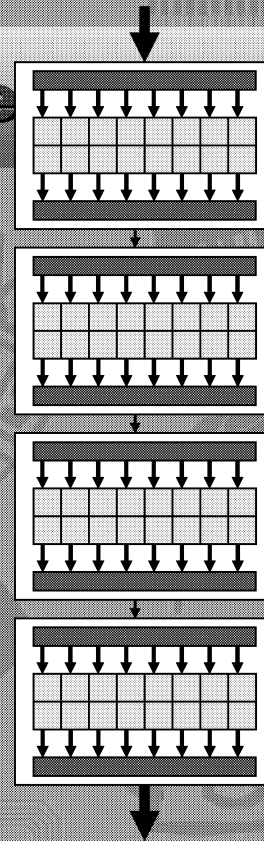
Cisco Toaster3



Cisco Toaster

*Massive multiprocessing

- ◆ 16 cores on a chip
- ◆ 4 chips in serial
- ◆ Routing:
 - 10 Gbps
 - @ 20 Mpackets/s
 - 1000 ops per packet passing through



29 November 2002

Embedded Computer Architecture

47

FPGA

*Field Programmable Gate Array

- ◆ Reconfigurable hardware: "soft logic"
 - "Program" is circuit layout
 - Can be changed after initial load
- ◆ Kilos to Megs of "gates" available

*Competitor to ASICs

- ◆ More expensive per unit, but no start-up cost for manufacturing
- ◆ Less flexible, slightly slower
- ◆ Perfect for low-volume products

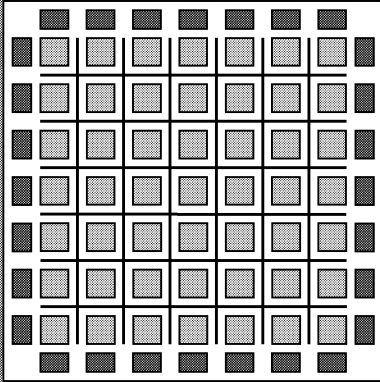
FPGA

29 November 2002

Embedded Computer Architecture

48

FPGA Architecture



□ Computation cells

◆ Programmable function

- Adder, Logic funcs, ...
- Memory, Registers, ...

■ Input/Output cells

≡ Interconnect

- ◆ Reconfigurable
- ◆ Programmable

FPGA Architecture

★ Computation cells

◆ Look-Up Table

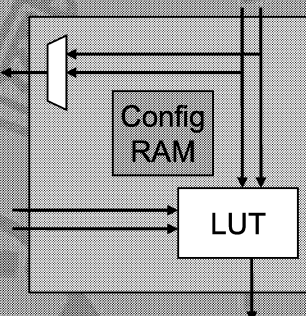
- Arbitrary 4-input, 1-output function

◆ Coarse-grained

- Lots of functionality
- Several LUTs
- Plus flip-flops etc.

◆ Fine-grained

- Little functionality



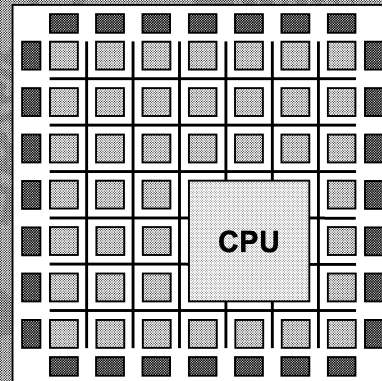
FPGA with CPU Cores

*CPU on-board FPGA

- ◆ HW accelerate critical tasks in FPGA fabric
- ◆ Data pumps in FPGA
- ◆ Control in CPU

*Cool new possibilities

- ◆ Reconfigure FPGA online
- ◆ Adapt to workloads



Soft CPUs in FPGAs

*Processor in the FPGA fabric

- ◆ "Soft" processor
- ◆ Special design considerations

*Examples

- ◆ Altera Nios
- ◆ Xilinx Microblaze
- ◆ Research projects
 - Västerås ARM clone
 - Leon processor also prototyped

Examples

*Altera Apex 20kC *Altera Stratix

- "Volume"
- 30k to 1.5M gates

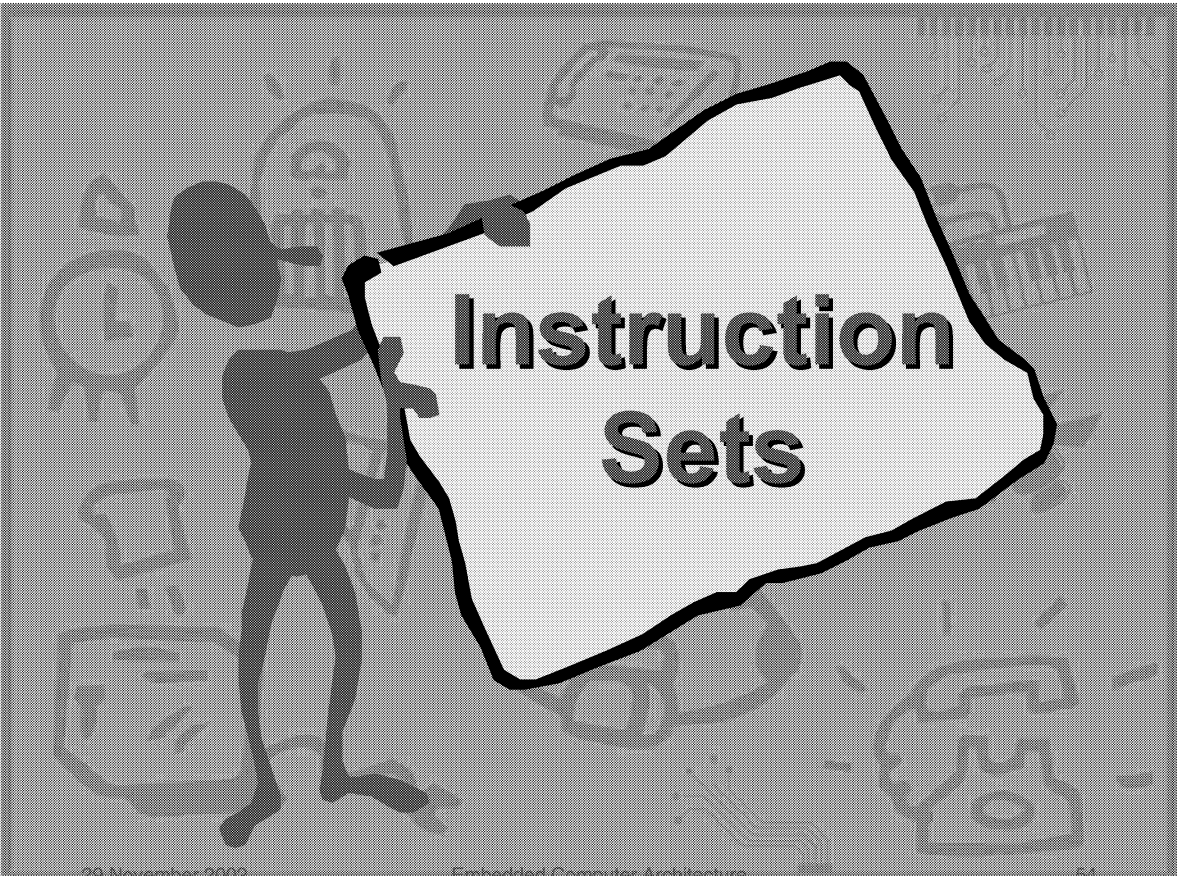
*Xilinx Virtex II:

- "High-end"
- 1-4 PPC405 cores (optional)
- 10M gates
- Price at about \$1000

- "Advanced"
- 10 Mbit RAM
- 28 DSP elements
- 100000 LE
- 1300 user I/O pins
- Optimized for Nios

*ATMEL FPSLIC:

- "Low-end"
- AVR 8-bit CPU
- 50k gates



Instruction Sets

IS Architectures

*New life for old architectures

- ◆ Z80, 6502, 8051, PIC,, 68000-ColdFire

*New career for failed desktops

- ◆ MIPS, PowerPC

*Fresh architectures

- ◆ AVR, dsPIC, V850, SH, ...

*Digital signal processing

- ◆ C5xxx, BlackFin, MSA, 56000, Oak, ...

Instruction Sets

*Code Size important

◆Variable instruction length

- Common instructions short
- Short and long branches
- RISC machines with 16-64 bit instructions
- Limited immediate operand sizes
- Two-operand rather than three-operand

◆Compact and powerful instructions

- Push/pop multiple
- Switch

Instruction Sets

*Special-purpose instructions

- ◆Digital Signal Processing
- ◆Bit-manipulation
 - Set bit in memory, test bit in memory
 - Several memory accesses per instruction
- ◆Application-specific
 - Fuzzy logic support (68HC12)
 - Table interpolation (68300)
- ◆Or even designed by customers!

*Do useful things=powerful

29 November 2002

Embedded Computer Architecture

57

Instruction Sets

*Compressed instruction sets

- ◆ARM/Thumb & MIPS16
- ◆16-bit encoding of (parts of) 32-bit instruction sets
- ◆Performs better on narrow buses
- ◆Limitations in ARM/Thumb:
 - Only access to 8 registers
 - No system operations
 - No multiply-accumulate
 - No general conditional execution

29 November 2002

Embedded Computer Architecture

58

Instruction Sets: Code Size

*Some data on code size:

| | Thumb | ARM | 386 | 8088 | 68020 | SPARC |
|----------|-------|--------|--------|--------|--------|--------|
| eqntott | 10608 | 16768 | 17640 | 19106 | 20542 | 22256 |
| | 0.63 | 1.00 | 1.05 | 1.14 | 1.23 | 1.33 |
| xlisp | 26388 | 40768 | 28097 | 29401 | 46746 | 44648 |
| | 0.65 | 1.00 | 0.69 | 0.72 | 1.15 | 1.10 |
| espresso | 72596 | 109923 | 125686 | 137194 | 131854 | 142752 |
| | 0.66 | 1.00 | 1.14 | 1.25 | 1.20 | 1.30 |

Source: Microprocessor Report, March 1995

Instruction Sets: Code Size

*ARM Thumb: fixed 16-bit size

- ◆Saves 28% compared to 32-bit ARM
- ◆Runs 20% slower than 32-bit ARM

*ARM Thumb 2: mixed 16/32

- ◆Saves 26% compared to 32-bit ARM
- ◆Runs 2% slower than 32-bit ARM
- ◆(Note that some new instructions are introduced)

*Conclusion: mixed length good!

Source: Microprocessor Report, June 2003

Instruction Sets: Code Size

*Compiler makes a difference

| Program | Compiler | | | |
|---------|----------|-------|-------|--------|
| | A | B | C | D |
| 1 | 4316 | 4929 | 4974 | 5214 |
| 2 | 16826 | 18176 | 26705 | 15968 |
| 3 | 1632 | 2594 | 3450 | 3244 |
| 4 | 5514 | 13804 | 22694 | 15000+ |

Source: IAR Internal Benchmarking

Instruction Sets: SIMD

*Many applications see gains from SIMD/Vector computation

*Add SIMD to regular ISA

- ◆Motorola AltiVec
- ◆ARM SIMD extensions
- ◆MIPS have it too
- ◆x86 MMX-SSE-SSE2-3Dnow!
- ◆SPARC VIS

Instruction Sets: SIMD

* Target

- ◆ Motorola
- ◆ PPC 7455 (G4+)
- ◆ 1 Ghz

* EEMBC

- ◆ Telemark suite
- ◆ Networking suite

* OOTB:

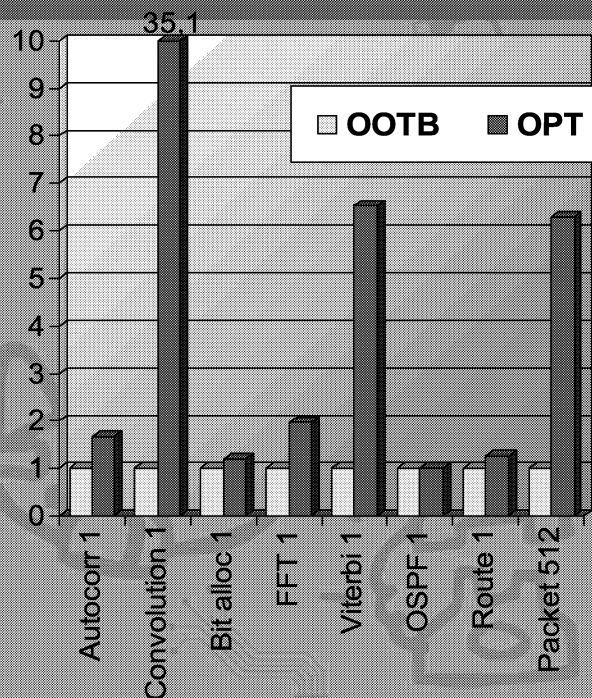
- ◆ Out-of-the-box

* OPT:

- ◆ Manually tuned to use Altivec

* Overall/Average:

- ◆ 3-4 times speed up can be expected



Instruction Sets: DSP

* Pure DSPs

- ◆ Not additions to regular ISAs

* Very specialized for DSP work

- ◆ Known & narrow class of problems
- ◆ Optimize for particular algorithms

* Categories

- ◆ VLIW vs. Regular
- ◆ Fixed vs. Floating Point
- ◆ Stationary vs. Mobile

Instruction Sets: DSP

*TI C64xx

- ◆ Fixed-point, 8-way VLIW
- ◆ 700-1000 Mhz, “Fastest DSP”
- ◆ Stationary applications

*TI C55xx

- ◆ Single pipeline, complex instructions
- ◆ Up to 300 Mhz approx.
- ◆ Mobile phones

Instruction Sets: DSP

*Assume very regular workloads

- ◆ Zero-overhead loop instructions
- ◆ Built to wade through large data sets

*Register sets

- ◆ Accumulators (often 40 bits)
- ◆ Data registers (often 16 bits)
- ◆ Address registers (16 to 32 bits)

*Addressing modes

- ◆ Index registers
- ◆ Post & preincrement
- ◆ Bit-reverse addressing
- ◆ Goal: more parallelizable work per instruction

Instruction Sets: DSP

*Example instructions from C55:

*"Finite impulse response"

- ◆ FIRSADD Xmem, Ymem, Cmem,

- ◆ Operation:

- $ACy = ACy + (ACx * Cmem)$

- $ACx = (Xmem \ll \#16) + (Ymem \ll \#16)$

Cmem, Xmem, Ymem:
memory accesses +
address updating

*"Conditional add or sub"

- ◆ ADDSUBCC Smem, ACx, TCx, ACy

- ◆ Operation:

- If $TCx = 1$, then $ACy = ACx + (Smem \ll \#16)$

- If $TCx = 0$, then $ACy = ACx - (Smem \ll \#16)$

C55 DSP has three
independent data
buses, X, Y, and C

Special
condition
register

Instruction Sets: Configure

*Configurable instruction sets

- ◆ Adapt to needs of application
- ◆ User can specialize the processor
- ◆ Less waste on generality
- ◆ Fast evolution of instruction sets

*Traditionally:

- ◆ Chip manufacturers determine instruction sets aimed at some niche
- ◆ Slow evolution of instruction sets

Instruction Sets: Configure

***Subsetting**

- ◆ There is a limited and predefined set of instructions available
- ◆ Easy to compile for: restrict code gen
- ◆ Remove instructions to simplify core

***Addition**

- ◆ Freedom to invent instructions
- ◆ Tool chain: assembly, C compilers
- ◆ Genuine development of ISAs

Configurable Instruction Sets

***Tight integration:**

- ◆ Add to regular pipeline
- ◆ Additional functional units
- ◆ Adding fine-grained instructions

***Loose integration:**

- ◆ Coprocessor interface
- ◆ Slower communication
- ◆ Offloading of macro-scale tasks
- ◆ Method to invoke accelerator circuits

Configurability Trend

★Pioneers

- ◆Tensilica Xtensa
- ◆Arc Arctangent
- ◆Configurability as key selling point

★Added to general architectures

- ◆MIPS: “CorExtend”
- ◆PowerPC: “BookE ASU”
- ◆Usually less tight integration

Benefit of Configurability

★ Target

- ◆ Xtensa III
- ◆ 200 Mhz

★ EEMBC

- ◆ Telemark suite
- ◆ Networking suite

★ OOTB:

- ◆ Out-of-the-box
- ◆ 25k gate core

★ OPT:

- ◆ Tuned code
- ◆ 25k base core gates
- ◆ 18k extra instr gates
- ◆ 100k DSP coproc
- ◆ 37k config gates

★ Speedups

| Benchmark | OOTB | OPT |
|-------------------------|------|------|
| <i>Telemark overall</i> | 1 | 37 |
| Autocorr | 1 | 9 |
| Convolution | 1 | 1249 |
| Bit alloc | 1 | 34 |
| FFT | 1 | 24 |
| Viterbi GSM | 1 | 14 |

Configuration Tools

The screenshot shows the ARCHitect configuration tool interface. The title bar reads "ARCHitect : C:\PROGRAM FILES\ARC CORES LIMITED\ARCHITECT DEMONSTRATION\...". The menu bar includes "File", "Build", "View", and "Help". The toolbar contains icons for "Open", "Save", "Summary", "Model", "Diagram", "Data", and "Build".

The left sidebar lists the following configuration sections:

- Environment
- EDA Configuration
- Instruction Set Architecture
- DSP Architecture
- Cache Configuration
- Memory Subsystem
- Debug Components
- Ancillary Components

The main window displays the "Instruction Set Architecture" configuration. It includes a "Basecase Information" tab and an "Instruction Library" section with the following options:

- ☒ 32 x 32 Barrel Shifter (Fast/Small)
- ☒ 32 x 32 Multiplier (Fast/Small)
- ☐ Swap
- ☐ Min/Max
- ☐ Normalise

Below the instruction library, it states: "The instructions supported by the" (partially obscured).

At the bottom left, a "Warnings in design: 0" section is visible, with a "View Warning Information" button. Below this, the following summary statistics are shown:

- Target Clock Speed: 160
- Est. Gate Count: 57200
- Number of Bits: 146944

At the bottom, a status bar indicates: "32x32 Multiply extension instruction with special result register".

Two callout boxes highlight specific features:

- A box labeled "instruction set choices" points to the "Instruction Library" section.
- A box labeled "Gate and memory size counters" points to the summary statistics.

The bottom of the slide contains the text: "29 November 2002 Embedded Computer Architecture 73".

Memory Systems

A cartoon illustration of a person holding a large, irregularly shaped sign that reads "Memory Systems". The background is a light gray with faint, stylized icons of various electronic components and systems, including a clock, a calculator, a circuit board, and a computer monitor. The person is a simple black silhouette.

The bottom of the slide contains the text: "29 November 2002 Embedded Computer Architecture 74".

Microcontroller Memory

*RAM:

- ◆ Small (32 bytes and up)
- ◆ Stacks, variables, loaded code

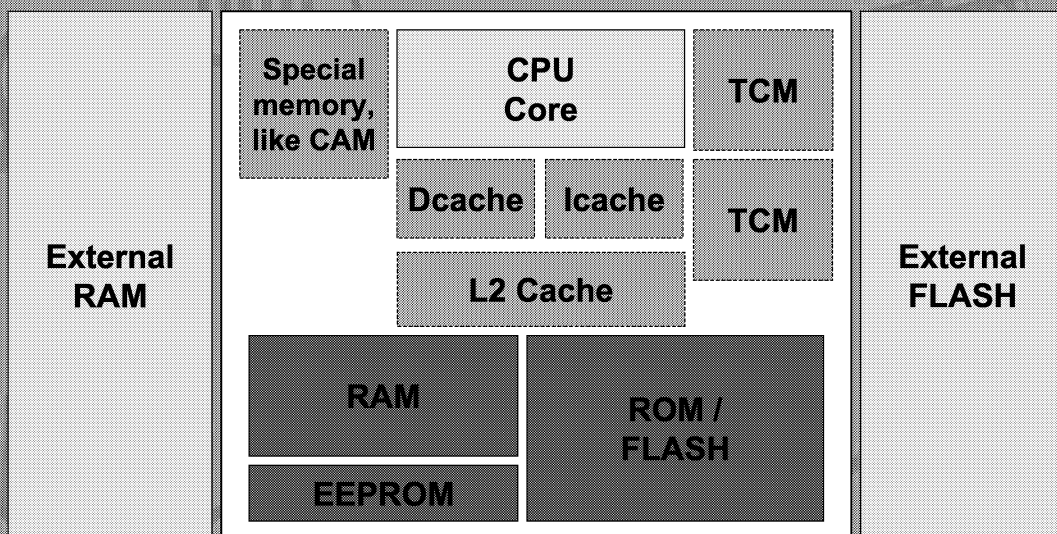
*ROM (or FLASH):

- ◆ Large (2kB and up)
- ◆ Programs, constant data

*Non-volatile memory

- ◆ Settings, writable code areas

Typical Memory Types



Caches

- * **Used on high-end parts**

- ◆ 32-bit, 64-bit, DSPs

- * **Not like desktop caches**

- ◆ Smaller, usually only single level
- ◆ Often high (128 ways) assoc
 - Due to locking

- * **Lockable**

- ◆ Sets or lines can be locked in cache
- ◆ Improve predictability

Icache

L2 Cache

Dcache

Tightly-Coupled Memory

- * **Used on high-end parts**

- ◆ Holds data and/or instructions

- * **Instead of/in addition to caches**

- ◆ Programmer-controlled
- ◆ Fast & close like caches
- ◆ In memory map, or tagged like caches

- * **Multiple banks**

- ◆ Better bandwidth
- ◆ Work in one, DMA data to/from other

- * **Special memories:**

- ◆ Content-addressable memory (CAM)
- ◆ Needs for particular applications

TCM

TCM

CAM

On-Chip RAM

*High-end parts

- ◆ Usually cached
- ◆ Faster & cheaper than off-chip memory

RAM

*Low-end parts

- ◆ Only data memory available
- ◆ Special “zero-page” memory

*Zero-page on 8-bit MCUs

- ◆ Small memory with single-cycle access
- ◆ Usually 8-bit index, contains 256 bytes
- ◆ Small, fast, instructions access the memory
- ◆ Often useable as extension to register set

FLASH/ROM

*Code storage on-chip

◆FLASH:

- Speed like regular RAM
- Rewritable, typically 1000 times or more

External
FLASH

◆ROM:

- Must be put in silicon masks
- Longer turn-around time
- Guaranteed not to change

ROM /
FLASH

*FLASH is replacing ROMs, fast

EEPROM

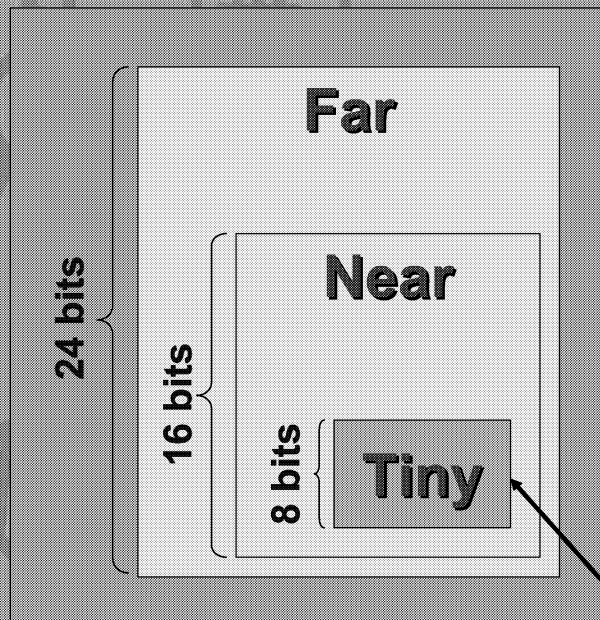
- * "Electrically-Erasable Programmable Read-Only Memory"
 - ◆ Only writable persistent memory until FLASH appeared
 - ◆ Infinitely rewritable
- * **Store persistent data**
 - ◆ User settings
 - ◆ Encryption keys
 - ◆ Phone numbers etc.
- * **Being replaced with FLASH**
 - ◆ Faster & easier to write
 - ◆ Cheaper to manufacture, higher capacity

EEPROM

Memory Architecture

- * **Narrow buses**
 - ◆ Saves silicon area, power, complexity
 - ◆ Especially to off-chip memory
 - ◆ (Not true for some high-performance parts)
- * **Small registers = small pointers**
 - ◆ 16-bit register can only hold 16-bits
 - ◆ Extend addressing using tricks
 - Banks (separate bank register)
 - Segments (base + offset)
 - Memory remapping (virtual memory)

Hierarchy of Pointers



- * 8-bit/16-bit
 - * Design for small pointers
 - * Visible to programmer
 - ◆ Data placement
 - ◆ Pointer types
 - ◆ Linker & compiler
- = on-chip zero-page

29 November 2002

Embedded Computer Architecture

83

Memory Architecture

*Harvard

- ◆ Two or more address spaces
 - ▢ Program
 - ▢ Data
 - ▢ Accompanied by physical separation
- ◆ Sometimes even more divided

*NULL pointer implementation?

- ◆ All addresses are valid...

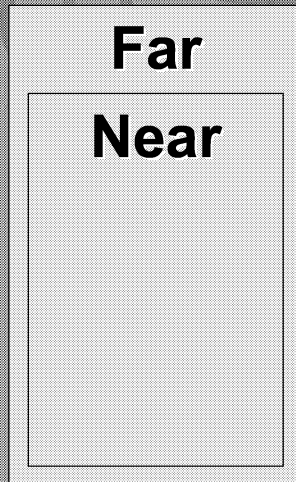
29 November 2002

Embedded Computer Architecture

84

Memory Architecture

*Example: ATMEL AVR 8-bit MCU



Far

Near

Tiny

256 B

RAM

Registers

I/O

* Code space

* Data space

* Read constants?

- ◆ Different instructions!
- ◆ Very slow process
- ◆ Hard to compile for
- ◆ Copy to RAM?

Banked Memory

*Extend addressing beyond N bits

- ◆ Like 8086/80286 segments

*Concept:

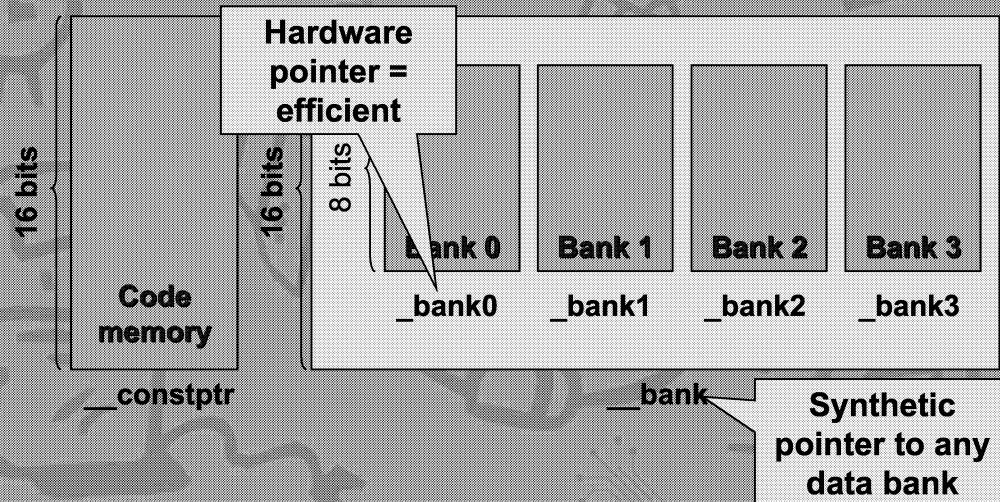
- ◆ Separate memories
- ◆ Mapped to same set of addresses
- ◆ One memory at a time accessible
- ◆ Easier for code than for data

*Selecting banks:

- ◆ Write value in bank-switch register

Banked Memory

*Example: Microchip PIC family



Power Aspects

Why is Power Important?

- ✱ **Battery-powered applications**

- ◆ Longer battery life is very desirable

- ✱ **Automotive applications**

- ◆ Electronics consumes up 30% of fuel!

- ✱ **Low-maintenance applications**

- ◆ Power=heat=cooling=moving parts

- ✱ **Server farms**

- ◆ Cooling & electricity are big costs



CMOS Power

- ✱ **Power = area*clock*voltage²**

- ◆ Area: transistors that are switching
- ◆ Clock: speed of switching
- ◆ Voltage: to keep running

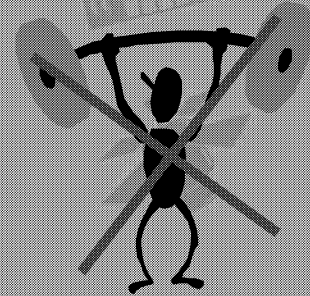
- ✱ **Save power by minimizing**

- ◆ Clock speed
- ◆ Active area
- ◆ Feed voltage

Area Reduction

*Simpler chips use less power

- ◆ 8-bit CPUs
- ◆ Simple RISC like ARM
- ◆ VLIW instead of superscalar



*Turn off inactive units

- ◆ Pipelines that are not used
- ◆ On-chip memory and caches
- ◆ Sleep/Nap/Idle modes on CPU

*Remove unnecessary features

29 November 2002

Embedded Computer Architecture

91

Clock Speeds

*Clock and voltage related

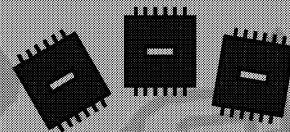
- ◆ Higher operating frequency requires higher voltage

*Use lower clock speeds

- ◆ Reduce speed until app barely works

*Use more processors

- ◆ $1/2$ speed = $1/4$ power
- ◆ 2 CPUs @ 100 Mhz = 1 CPU @ 200 Mhz, but requires half the power



29 November 2002

Embedded Computer Architecture

92

Dynamic Voltage Scaling

*Adjust CPU speed to workload

- ◆ Reduce operating voltage when clock speed is reduced
- ◆ Cubic power savings possible!
- ◆ Analyze load to determine speed
- ◆ More advanced than sleep modes

*Special hardware required:

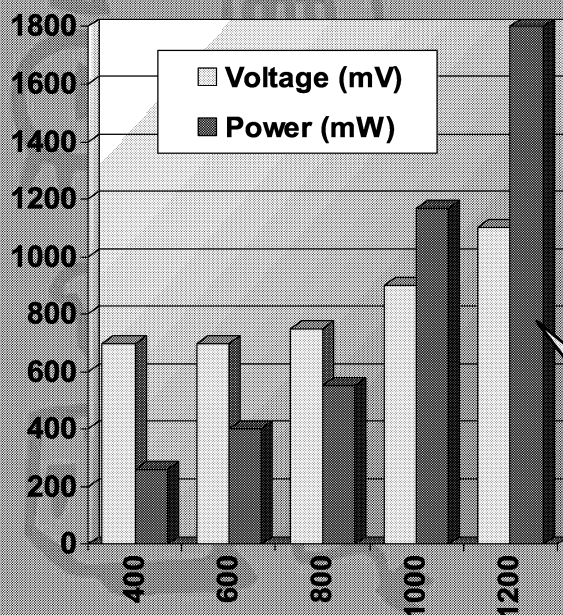
- ◆ Transmeta Crusoe was a pioneer
- ◆ Intel Xscale, getting common

29 November 2002

Embedded Computer Architecture

93

Power, Voltage, Frequency



*Samsung Halla

- ◆ ARM 1020E core
- ◆ 6-stage pipeline (!)
- ◆ 0.13 um process
- ◆ Clock: 400 Mhz to 1.2 Ghz

3x clock freq,
9x power!

(source: Microprocessor Report, Oct 16, 2002)

29 November 2002

Embedded Computer Architecture

94

Other Factors

*Manufacturing process:

- ◆Smaller features=lower power
 - (0.13 micron mobile PIII, for example)
- ◆Tweak process for lower power

*Development effort:

- ◆Tweak the low-level chip layout
 - (Classic StrongARM)
 - “Assembly language programming”
- ◆Cannot be synthesized efficiently
 - = Not possible for IP blocks

System Issue

*Much more than CPU

- ◆Displays, LEDs, ...
- ◆Memory, Disks, ...
- ◆Radio interfaces, Networks, ...

*Turn off unused peripherals

- ◆GSM phones: 300 hours standby vs. 60 minutes talk time
- ◆Ericsson: reduce frequency of LED blink



Memory & Power

- ✱ **Large area=high power:**
 - ◆ Use smallest possible memory
- ✱ **Talking to DRAM is expensive**
 - ◆ Use on-chip SRAM/ROM
 - ◆ Reduce external memory activity
 - ◆ Use caches to keep activity internal
- ✱ **Use energy-efficient RAMs**
 - ◆ Low-power DRAM is coming!
 - ◆ RAMBUS is horrible

29 November 2002

Embedded Computer Architecture

97

Memory & Power

- ✱ **Power for a LOAD instruction on an ARM7 development board**

| code memory | data memory | energy (nJ) | ratio |
|-------------|-------------|-------------|-------|
| off-chip | off-chip | 115.8 | 100% |
| off-chip | on-chip | 51.6 | 44.6% |
| on-chip | off-chip | 76.5 | 66.1% |
| on-chip | on-chip | 16.4 | 14.2% |

Source: Compilation Techniques for Energy-, Code-Size-, and Run-Time Efficient Embedded Software (Marwedel et al 2001)

29 November 2002

Embedded Computer Architecture

98

Future Issues: Static

*Dynamic Power

- ◆ Dissipated when circuits active
- ◆ Discussion so far
- ◆ Dominant down to 0.13μ

*Static Power

- ◆ "Leakage current"
- ◆ Becoming significant at $< 0.13\mu$
- ◆ Much harder to reduce
 - Major problem looming!



Closing Remarks

This is where the action is!

- *Fragmented market**
 - ◆ No dominant big player like PCs
 - ◆ Incredibly wide span of products
- *Tailor-made, not mass-produced**
 - ◆ Everybody searches for perfect fit
- *High innovation in comp arch**
- *Large number of new players**

Abbreviations

- *DSP**
 - ◆ Digital Signal Processor
- *NPU**
 - ◆ Network Processing Unit
- *MCU**
 - ◆ Microcontroller Unit
- *ASIC**
 - ◆ Application-Specific Integrated Circuit
- *FPGA**
 - ◆ Field-Programmable Gate Array

**THE
END!**

(C) Programming & Compiling for Embedded Systems



Jakob Engblom, PhD

Uppsala University & Virtutech Inc.

jakob.engblom@it.uu.se
jakob@virtutech.com

virtutech



Purpose of this Talk

**To give an understanding of
current industrial practice
regarding compilers and
programming technology for
embedded systems.**

Embedded programming

*Close to hardware

- ◆Function comes from HW
- ◆Each project has its unique HW
- ◆(+ lots of standard hardware)
- ◆Need to understand controlled system and the hardware being built

*Control functions dominate

- ◆External world very important
- ◆User interfaces usually non-existent

Cross-Compilation

*Cross-compilation:

- ◆Development system != target system

*Debugging:

- ◆Signals back to PC
- ◆Need communications link
 - Serial, USB, Ethernet, ...
- ◆Target can “run away”
- ◆Limited user interface on target (an LED?)



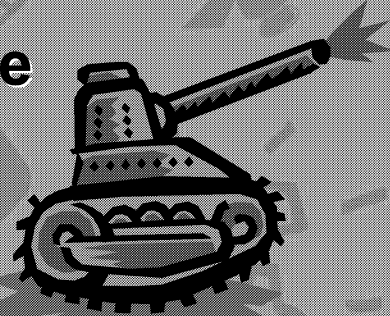
Use of Simulators

★ Hardware developed in parallel

- ◆ Work with software in simulators
- ◆ Simulate the world
- ◆ Simulate the peripherals

★ Hardware not available

- ◆ Large systems
- ◆ Dangerous systems
- ◆ Expensive systems



Operating Systems

★ RTOS: Real-Time Operating System

- ◆ What all embedded OSeS are called

★ No RTOS at all

- ◆ Common on simple devices

★ In-house RTOS

- ◆ By tradition in many cases
- ◆ Motivated for special needs (telecom)

★ Third-party RTOS

- ◆ Some rather big (VxWorks, OSE)
- ◆ Some small & lean (Rubus, SSX5)
- ◆ Available for 8-bit to 64-bit targets

Developers: Any Size!

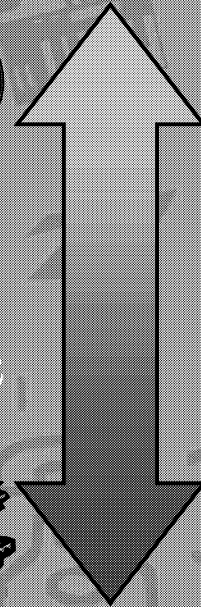
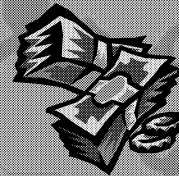
*Joe's garage

- ◆1 man,
- ◆PC from '97
- ◆Some old compiler



*MegaConsumerElectronics

- ◆1000s of engineers
- ◆Latest tools
- ◆Special tool department



Average project

*Medium-volume (100s - 1000s)

*Build system from components

- ◆Standard microcontroller(s)
- ◆Standard communications
- ◆Custom or standard sensors/actuators
- ◆"Board design"

*Costs important

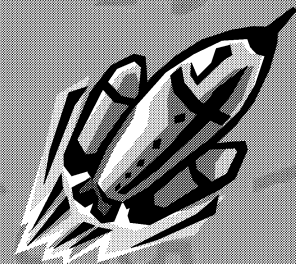
- ◆Cheap hardware & short dev. time

High-Volume Systems

- ★ **“Engineering time is free”**
 - ◆ Manufacturing cost is dominant
 - ◆ Small savings add up
 - ▬ Save 50¢ per unit, ship 1 million units...
 - ◆ Use cheapest possible hardware
 - ◆ Custom ASICs, in-house RTOS, ...
- ★ **Examples:**
 - ◆ Mobile phones, toys, automotive components, power tools, ...

High-Integrity Systems

- ★ **Have to work, always**
 - ◆ Rugged components
 - ◆ Old, well-known processors
 - ◆ Quality software, heavy processes
 - ◆ Certification requirements
 - ◆ Long development times
- ★ **Examples**
 - ◆ Aircraft, military equipment, space applications



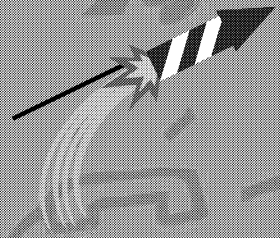
High-Integrity Systems

★Have to work, always

- ◆Rugged components
- ◆Old, well-known processors
- ◆Quality software, heavy processes
- ◆Certification requirements
- ◆Long development times

★Examples

- ◆Aircraft, military equipment, space applications



Programming Tools



Compilers & IDEs

*Languages:

- ◆ C, C++, assembler, Java, Ada
- ◆ Best support is for C
- ◆ C++ quite common, works on 8 bits!

*Providers

- ◆ Chip manufacturers (ARM, NEC, ...)
- ◆ Third-party compiler vendors (IAR, ...)
- ◆ Open-source compilers (GCC)

The Compiler Market

*8-bit & 16-bit

- ◆ IAR Systems, Keil, Cosmic, Tasking

*32-bit: ARM

- ◆ IAR, ARM, Keil

*32-bit: MIPS, PPC, 68k, x86:

- ◆ WindRiver, GreenHills, MetroWerks

*DSP

- ◆ Almost solely in-house compilers
- ◆ TI, ST, Motorola, Lucent, Intel

Debuggers

*Use simulator

- ◆ Requires simulator for target
- ◆ Easy and cheap solution

*Use debug server on target

- ◆ Simple solution
- ◆ Risk of run-away
- ◆ Requires spare ports on target
- ◆ USB, Serial/RS-232, Ethernet

Debuggers

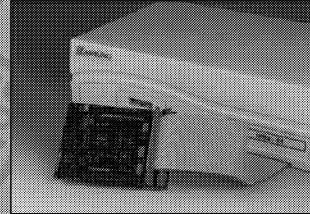
*Processors with debug support

- ◆ Designed into processor
- ◆ Use a few dedicated processor pins
 - Bandwidth not very high
- ◆ Variants:
 - BDM: Background Debug Mode
 - JTAG: The most common solution
 - Nexus 2001: New, powerful standard
- ◆ Very common on modern chips
 - ARM "Embedded Trace Macrocell"

Debuggers

* (In-Circuit) Emulators (ICE)

- ◆ “Hardware simulator”
- ◆ Extreme visibility
- ◆ High-bandwidth
- ◆ A dying breed
 - Modern processors too fast to emulate
 - Modern processors too complex
 - Being replaced with JTAG debuggers
- ◆ Last breaths
 - Special debug versions of CPU cores



Debuggers: Issues

* Support for hardware debug

- ◆ Drivers for each solution

* Object-code formats

- ◆ Many different formats in use
 - Not just ELF/DWARF and COFF...
- ◆ Open & vendor-specific
 - Example: IAR UBROF, closed format
- ◆ Information amount:
 - Greater in closed formats
 - Close integration of compiler and debugger

The Debugger Market

- ★ **Part of compiler/IDE suite**
 - ◆ GreenHills, Metrowerks, IAR, etc.
- ★ **Separate products**
 - ◆ Mentor XRAY
 - ◆ Accelerated Technologies Code|Lab
 - ◆ Lauterbach PowerView
- ★ **Part of RTOS packages**
 - ◆ OSE Illuminator

Simulators

- ★ **Instruction-set simulators**
 - ◆ Just churn the code, limited I/O
 - ◆ Run user-level applications
- ★ **Full-system simulation**
 - ◆ Simulate peripheral devices & CPU
 - ◆ Run full OS, drivers, applications
 - ◆ Rather rare category of product

Simulators

✱Hardware simulators

- ◆VHDL/Verilog-level sim of hardware

✱HW-SW co-simulation

- ◆VHDL-peripherals + CPU ISA
- ◆Big, expensive tools (Seamless)
- ◆Very useful for ASIC development
- ◆More on this next week at ESSES

Graphical Programming

✱Technology:

- ◆UML, UML-RT, SDL, StateCharts, ...
- ◆Generates C code
- ◆Needs to add in drivers to complete

✱Model-driven architecture:

- ◆Dream of researchers & companies
- ◆Program by composing models
- ◆Test composed models
- ◆Generate code = no low-level debug!

RTOS Support Tools

*RTOS have complex options

- ◆Much more modular than desktop OS
- ◆Many optional components
- ◆Tailor for a particular application
- ◆= configuration tools!

*Timing analysis tools

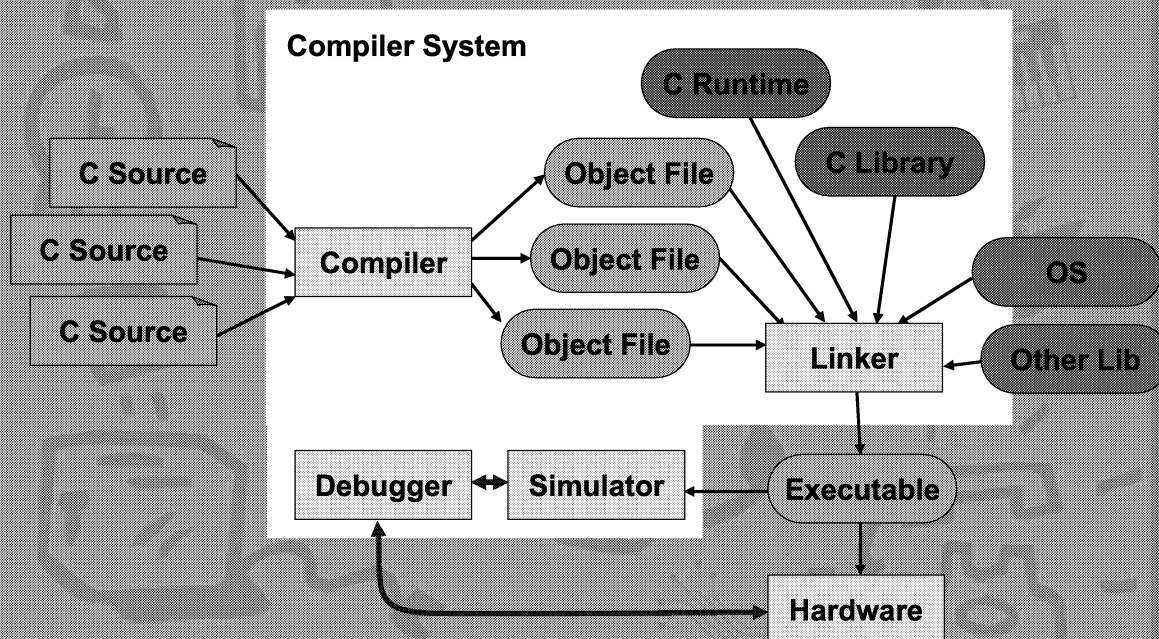
- ◆Profiling, timing measurements
- ◆Response time analysis

*Etc...



Compiler Tool Chain

Tool Chain



29 November 2002

Embedded Computer Architecture

25

Compiler

✱Obvious, right?

29 November 2002

Embedded Computer Architecture

26

Linker

*Combine object files

- ◆ Locate variables and functions in memory
- ◆ Resolve function calls and jumps

*Remove unused data and code

- ◆ File, function, or sub-function granularity?

*Link-time transformations

- ◆ Cross-call / code factoring
- ◆ Remove unnecessary bank/page switches
 - ▀ Knows precise address of all variables & functions
- ◆ Optimize branch sizes

The C Library

*Programmer-visible

- ◆ Standardized in ANSI C

*“Standalone” profile:

- ◆ No file operations
- ◆ Suitable for run-from-ROM systems

*Limited versions:

- ◆ Smaller code through less functionality
- ◆ Provided with compiler
- ◆ Non-standard

The C Run-Time System

*Part of compiler

- ◆ Designed with the compiler
- ◆ Implements complex functions
- ◆ Called from compiled code
- ◆ Not visible to programmer

*Whatever is needed that the hardware does not support

*Bigger for simpler machines

The C Run-Time System

*Example functionality

- ◆ 32-bit arithmetic
- ◆ Floating-point arithmetic
- ◆ Pointer use:
 - Banked, generic, large, function pointers, ...
- ◆ Complex C operations:
 - Switches (jump tables, assoc-tables)
 - Variable-length shifts (" $a \ll x$ ")
- ◆ May have a large footprint
 - Tens of bytes of data
 - Tens of kilobytes of code

Executable

*Relocatable binary

- ◆ Desktop-style, with loaders
- ◆ For "big" RTOS

*Absolute-located binary

- ◆ The most common case!
- ◆ For burning into ROM or FLASH
- ◆ Adapted to "run from ROM"
- ◆ No dynamic code loading
- ◆ OS is part of binary image



**Compiler
useability**

The Problem

*Desktop tool tradition

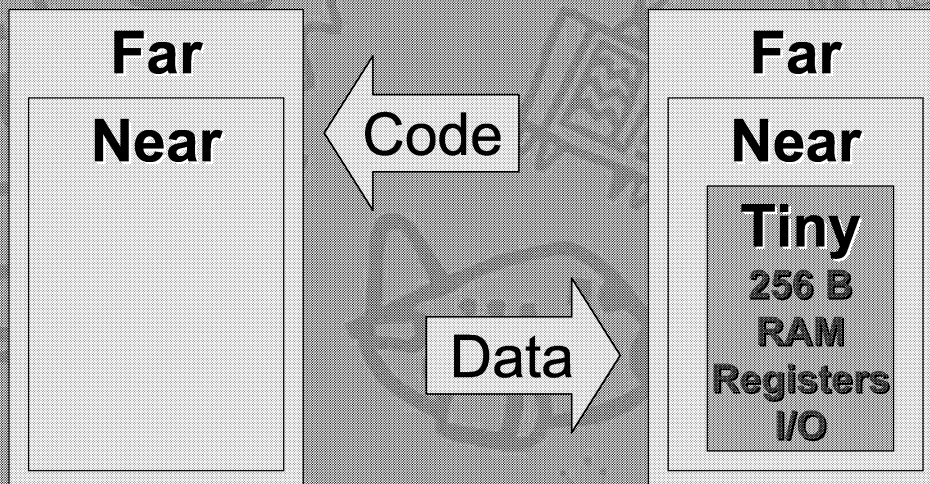
- ◆ Assume OS present
- ◆ Assume host=target
- ◆ Assume uniform memory space

*Embedded programming tools

- ◆ Address irregularities
- ◆ Access hardware
- ◆ ... but still adhere to standards

Remember This?

*AVR Memory Setup



The Solution

- *Cross-compilers
 - *Extension to the C language
 - *Intrinsic functions
 - *Linker scripts
- *All these help a programmer achieve an acceptable solution

Extensions to C

- *Styles
 - ◆#pragma
 - Allowed in ANSI C, but unreadable in practice
 - ◆Extra keywords
 - Convenient to use: `__near int * x;`
 - Breaks syntax, but can be `#define`d away
 - ◆gcc-style attributes
 - Slightly cumbersome, but easy to define away
 - Example: `int x __attribute__((“near”));`
 - ◆Creative new syntax
 - Breaks syntax irrevocably: `char port @ 0xFF`

Extensions to C

*Direct access to hardware

◆Located variables

- Standard C: cast constants to pointers
- "volatile char input @0x0020;"

◆Read-only/write-only locations

- Interpret const keyword strictly = read-only
- Keywords for write-only

*Special write methods:

- "__flash char setting;"

Extensions to C

*Handle persistent memory:

- "__no_init __eeprom char setting;"

*Placing variables in good areas

- "__zeropage char CommonVar"
- "__bank1 int LessCommonVar"

*Giving pointers good types

- "__bank1 char *bank1ptr"
- "__generic char *ptr"

Extensions to C

*Functions can also be special

◆Interrupt handlers

➤ `"__interrupt void Handler(void)"`

◆Tasks that never return

➤ Automatically detect, or use keyword

◆Special calling conventions

➤ `"__fastcall int foo(int)"`

Intrinsic Functions

*Caught by compiler

◆Replaced by inline assembler

◆Can be emulated on other targets

*Special features of the CPU:

➤ `"__multiply_accumulate(x,y,z)"`

➤ `"z=saturated_addition(x,y)"`

*Things outside C semantics:

➤ `"__disable_interrupts()"`

➤ `"__delay(n)"`

Linking

*Object code organized by segments

- ◆ Sets of code and data
- ◆ With the same destination in memory

*Desktop segments:

- ◆ Code and readonly data
- ◆ Zeroed variables
- ◆ Initialized variables

*Embedded: more complicated

Linking: segments

*Embedded segments

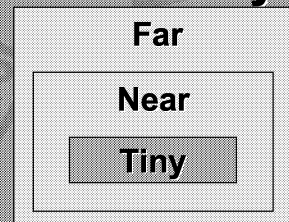
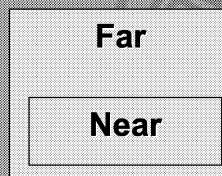
- ◆ One segment for each distinct memory

◆ Code

- Near code
- Far code

◆ Variables

- Tiny, near, far
- Zero-initialized, data-initialized, not initialized
- Initialization data, put in code space?



*Linker command files

- ◆ Directions for placing segments
- ◆ Sometimes quite complex...

Deviant C Variants

*Some deviant versions of C

◆ "Micro-C":

- 8-bit ints, no 32-bit support, etc
- Easy to compile to 8-bit machines
- Hard to port code to and from

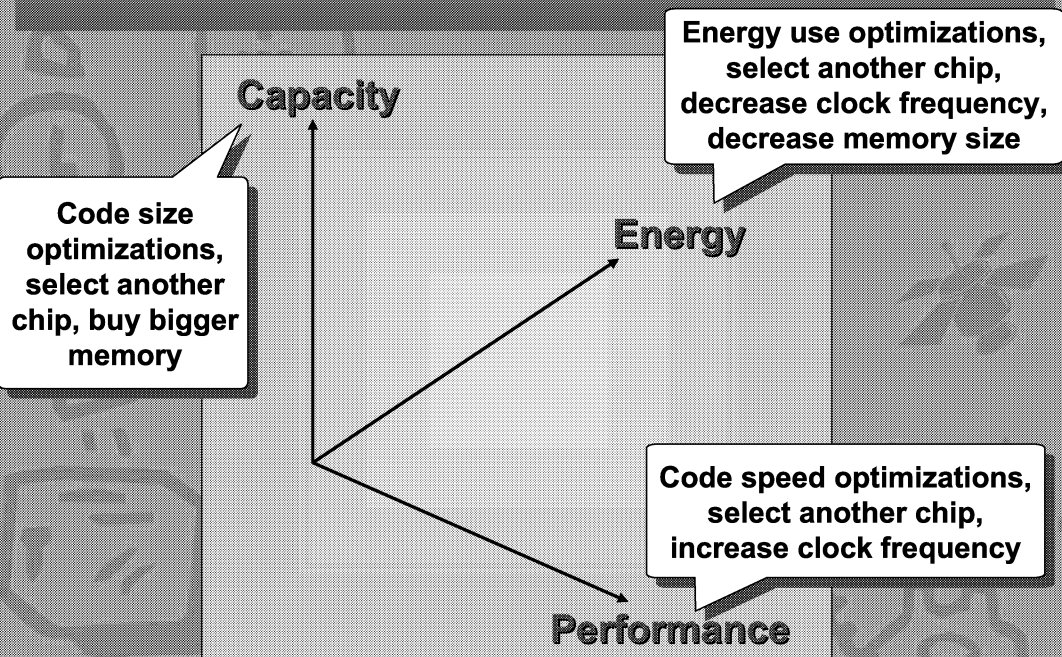
◆ "DSP-C":

- Add new types to C to represent DSP types
- Overload built-in operators
- Syntactically incompatible with ANSI C
- Bad idea, can be solved in better ways
- Sold by "ACE" and its CoSy system



Optimization Challenges

The Solution Space



29 November 2002

Embedded Computer Architecture

45

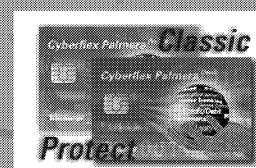
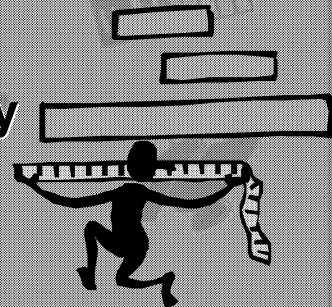
Why Care About Size?

*Keep memory cost down

- ◆ Avoid external memory
- ◆ Use smaller external memory
- ◆ Use a smaller derivative

*Keep absolute limits

- ◆ HW done before the software
- ◆ Application can only use on-chip memory



29 November 2002

Embedded Computer Architecture

46

Size Optimization, basics

✳ Many classic optimizations

- ◆ Strength reduction
- ◆ Dead code elimination
- ◆ Copy / constant propagation

✳ Select small over fast

- ◆ Use library calls to implement complex ops
- ◆ Use slow but powerful instructions

✳ Function inlining

- ◆ Can help by reducing function call overhead

✳ Avoid loop unrolling

Size Optimization

✳ Code Factoring

- ◆ Linker or compiler optimization

```
foo()
...
MOV R30,R26
?foo_end:
MOV R31,R27
LDD R20,Z+15
LDD R21,Z+16
LDD R22,Z+17
OR R20,R21
OR R20,R22
RET
```

```
bar()
...
OUT 62,R25
MOV R31,R26
JMP foo_end
```

```
bar()
...
OUT 62,R25
MOV R31,R26
MOV R31,R27
LDD R20,Z+15
LDD R21,Z+16
LDD R22,Z+17
OR R20,R21
OR R20,R22
RET
```


**THE
END!**

Discussion Topics ESSES 2003

To Think About

- *Can a compiler automatically remove the need for keywords?**
 - ◆ Automatically place variables?
 - ◆ Set pointer types?
 - ◆ Resize variables to smallest size
 - ◆ Reduce use of intrinsics?
 - ◆ Identify complex instructions?
- *Are there novel optimizations that can reduce the size of code, beyond what is done in industry today?**

To Think About

- *How can compilers help when building software for multiprocessors-on-a-chip?**
- *What kind of novel optimizations should be made for deeply embedded systems, breaking away from the usual "speed or size" thinking?**
- *Can a compiler impact power consumption in any reasonable way?**

To Think About

- *How can we make use of a heterogeneous memory system in a compiler? (On-chip SRAM, caches, on-chip slower memory, off-chip memory)**
- *Is it reasonable to automatically construct hardware from a program, to use in an FPGA-CPU combination?**

To Think About

- *What kind of instruction sets are best suited for fast and small code? How can THUMB etc. be improved on?**
- *How can compiler technology help a company sell their embedded processors? How can a compiler add value to a chip? Think business!**

To Think About

- *How could current compilers and languages be extended to better support static configuration in dynamic languages like C++?**
 - ◆An example is where a program always creates the same set of objects at startup, with the same configuration parameters, and this is known from the source code. But the usual implementation is to create objects, use pointers to point out other objects, and store the configuration in variables.**

To Think About

- ✱ **How does Java impact compiler technology? What are the most promising techniques for enhancing Java runtime performance (software-based, hardware-based, mixtures - cf. ARM Jazelle) and improving code size?**
- ✱ **Is Just-in-Time compilation compliant with safety requirements of embedded systems? What changes are required for test and validation procedures?**

To Think About

- ✱ **Achieving code predictability is an important topic for real-time systems. Does this influence compiler technology and code generation?**
- ✱ **How can C be adapted to the needs of DSP programming?**
 - ◆ Fixed point types, automatic conversion from float?
 - ◆ Intrinsic functions or new operators?
 - ◆ Powerful but weird addressing modes, how to use?
 - ◆ Smart inference of types and operations?
 - ◆ Other compiler support that helps?

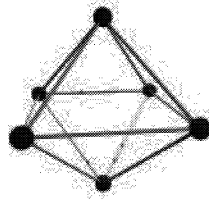
**THE
END!**

Dr. – Ing. Daniel Kästner

AbSint
Angewandte Informatik GmbH
kaestner@absint.com

Compilation for Embedded Processors

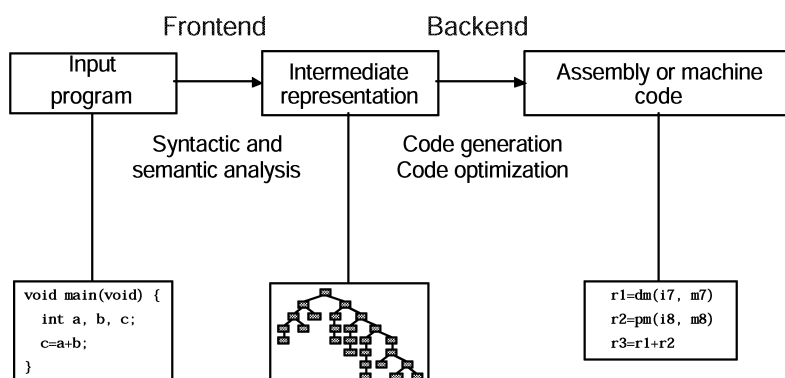
Basics and Principles



Dr.-Ing Daniel Kästner
kaestner@absint.com

AbsInt
Angewandte Informatik GmbH

Compiler Structure

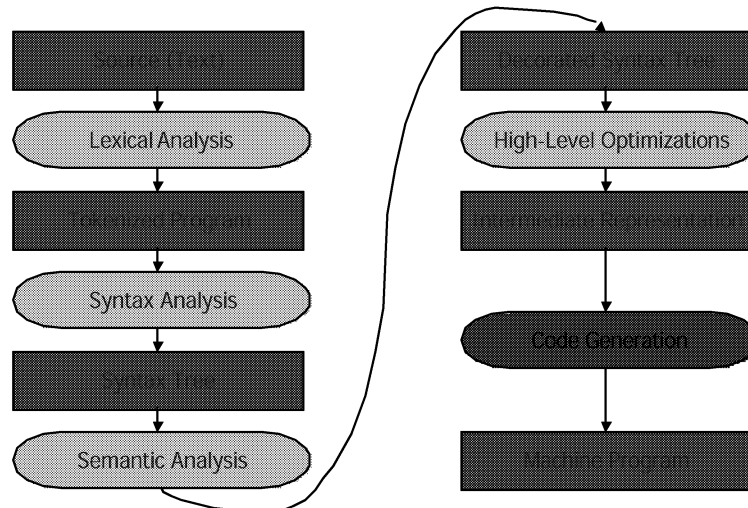


AbsInt
Angewandte Informatik



2

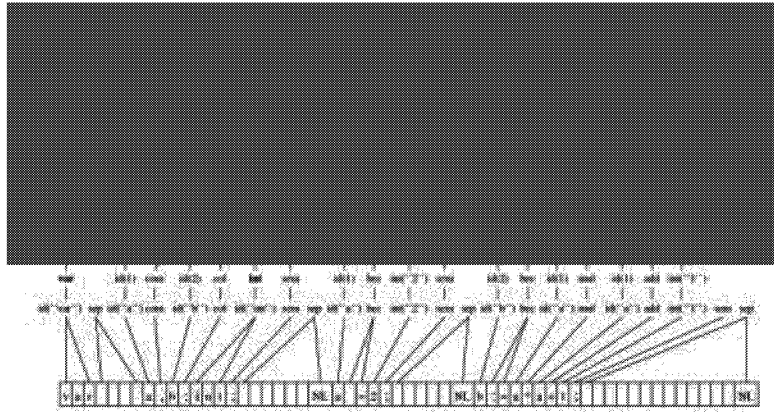
Detailed Compiler Structure



Lexical Analysis (Scanning & Screening)

- Input: Program text as sequence of characters.
Output: Program text as sequence of symbols (tokens).
- 1. Read Input file.
- 2. Report errors about symbols illegal in the programming language.
- 3. Screening subtask:
 - Identify language keywords and standard identifiers.
 - Eliminate "white-spaces", e.g. consecutive blanks and newlines.
 - Count line numbers.

Lexical Analysis (Scanning)





Syntax Analysis (Parsing)

- Input: Sequence of symbols (tokens).
Output: Structure of the program:
 - concrete syntax tree,
 - abstract syntax tree, or
 - parse (derivation).
- Syntax errors:
 - report (as many as possible) syntax errors,
 - diagnose syntax errors,
 - correct syntax errors.

[illegible]

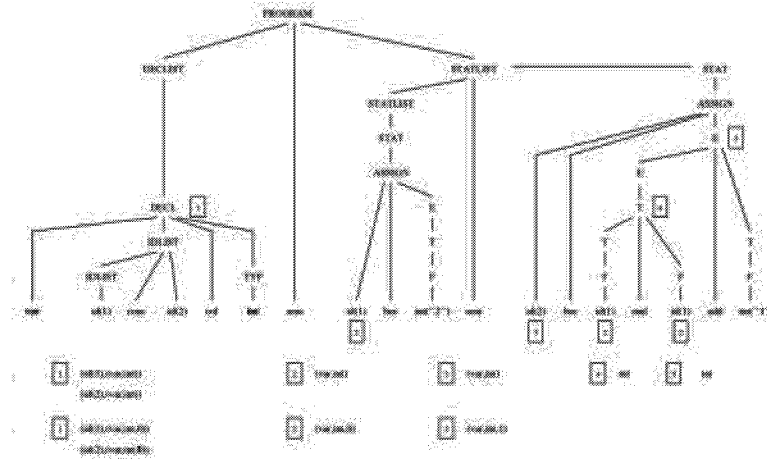
Semantic Analysis

- Input: Abstract syntax tree.
Output: Abstract syntax tree decorated with attributes, e.g. types of subexpressions.
- Report semantic errors, e.g. undeclared variables, type mismatches.
- Resolve usage of variables: identify applied occurrences of variables with their declarations.
- Compute type of every (sub-)expression.



8

Semantic Analysis



AbsInt
Angewandte Informatik



9

Middle End: High-Level Optimizations

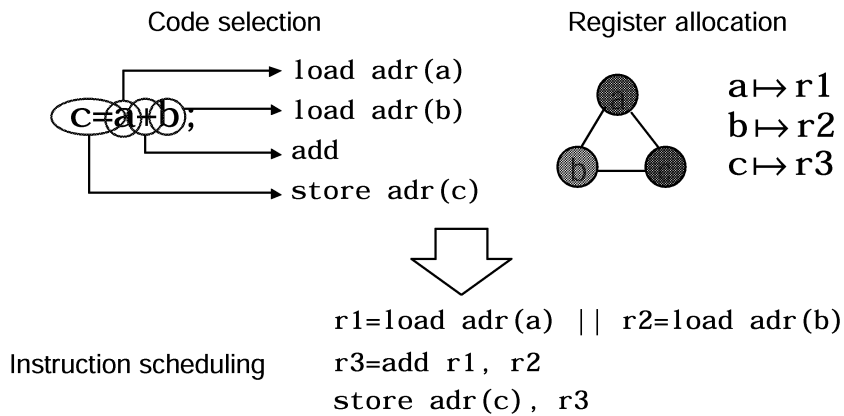
- High-level optimizations are usually termed machine-independent optimizations. They comprise e.g. dead code elimination, constant propagation, constant folding, common subexpression elimination, loop unrolling, loop fusion, software pipelining,...
- BUT: Many machine-independent optimizations are not machine-independent at all. For example:
 - constant folding may lead to large immediate constants resulting in code growth or preventing instruction-level parallelism
 - common subexpression elimination may increase the register pressure and cause problems if few registers are available
 - loop unrolling may cause the instruction cache to overflow

AbsInt
Angewandte Informatik



10

Backend: Code Selection, Register Allocation, and Instruction Scheduling

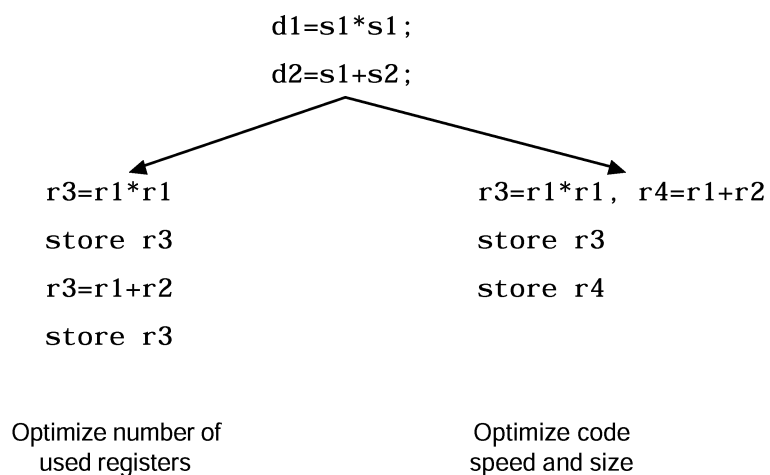


AbsInt
Angewandte Informatik



11

Phase Coupling Problem



AbsInt
Angewandte Informatik



12

Main Tasks of Code Generation (1)

- Code selection: Map the intermediate representation to a semantically equivalent sequence of machine operations that is as efficient as possible.
- Register allocation: Map the values of the intermediate representation to physical registers in order to minimize the number of memory references during program execution.
 - Register allocation proper: Decide which variables and expressions of the IR are mapped to registers and which ones are kept in memory.
 - Register assignment: Determine the physical registers that are used to store the values that have been previously selected to reside in registers.

Main Tasks of Code Generation (2)

- Instruction scheduling:
Reorder the produced operation stream in order to minimize pipeline stalls and exploit the available instruction-level parallelism.
- Resource allocation / functional unit binding:
Bind operations to machine resources, e.g. functional units or buses.

The Code Generation Problem

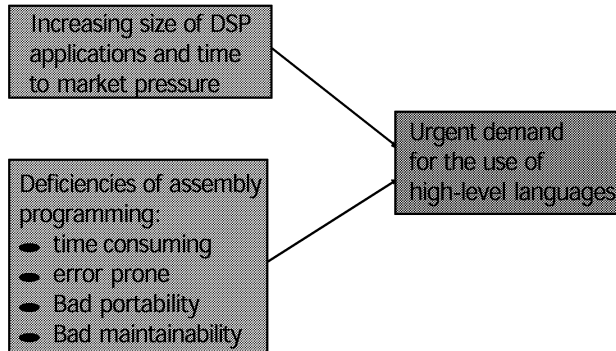
- Instruction scheduling, register allocation and code selection are NP complete problems.
- In classical approaches they are addressed by heuristic methods in separate phases.
- Unfortunately, all the code generation phases are interdependent, i.e. decisions made in one phase may impose restrictions to the other phases.
- Thus: often suboptimal combination of suboptimal partial results.
- Moreover: specific/irregular hardware features not well covered by standard code generation methods.

The DSPStone Study

- Evaluation of the performance of DSP compilers and joint compiler/processor systems. Evaluated compilers:
 - Analog Devices ADSP2101,
 - AT&T DSP1610,
 - Motorola DSP56001,
 - NEC mPD77016,
 - TI TMS320C51.
- Hand-crafted assembly code is compared to the compiler-generated code.
- Result: overhead between 100% and 1000% of compiler-generated code is typical !

Compiling for DSPs

- Code quality of traditional high-level language compilers is not satisfactory.
- Thus: Assembly programming.



Case Study: Infineon TriCore

C code (FIR Filter):

```

int i,j,sum;
for (i=0;i<N-M;i++) {
    sum=0;
    for (j=0;j<M;j++) {
        sum+=array1[i+j]*coeff[j];
    }
    output[i]=sum>>15;
}
  
```

Compiler-generated code (gcc):

```

.L21: add %d15,%d3,%d1
      addsc.a %a15,%a4,%d15,1
      addsc.a %a2,%a5,%d1,1
      mov %d4,49
      ld.h %d0,[%a15]0
      ld.h %d15,[%a2]0
      madd %d2,%d2,%d0,%d15
      add %d1,%d1,1
      jge %d4,%d1,.L21
  
```

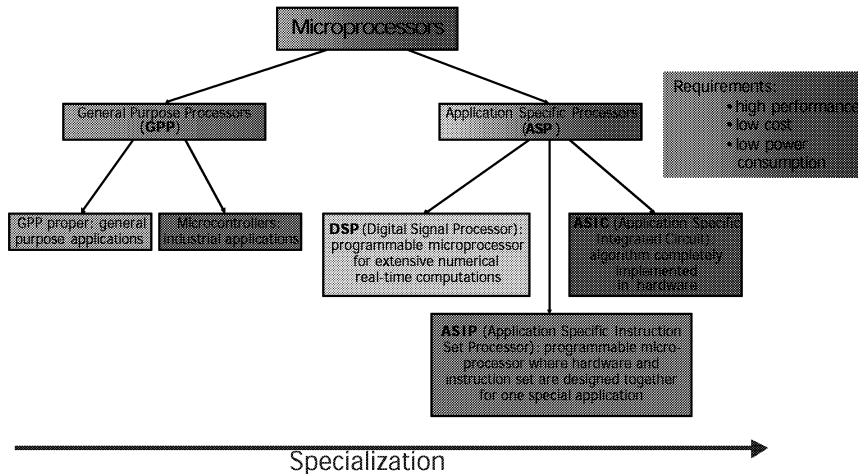
Hand-written code (inner loop):

```

_8: ld16.w d5, [a2+]
    ld16.w d4, [a3+]
    madd.h e8,e8,d5,d4u1,#0
    loop a7,_8
  
```

- ✓ Zero-Overhead Loops
- ✓ SIMD Instructions
- ✓ Auto-modify addressing
- **6 times faster!**
(execution in SRAM)

Classification of Microprocessors



Architectural Valuation

- [Campbell, Northrop Grumman Corporation, 1998]
- More efficient architectures will use less energy to complete the same task on the same generation CMOS solid state technology
- Power consumption $P = CV^2 f \Delta N$
 - C : Capacitance
 - V : CPU Core Voltage
 - f : CPU clock frequency
 - ΔN : Number of gates changing state

Architectural Valuation

■ Observations:

- Higher performance by increasing the clock frequency does not change the performance per power ratio

$$\frac{\text{Performance}}{\text{Power}} = \frac{\frac{O}{\text{Cycle}} \cdot \frac{\text{Cycle}}{\text{Time}}}{CV^2 f \Delta N} = \frac{Of}{CV^2 f \Delta N} = \frac{O}{CV^2 \Delta N}$$

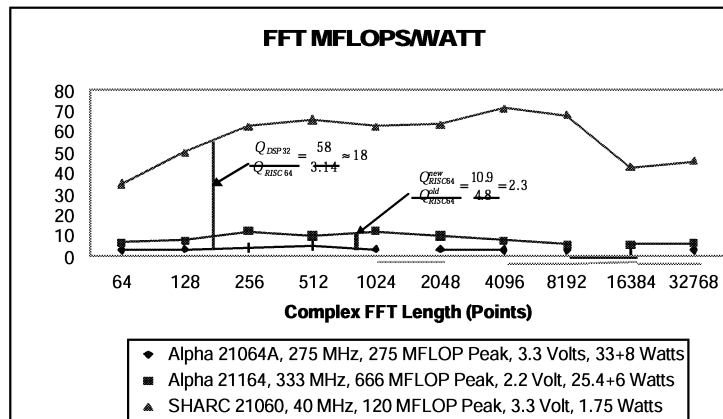
- A voltage decrease improves performance per power non-linearly

$$\frac{\text{Performance}}{\text{Power}} = \frac{O}{CV^2 f \Delta N}$$

Architectural Valuation

- Architectural specialization as a measure for how well the architecture fits a given target application.
- Estimation of architectural specialization:
Performance per power.

Comparison of Performance Per Power Ratios



Program Representations

- Abstract Syntax Tree (AST)
- Static Single Assignment (SSA)
- Control Flow Graph (CFG), Call Graph (CG) and Interprocedural Control Flow Graph (ICFG)
- Data Dependence Graph (DDG)
- Low-Level Intermediate Representation:
Abstract Machine Code / Register Transfer Languages

High-Level and Low-Level IRs

- High-level intermediate representation: close to source level. Typically centered around source language constructs. Constructs: implicit memory addressing, expression trees, for- while-, switch-statements, etc.
- Low-level intermediate representation: close to machine level. Typically centered around basic entities that specify properties of machine operations.
- Most program representations can be defined at high-level and at low-level.

IR Levels

| <i>High-Level</i> | <i>Medium-Level</i> | <i>Low-Level</i> |
|------------------------|--|---|
| t1 = a[i][j+3]; | t1 = addr(a); t2 = i*20; t3 = j+3; t4 = t2+t3; t5 = 4*t4; t6 = t1+t5; t7 = *t6; | v1 = fp-216; v2 = [fp-4]; v3 = v2*20; v4 = [fp-8]; v5 = v4+3; v6 = v3+v5; v7 = 4*v6; v8 = [v1+v7]; |

Assumption: Input language C, a declared as int a[10][20];

IR Levels

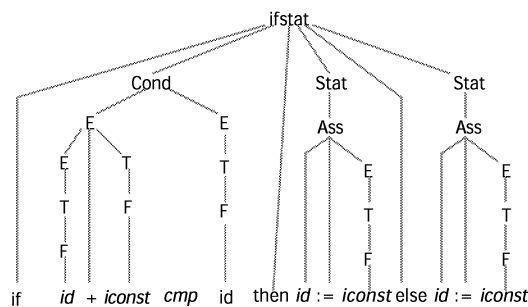
■ High-level IRs:

- abstract syntax tree
 - control flow graph and data dependence graph used for array dependence analysis and high-level code transformations

■ Low-level IRs:

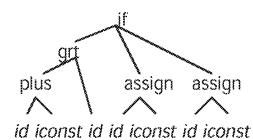
- abstract machine code (medium-level)
- direct representation of target machine instructions
- register transfer language (machine-independent representation for machine-specific instructions)
- control flow graph and data dependence graph used for machine-level dependence analysis and low-level code transformations

Decorated Abstract Syntax Tree



concrete syntax tree

abstract syntax tree



Call Graph

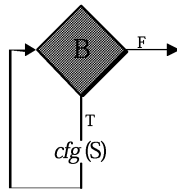
- There is a node for the main procedure – being the entry node of the program – and a node for each procedure or function declared in the program.
- The nodes are marked with the procedure names.
- There is an edge between the node for a procedure p to the node of procedure q , if there is a call to q inside of p .

Control Flow Graph

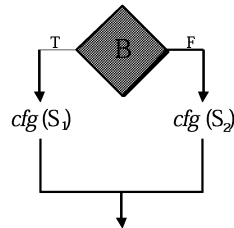
- The control flow graph of a procedure is a directed graph $G_C = (N_C, E_C, n_A, n_W)$ with node and edge labels. For each instruction i of the procedure there is a node n_i that is marked by i . The edges (n, m, l) denote the control flow of the procedure: $l \in \{T, F, \emptyset\}$ is the edge label. The nodes for composed statements are shown on the next slide. Edges belonging to unconditional branches lead from the node of the branch to the branch destination. The node n_A is the uniquely determined entry point in the procedure; it belongs to the first instruction to be executed. n_W denotes the end node that is reached by any path through the control flow graph.
- Nodes with more than one predecessor are called joins and nodes with more than one successor are called forks.

Control Flow Graph – Composed Statements

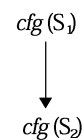
$cfg(\text{while } B \text{ do } S \text{ od}) =$



$cfg(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) =$



$cfg(S_1; S_2) =$



Basic Block Graph

- A basic block in a control flow graph is a path of maximal length which has no joins except at the beginning and no forks except possibly at the end.
- The basic block graph $G_B = (N_B, E_B, b_A, b_W)$ of a control flow graph $G_C = (N_C, E_C, n_A, n_W)$ is formed from G_C by combining each basic block into a node. Edges of G_C leading into the first node of a basic block lead to the node of that basic block in G_B . Edges of G_C leaving the last node of a basic block lead out of the node of that basic block in G_B . The node b_A denotes the uniquely determined entry block of the procedure; b_W denotes the exit block that is reached at the end of any path through the procedure.

Interprocedural Control Flow Graph

The interprocedural control flow graph consists of three parts:

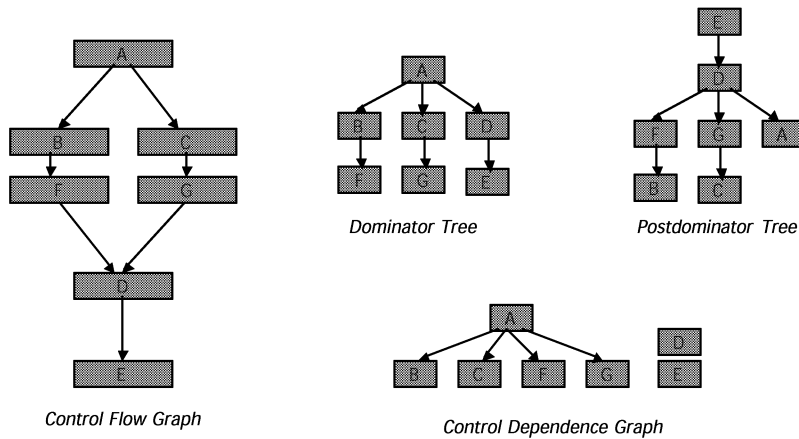
1. Call graph whose nodes are meta-nodes containing basic block graphs.
2. Basic block graph for each procedure in the program.
3. Ordered list of instructions for each block in the basic block graph of each procedure.

The ICFG describes the control flow of a program completely.

Control Dependence Graph

- Operation i dominates an operation j , if i appears on every path from the entry node of the procedure to j . Each operation dominates itself.
- Operation j postdominates i , if j appears on every path from i to the exit node of the procedure.
- Given a control flow graph $G_C = (N_C, E_C, n_A, n_W)$. Node $m \in N_C$ is control dependent on $n \in N_C$ if
 - (n, a) is an edge of the control flow graph
 - m does not postdominate n
 - there is a path from n, a, \dots, m so that m postdominates all nodes between n and m .
- The dominance frontier of a node x of the CFG (or BBG) is the set of all nodes y so that x dominates an immediate predecessor of y , but not y itself.

Control Dependence



{D} is dominance frontier of B, C, F, G

AbsInt
Angewandte Informatik



35

Data Dependence Graph Low Level View

- Let G_C be a control flow graph. Its data dependence graph is a directed graph $G_D = (N_D, E_D)$ with node and edge labels whose nodes are labeled by the operations of the procedure. An edge runs from the node of an operation i to the node of an operation j , if i has to be executed before j , i.e. if there is a path from i to j in the control flow graph and if
 - i defines a resource r , j uses it and the path from i to j does not contain other definitions of r (true dependence, RAW): $(i, j, r, t) \in E_D$
 - i uses a resource, j defines it and the path from i to j does not contain any definitions of r (anti dependence, WAR): $(i, j, r, a) \in E_D$
 - i and j define the same resource and the path from i to j does not contain any uses nor definitions of r (output dependence, WAW): $(i, j, r, o) \in E_D$

(1) $r1 = r2 * r3;$
 \swarrow (1, 2, r1, t)
 (2) $r5 = r1 + r1;$

AbsInt
Angewandte Informatik



36

Data Dependence Graph High Level View

- Iteration distance: Number of loop iterations between two dependent instruction instances (0 for intra-iteration dependences).
- Delay: Minimal number of clock cycles between the issuing of two dependent operation instances.
- Edges of the DDG are labeled with $(itDist, delay, type)$.
- The delay for a dependence $a \rightarrow b$ depends on the latencies of a and b and the type of the dependence:
 - true dependence (def-use): $latency(a)$
 - anti dependence (use-def): $1 - latency(b)$
 - output dependence (def-def): $1 + latency(a) - latency(b)$

```

for (i=2; i<100; i++) {
  (a) A[i]=B[i]+C[i];
      (a, b, 2, 1, t)
  (b) D[i]=A[i-2];
}

```

Life Range and Register Interference

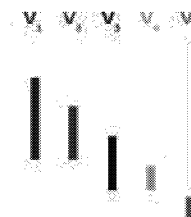
- A symbolic register (a variable) r is live at a program point p , if there is a program path from the entry node of the procedure to p that contains a definition of r and there is a path from p to a use of r on which r is not defined. The life range of a symbolic register r is the set of program points at which r is live.
- Two life ranges of symbolic registers interfere, if one of them is defined during the life range of the other. The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.

Life Range and Register Interference

```

a)  $v_1 = \text{Mem}[0xAFA0]$ 
b)  $v_2 = \text{Mem}[0xAFC0]$ 
c)  $v_3 = v_1 + v_2$ 
d)  $v_4 = v_1 * v_2$ 
e)  $v_5 = v_3 + v_4$ 
f) return  $v_5$ 

```



Global Register Allocation

- 'Global' means Register allocation across whole procedures / programs.
- Symbolic registers: hold intermediate results and modified variables.
- A definition of a symbolic register is the computation of an intermediate result of the modification of a variable
- Two symbolic registers collide/interfere, if their contents are live at the same time, ie if one of them is defined during the life range of the other.
- Colliding symbolic registers cannot be allocated to the same real register.
- Goal: allocate the (unbounded number of) symbolic registers to the fixed number of physical registers without collision.
- The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.

Register Allocation by Graph Coloring

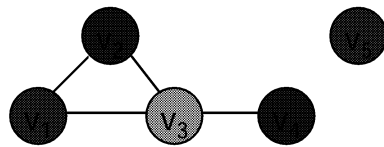
- If k physical registers are available, the k -coloring problem must be solved.
- NP-complete for $k > 2$ -> Use heuristics
- Algorithm:
 - If G contains a node n with degree $< k$:
 - n and its neighbors can be colored with different colors
 - Remove n from G , decreasing the size of G .
 - G is k -colorable, if we arrive at the empty graph.
 - If G is not empty and there exists no node with degree $< k$:
 - use heuristics to select one node to remove (spilling)
 - modify program inserting spills at definitions and loads at uses
 - reflect changes in graph.

Heuristics for Node Removal

- Degree of the node: high degree causes many deletions of edges.
- Costs of spilling.

Example: Graph Coloring

```
v1=Mem[0xafa0]
v2=Mem[0xafc0]
v3=v1+v2
v4=v1*v2
v5=v3+v4
return v5
```



Instruction Scheduling

- Definition: Reordering an operation sequence in order to exploit instruction-level parallelism and to minimize pipeline stalls.

- Complexity: NP-complete.

- Terminology:

- An operation is a basic machine operation like add, sub, etc.
- An instruction is a set of machine operations that are issued simultaneously (cf. VLIW).

Example:

```
r3=r1+r4, r3=r10+r14, r11=dm(i6,m6), r12=pm(i15,m15);
```


Instruction Scheduling

- Scope of instruction scheduling:
 - local acyclic instruction scheduling: reordering operations inside basic blocks. Standard technique: list scheduling.
 - global acyclic instruction scheduling: reordering operations across basic block boundaries but not across loop boundaries. Standard technique: trace scheduling.
 - cyclic instruction scheduling: reordering operations across loop boundaries. Standard technique: software pipelining.

List Scheduling

```
SET data_ready;  
int cycle=0;
```

Insert operations without predecessors in the data dependence graph into the data_ready set.

```
while (data_ready != ~) do {  
    cycle = cycle+1;
```

Choose operations from data_ready in priority order and insert them into the current cycle, until data_ready is empty or the insertion leads to a resource conflict.

Insert all operations into data_ready that can be scheduled in the next cycle without violating data dependences.

```
}
```

List Scheduling

- The priority in which operations from the data ready set are chosen is determined by heuristics.
- Common heuristics: highest-level-first heuristics.
 - The priority of each operation is the length of the longest path in the data dependence graph starting from this operation.
- Code quality: often within 10% from local optimum (inside basic blocks)

Exposing More Instruction-Level Parallelism

- Degree of instruction-level parallelism inside basic blocks is limited – typically to 2.
- The available parallelism in contemporary microprocessors grows. Better exploitation by
 - scheduling sequences of consecutive basic blocks
 - scheduling entire loops
 - speculation
- Two kinds of speculation:
 - dynamic: based on hardware branch prediction. In case of mispredicted branch forgetting or undoing effects of speculatively executed instructions.
 - static: generating compensation code for "speculatively placed" instructions.
 - Recently: Static data and control speculation with hardware support to deal with mispredictions (Intel IA-64).

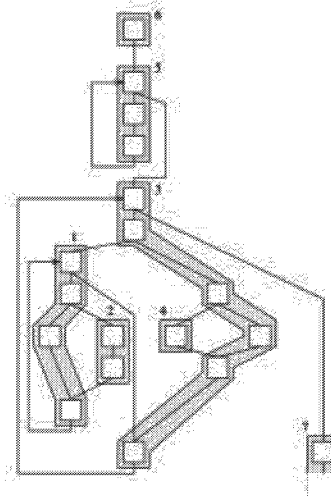
Trace and Superblock Scheduling

- Profile-directed scheduling of sequences of consecutive basic blocks (traces, superblocks)
- Program annotated with profiling information: each branch of a conditional associated with a relative frequency, or each basic block annotated with its execution frequency.
- Profile used to identify frequently taken paths.
- Idea: if a sequence of basic blocks is often executed one after another, they should be optimized jointly.
- Frequently taken paths (traces) are optimized at the cost of less frequently taken traces.

Trace Scheduling

- A trace is a sequence of consecutive basic blocks that are frequently executed one after another. A trace is never extended across loop boundaries.
- The control flow graph of a procedure is partitioned into a disjoint set of traces.
- The traces are formed in the order of decreasing execution frequency by repeatedly
 - selecting the basic block with highest execution frequency that has not been assigned to a trace yet as a seed of the new trace
 - joining predecessors and successors to that trace in decreasing frequency order until the frequency falls below a given threshold, or a loop boundary is reached.

A Partitioning into Traces



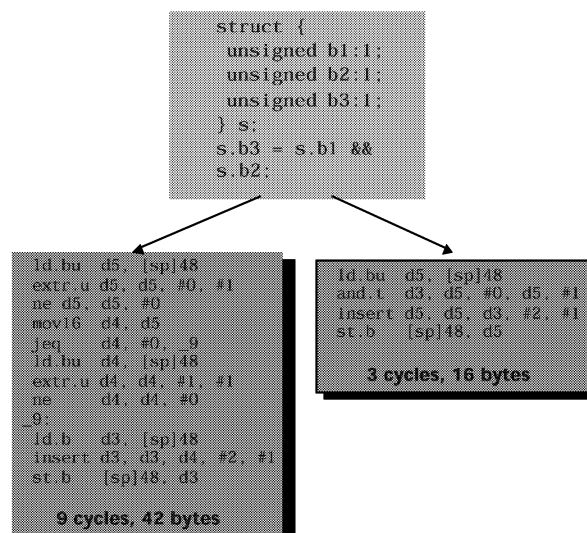
Trace Scheduling

- List scheduling of a trace leads to problems: the semantics may be destroyed by
 - code motion past side exists from the trace
 - code motion past side entrances to the trace
- Consequence: compensation code has to be inserted on off-trace paths.
- Problems of compensation code:
 - code growth
 - exceptions raised by compensation code

Superblock Scheduling

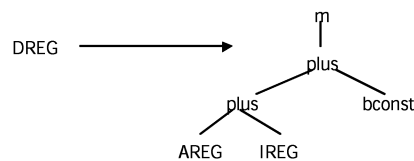
- Goal: Avoiding code motion past side entrances by tail duplication: copying code starting with side entrance and redirecting the branches.
- Superblock formation:
 - starts with a trace
 - produces a trace without side entrances
- Compensation code only for code motion past side exits.

Impact of Code Selection



Generating Code Selectors

- Machine grammar: regular tree grammar;
 - terminals: operators from the program representation
 - non-terminals: represent storage resources
 - often ambiguous
 - each rule has associated costs
 - factorization, e.g. of addressing modes reduces size.



Generating Code Selectors

- A machine grammar enables IR trees for expressions to be derived. The derivation tree for an IR tree represents one possibility of generating code for the IR tree.
- The generated code selector
 - parses intermediate representations of programs
 - computes derivations according to the machine grammar, each corresponding to a sequence of machine instructions
 - has to select a cheapest derivation, corresponding to the (locally) cheapest code sequence
 - may compute costs in states or use dynamic programming

Tree Languages

- An alphabet with arity is a finite set Σ of operators together with a function $\rho: \Sigma \rightarrow \mathbb{N}_0$ (arity).
- $\Sigma_k = \{a \in \Sigma \mid \rho(a) = k\}$
- The homogeneous tree language over Σ is the following inductively defined set $T(\Sigma)$:
 - $a \in T(\Sigma)$ for all $a \in \Sigma_0$
 - Are $b_1, b_2, \dots, b_k \in T(\Sigma)$ and is $f \in \Sigma_k$, then $f(b_1, b_2, \dots, b_k) \in T(\Sigma)$
- Example: $\Sigma = \{a, \text{cons}, \text{nil}\}$; $\rho(a) = \rho(\text{nil}) = 0$, $\rho(\text{cons}) = 2$
Some trees over Σ : a , $\text{cons}(\text{nil}, \text{nil})$, $\text{cons}(\text{cons}(a, \text{nil}), \text{nil})$

Tree Grammars

- A Regular Tree Grammar is a grammar $G = (N, \Sigma, P, S)$ where
 - N is a finite set of non-terminals
 - Σ is a finite alphabet with arity of terminals (operators labeling nodes)
 - P is a finite set of rules $X \rightarrow s$ where $X \in N$ and $s \in T(\Sigma \cup N)$
 - $S \in N$ is the start symbol

Example: Machine Grammar

$$G_m = (N_m, \Sigma, P_m, REG)$$

$$\Sigma = \{const, m, plus, REG\}$$

$$\text{where } \rho(const) = 0; \rho(m) = 1, \rho(plus) = 2,$$

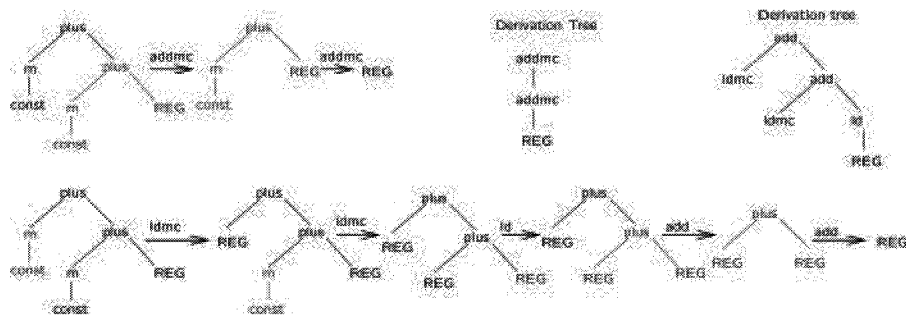
$$N_m = \{REG\}$$

$$P_m = \{ \begin{array}{ll} addmc : REG \rightarrow plus(m(const), REG), \\ addm : REG \rightarrow plus(m(REG), REG), \\ add : REG \rightarrow plus(REG, REG), \\ ldmc : REG \rightarrow m(const), \\ ldc : REG \rightarrow const, \\ ld : REG \rightarrow REG \end{array} \}$$

Generating Code Selectors

- Let $G = (N, \Sigma, P, S)$ be a regular tree grammar. From G a non-deterministic finite tree automaton (NTFA) is generated whose computations correspond to the derivation trees wrt G . In a second step, the non-deterministic finite tree automaton is transformed in a DFTA.
- Starting with the leaves of the input tree, the NFTA guesses the correct right-hand sides of grammar rules and checks that the right-hand sides really fit. This way, the input tree is covered by the right-hand side of the grammar rules.
- Code selector generators are, e.g., BEG [Emm89], lburg [FraHan95], ...

Example: Generating Code by Computing the Derivation Tree



AbsInt
Angewandte Informatik



61

Excerpt from an Iburg Specification

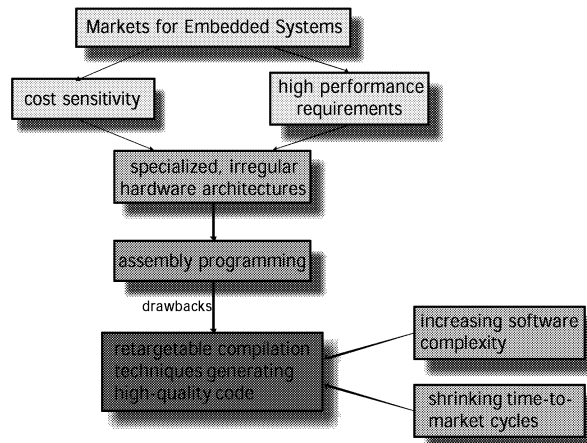
| | | |
|-----------------------------|----------------------------|----------|
| rc1: reg | "%0" | |
| rc1: con1 | "%0" | |
| reg: con | "%c = %0\n" | 1 |
| reg: ADDI4(reg, rc1) | "%c = %0 + %1\n" | 1 |
| reg: ADDU4(reg, rc1) | "%c = %0 + %1\n" | 1 |
| reg: ADDP4(reg, rc1) | "%c = %0 + %1\n" | 1 |
| reg: ADDF4(reg, rc1) | "f%c = f%0 + f%1\n" | 1 |
| reg: ADDF5(reg, rc1) | "f%c = f%0 + f%1\n" | 1 |
| reg: SUBI4(reg, rc1) | "%c = %0 - %1\n" | 1 |
| reg: SUBU4(reg, rc1) | "%c = %0 - %1\n" | 1 |
| reg: SUBP4(reg, rc1) | "%c = %0 - %1\n" | 1 |
| reg: SUBF4(reg, rc1) | "f%c = f%0 - f%1\n" | 1 |
| reg: SUBF5(reg, rc1) | "f%c = f%0 - f%1\n" | 1 |

AbsInt
Angewandte Informatik



62

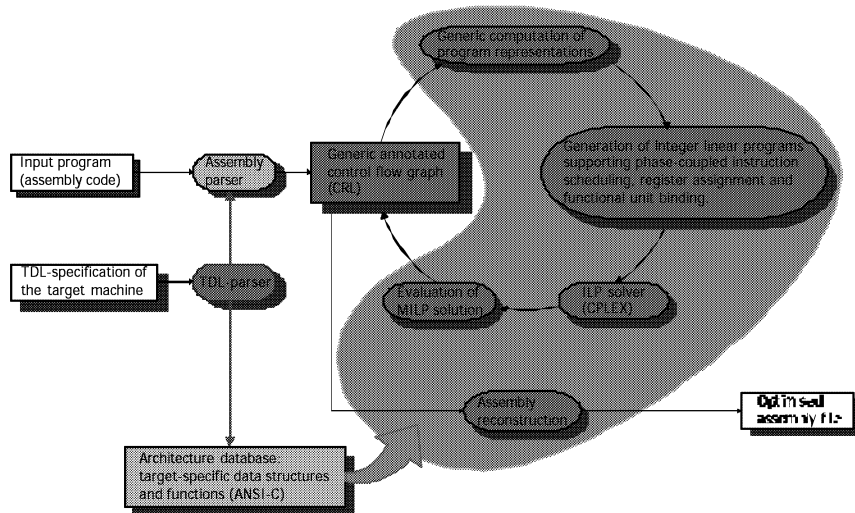
Retargetable Compilation



Retargetable Compilation

- Code selector generators: iburg, twig, olive,...
- Retargetable optimizers:
 - PO [DaFra80,DaFra84], vpo[BeDa88,BeDa94]
- Retargetable compilers:
 - lcc [FraHa95], GNU gcc [Stall98], CoSy [COSY], Trimaran [Tri98]
- Retargetable compilers/optimizers for embedded processors:
 - RECORD [Le97], CHESS [FPF95], PROPAN [DK00], Express [HTGDN99], ...

The PROPAN System



Architecture Description Languages

- Behavioral: specification of the instruction set, focussing on instruction semantics.
- Structural: close to gate level; specification of hardware modules with their interconnections.
- Mixed: this type is often used for code generation and optimization.
- Hardware description languages: VHDL, Verilog, MIMOLA [No87], nML [FaVPFre95], SALTO [BChaRSe97], ISDL [Ha98], EXPRESSION [HGGKDN99], CSDL [DaRa98], TDL [Kaest03].

Example: TDL (Target Description Language)

- Developed within the PROPAN framework
- Design goals:
 - Generating an assembly parser
 - Supporting instruction scheduling, register assignment and resource allocation
 - Easy extendability
 - Specification of irregular hardware properties in a way that supports:
 - generic incorporation into ILP-based optimisations
 - generic program analyses
 - Semantical analysis of the specification itself

TDL Descriptions

- Resource section: Declaration of the relevant hardware resources with their properties.
- Instruction set section: Definition of the instruction set in the form of an attribute grammar.
- Constraint section: Logical constraints that have to be respected to preserve correctness during code transformations → Support for architectural irregularities
- Assembly section: Syntactic details of the assembly language

Resource Spezifikation of Analog Devices SHARC ADSP-2106x

Resources-Section

```
FuncUnit ALU 1;  
Register ireg "r%d" [0:15] size=40, type=signed<32>;  
ResourceClass iregA {ireg[0:3]};  
ResourceClass iregB {ireg[4:7]};  
ResourceClass iregC {ireg[8:11]};  
ResourceClass iregD {ireg[12:15]};  
  
DefineAttribute replacement {"LRU", "FIFO"} associated to Cache;  
  
Cache InstrCache assoc=2, sets=16, linesize=48, type=instr,  
replacement=LRU;
```



69

Specification of the Instruction Set

Instructionset-Section

```
DefineOp SharcAdd "%s = %s + %s" {  
    dst1 = "$1" in {ireg}, src1 = "$2" in {ireg},  
    src2 = "$3" in {ireg}},  
    {ALU(exectime=1, latency=1, slots=0)};  
    {dst1 := src1 + src2};
```



70

Bibliography

■ Compiler Design:

- [ASU86] Aho, Sethi, Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.
- [WiMa95] Wilhelm, Maurer. Compiler Design. Addison-Wesley, 1995.
- [M97] Muchnick. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, 1997.
- [SS03] The compiler design handbook: Optimizations and Machine Code Generation. Ed. Srikant, Shankar. CRC Press, 2003.

Bibliography

■ Code Selection:

- [AJ76] Aho, Johnson. Optimal Code Generation for Expression Trees. Journal of the ACM, vol 23, no 3, 1976.
- [GG78] Glanville, Graham. A new Method for Compiler Code Generation. Proceedings of the 5th ACM Symposium on Principles of Programming Languages, 1978.
- [AG85] Aho Ganapathi. Efficient Tree Pattern Matching: An Aid to Code Generation. Proceedings of the 12th ACM Symposium on Principles of Programming Languages, 1985.
- [FSW94] Ferdinand, Seidl, Wilhelm. Tree Automata for Code Selection. Acta Informatica, vol 31, 1994.

Bibliography

■ Register Allocation:

- [Cha82] Chaitin. Register Allocation and Spilling via Graph Coloring. Proceedings of the SIGPLAN'82 Symp. on Compiler Construction. SIGPLAN Notices, vol 17, no 6, 1982.
- [CH84] Chow, Hennessy. The Priority-Based Coloring Approach to Register Allocation. ACM Transactions on Programming Languages and Systems, vol. 12, no 4, 1990.
- [Bri92] Briggs. Register Allocation via Graph Coloring. Phd Thesis, Rice University, 1992.
- [BCT94] Briggs, Cooper, Torczon. Improvements to Graph Coloring Register Allocation. ACM Transactions on Programming Languages and Systems, vol 16, no 3, 1994.

Bibliography

■ Instruction Scheduling and Parallelization:

- [LDSM80] Landskov, Davidson, Shriver. Local Microcode Compaction Techniques. ACM Computing Surveys, vol 12, no 3, 1980.
- [Fis81] Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, vol 20, no 7, 1981.
- [AJLA95] Allan, Jones, Lee, Allan. Software Pipelining. ACM Computing Surveys, 1995.
- [R96] Rau. Iterative Modulo Scheduling. International Journal of Parallel Processing, vol 24, 1996.
- [LA00] Larsen, Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. ACM SIGPLAN Notices, 2000.

Bibliography

■ Retargetable Compilation and Optimization:

- [FraHan91b] Fraser, Hanson. A Retargetable Compiler for ANSI C. SIGPLAN Notices, vol 26, no 10, 1991.
- [FraHan95] Fraser, Hanson. A Retargetable C Compiler: Design And Implementation. Benjamin/Cummings Publishing Company, Inc., 1995.
- [DaFra80] Davidson, Fraser. The Design and Application of a Retargetable Peephole Optimizer. ACM Transactions on Programming Languages and Systems, vol 2, no 2, 1980.
- [DaFra84] Davidson, Fraser. Code Selection through Object Code Optimization. ACM Transactions on Programming Languages and Systems, vol 6, no 4, 1984.
- [BeDa88] Benitez, Davidson. A Portable Global Optimizer and Linker. Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, in SIGPLAN Notices, vol 23, no 7, 1988.
- [BeDa94] Benitez, Davidson. Target-Specific Global Code Improvement: Principles and Applications. Department of Computer Science, University of Virginia, 1994.
- [Sta98] Stallman. Using and Porting GNU CC. Free Software Foundation, 1988.
- [Tri98] TRIMARAN: An Infrastructure for Research in Instruction-Level Parallelism. <http://www.trimaran.org>.
- [COSY] Ace Associated Computer Experts. <http://www.ace.nl/products/cosy.htm>

Bibliography

■ Retargetable Compilation and Optimization for Embedded Processors:

- [CHES95] Lanneer, Van Praet, Kifli, Schoofs, Geurts, Thoen, Goossens. CHES: Retargetable Code Generation For Embedded DSP Processors. In [MaGo95]. Kluwer Academic Publishers, 1995.
- [MaGo95] Marwedel, Goossens, G. Code Generation for Embedded Processors. Kluwer Academic Publishers, 1995.
- [Le97] Leupers. Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997.
- [HTGDN99] Halambi, Tomiyama, Gruen, Dutt Nicolau. Automatic Software Toolkit Generation for Embedded Systems-on-Chip. Proceedings of the 1999 International Conference on VLSI and CAD (ICVC99), 1999.
- [DK00] Kästner. Retargetable Postpass Optimisation by Integer Linear Programming. Saarland University, 2000.
- [DK01] Kästner. ILP-based Approximations for Retargetable Code Optimization. Proceedings of the 5th International Conference on Optimization: Techniques and Applications, Hong Kong, 2001.

Bibliography

■ Architecture Description Languages:

- [Emm89] Emmelmann. BEG -- a Back End Generator. GMD Forschungsstelle an der Universitaet Karlsruhe, 1989.
- [LSU93] Lipsett, Schaefer, Ussery. VHDL: Hardware Description and Design. Kluwer Academic Publishers, 1993.
- [TM95] Thomas, Moorby. The Verilog Hardware Description Language. Kluwer Academic Publishers, 1995.
- [FaPraFre95] Fauth, Van Praet, Freericks. Describing Instruction Set Processors Using nML. Proceedings of the European Design and Test Conference. IEEE, 1995.
- [BCRS97] Bodin, Chamski, Rohou, Seznec. Functional Specification of SALTO. A Retargetable System for Assembly Language Transformation and Optimization, rev. 1.00 beta, INRIA, 1997.
- [RaFe97] Ramsey, Fernandez. Specifying Representations of Machine Instructions. ACM Transactions on Programming Languages and Systems, vol 19, no 3, 1997.
- [DaRa98] Davidson, Ramsey. Machine Descriptions to Build Tools for Embedded Systems. Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems. Springer LNCS, Volume 1474, 1998.
- [DK03] Kaestner. TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses. Proceedings of the Second ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, 2003.

Bibliography

■ Postpass Optimizations

- [DK00a] Kästner. A Retargetable System for Postpass Optimisations and Analyses. Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools, Montreal, 2000.
- [CF00] Ferdinand. Post Pass Code Compaction at the Assembly Level for C16x. Contact, vol 3, no 9, 2000.
- [BKCP03] De Bus, Kaestner, Chanet, Van Put, De Sutter. Post-Pass Compaction Techniques. Communications of the ACM, vol 46, no 8, 08/2003.

Bibliography

■ Program Analysis

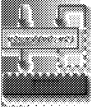
- [CC79] Cousot, Cousot. Systematic Design of Program Analysis Frameworks. Proceedings of the 6th ACM Symposium on Principles of Programming Languages POPL, 1979.
- [F97] Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD thesis, Saarland University, 1997.
- [NNH99] Nielson, Nielson, Hankin. Principles of Program Analysis. Springer, 1999.
- [M99] Florian Martin. Generation of Program Analyzers. PhD thesis, Saarland University, 1999.

Jaejin Lee

Seoul National University
jlee@cse.snu.ac.kr

Lecture #1

Jaejin Lee
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University
jlee@cse.snu.ac.kr
<http://aces.snu.ac.kr/~jlee>



Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University



1

Data Flow Analysis

- Most of the compiler optimizations require data flow analysis
- Propagate information from one point of the program to another

... = a + b

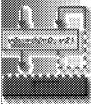
...

a = ... ?

...

... = a + b

Is there any intervening definition to a or b?



Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

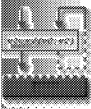


2

Basic Blocks

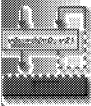
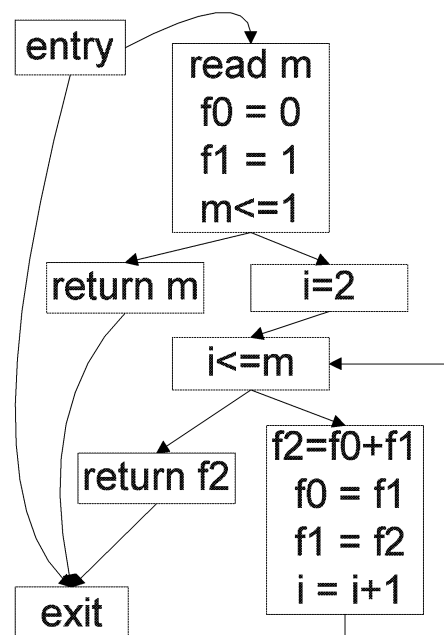
- A sequence of statements that is always entered at the beginning and exited at the end without halt or possibility of branching except at the end
- Algorithm?

| |
|----------------------|
| read m |
| f0 = 0 |
| f1 = 1 |
| if m<=1 goto L3 |
| i = 2 |
| L1: if i <=m goto L2 |
| return f2 |
| L2: f2 = f0 + f1 |
| f0 = f1 |
| f1 = f2 |
| i = i + 1 |
| goto L1 |
| L3: return m |



Control Flow Graphs

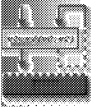
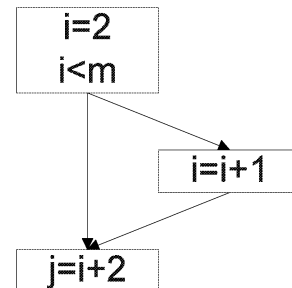
- Flow-of-control information
- Directed graph
- Node = basic block
- There is a directed edge from B1 to B2 if B2 can immediately follow B1 in some execution sequence



Data Flow Problem #1

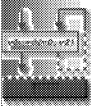
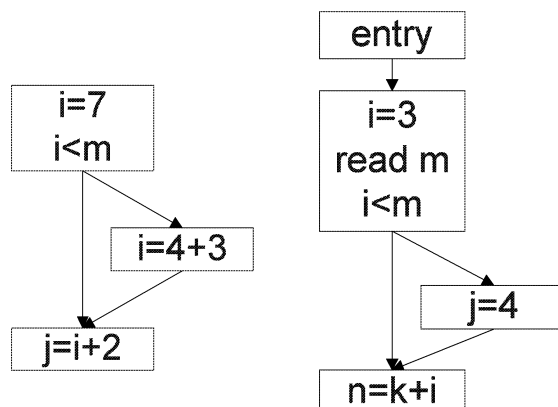
Reaching Definitions

- Which definitions of a variable may reach each use of the variable in a procedure?
- A definition d reaches a point p if there is a path from the point immediately following d to p , such that d is not killed along that path
- A statement defines a variable x if it may assign x a value
 - An assignment of x
 - A procedure that can access x
 - $*p = 3$ (p may points to x)



Usages of Reaching Definitions

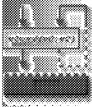
- Detect constant computation
- Detect uninitialized uses of variables



Effects of a Basic Block

- Compose effects of statements
- A basic block B
 - Generates definitions: $\text{Gen}[B]$
 - If a definition d in B reaches at the end of B , d is in $\text{Gen}[B]$
 - Kills definitions: $\text{Kill}[B]$
 - If a definition d in B never reaches at the end of B , d is in $\text{Kill}[B]$

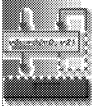
| | | | |
|----------------------------------|---------------------------|--|--------------------------------|
| $\text{Gen}[B] = \{d2, d3\}$ | $\text{Kill}[B] = \{d4\}$ | $\text{in}[B] = \{d0, d4\}$ $d1: x = a + b$ $d2: y = x + 3$ $d3: x = x + 4$ | $d0: z = 7$ $d4: y = 4 + 8$ |
| $\text{out}[B] = \{d0, d2, d3\}$ | | | |



Transfer Functions of a Basic Block

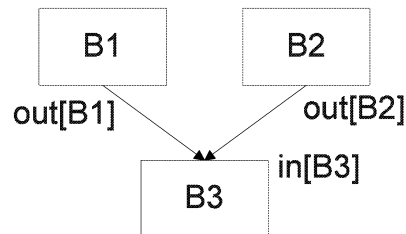
- The relationship between $\text{in}[B]$ and $\text{out}[B]$ of a basic block B
- $\text{Out}[B]$ is a function of $\text{in}[B]$

| | | | |
|---|---------------------------|--|--------------------------------|
| $\begin{aligned} \text{out}[B] &= f_B(\text{in}[B]) \\ &= \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B]) \end{aligned}$ | | | |
| $\text{Gen}[B] = \{d2, d3\}$ | $\text{Kill}[B] = \{d4\}$ | $\text{in}[B] = \{d0, d4\}$ $d1: x = a + b$ $d2: y = x + 3$ $d3: x = x + 4$ | $d0: z = 7$ $d4: y = 4 + 8$ |
| $\text{out}[B] = \{d0, d2, d3\}$ | | | |



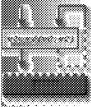
Effects of Control Flow

- Deal with incoming information from different predecessors of a basic block B



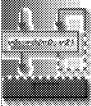
$$\text{in}[B] = \text{out}[P_1] \cup \text{out}[P_2] \cup \dots \cup \text{out}[P_n]$$

P_1, P_2, \dots, P_n are predecessors of B



Solving Reaching Definitions Problem

- Create data flow equations and solve for all the basic blocks in the CFG
 - $\text{out}[B] = \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B])$
 - $\text{in}[B] = \bigcup_{\text{pred. } P \text{ of } B} \text{out}[P]$
- Data flows forwards
- Use iterative algorithm to solve the equations
- Use bit vectors to represent sets (not necessarily)
 - One bit for each definition
 - \cap becomes bitwise and
 - \cup becomes bitwise or

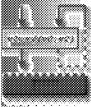


Iterative Solution

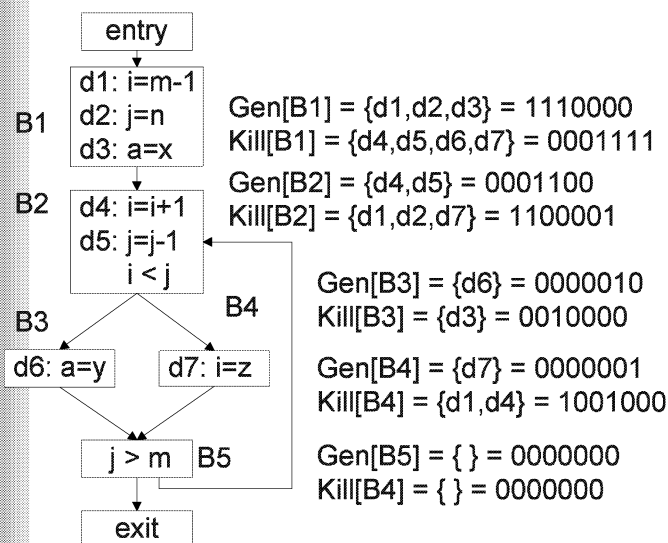
- Repeatedly visit all the nodes and update in and out
- Worklist algorithm?
 - excluding unreachable nodes

```

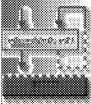
for each block B do
    in[B] =  $\emptyset$ 
    out[B] =  $\emptyset$  // or out[B] = Gen[B]
enddo
while changes to any out occur do
    for each block B do
        in[B] =  $\bigcup_{\text{pred. P of B}} \text{out}[P]$ 
        out[B] = Gen[B]  $\cup$  (in[B] - Kill[B])
    enddo
enddo
    
```



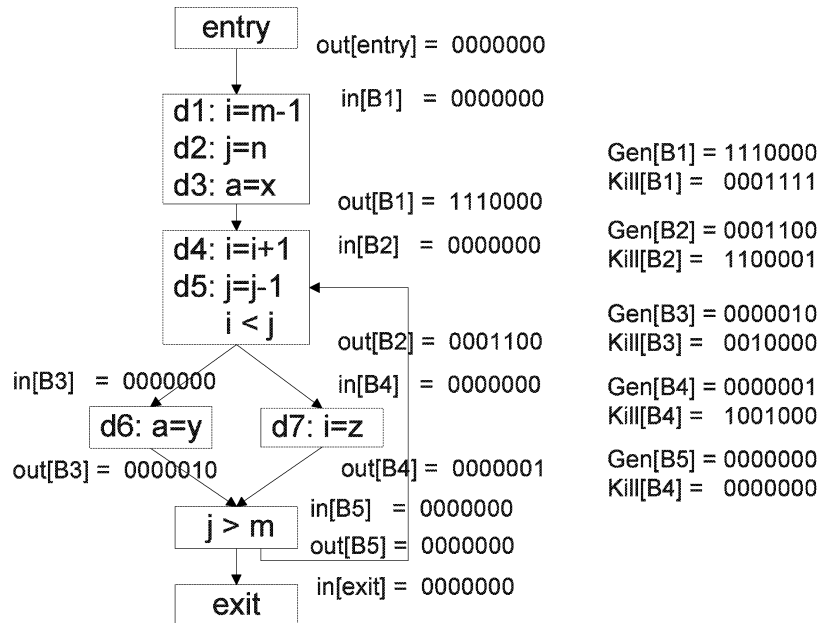
Example (Reaching Definitions)



out[B1]=Gen[B1] \cup (in[B1]-Kill[B1])
 out[B2]=Gen[B2] \cup (in[B2]-Kill[B2])
 out[B3]=Gen[B3] \cup (in[B3]-Kill[B3])
 out[B4]=Gen[B4] \cup (in[B4]-Kill[B4])
 out[B5]=Gen[B5] \cup (in[B5]-Kill[B5])
 in[B1]=out[entry]
 in[B2]=out[B1] \cup out[B5]
 in[B3]=out[B2]
 in[B4]=out[B2]
 in[B5]=out[B1] \cup out[B5]



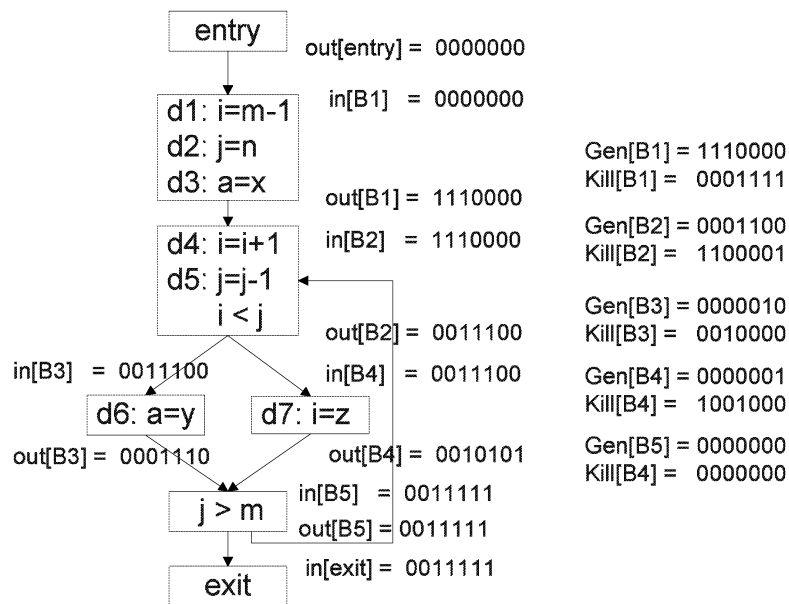
Example (Reaching Definitions)



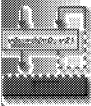
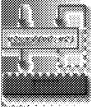
13



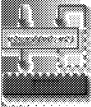
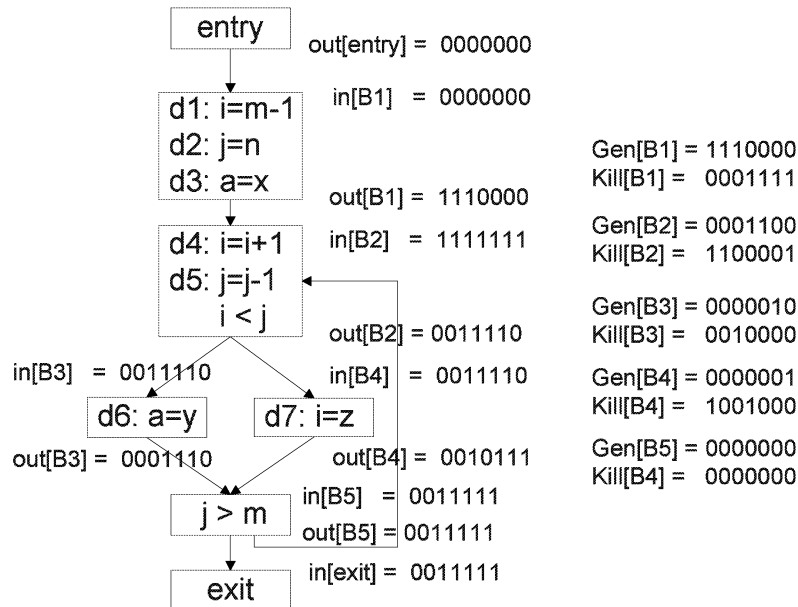
Example (Reaching Definitions)



14

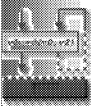
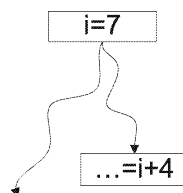


Example (Reaching Definitions)



Data Flow Problem #2 Live Variable Analysis

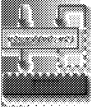
- A variable x is live at a point p if the value of x at p could be used along some path in the flow graph starting at p
- Used in
 - Register allocation
 - Code motion in loops
 - Elimination of useless assignments (dead code elimination)
- Data flows backwards



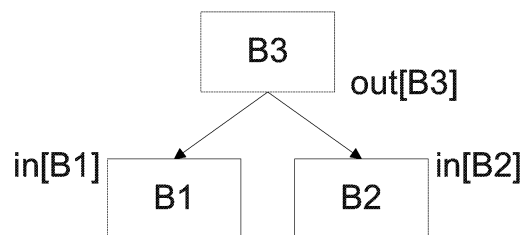
Transfer Functions of a Basic Block

- $\text{Def}[B]$: the set of variables definitely assigned values in B prior to any use of that variable in B
- $\text{Use}[B]$: the set of variables whose values may be used in B prior to any definition of the variable
 - Uses not covered by the definitions in B
- $\text{in}[B]$ is a function of $\text{out}[B]$

$$\begin{array}{l} \text{Use}[B] = \{a, b\} \\ \text{Def}[B] = \{x, y\} \end{array} \quad \boxed{\begin{array}{l} x = a + b \\ y = x + 3 \\ x = x + 4 \end{array}} \quad \begin{array}{l} \text{in}[B] = f_B(\text{out}[B]) \\ = \text{Use}[B] \cup (\text{out}[B] - \text{Def}[B]) \end{array}$$
$$\text{in}[B] = \{a, b, z\} \quad \text{out}[B] = \{x, z\}$$

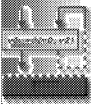


Effects of Control Flow

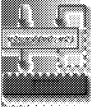
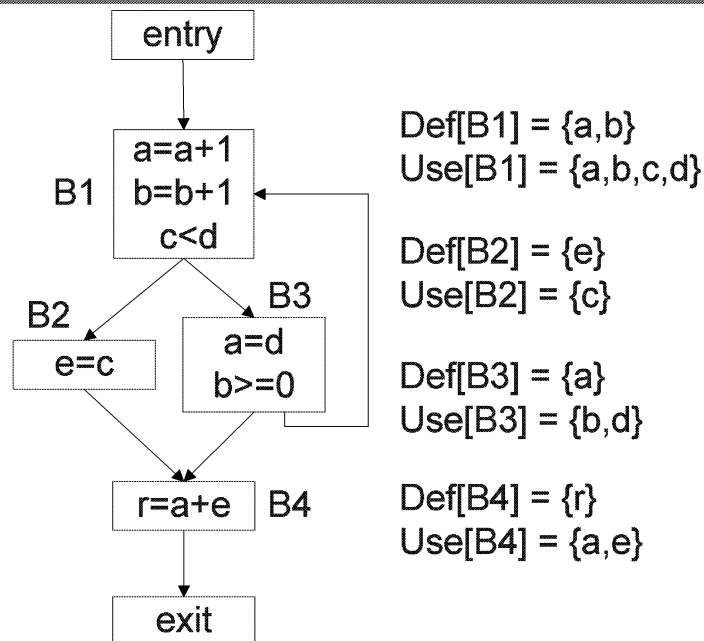


$$\text{out}[B] = \text{in}[P1] \cup \text{in}[P2] \cup \dots \cup \text{in}[Pn]$$

$P1, P2, \dots, Pn$ are successors of B



Example (Live Variable Analysis)

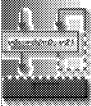


Iterative Solution for Live Variable Analysis

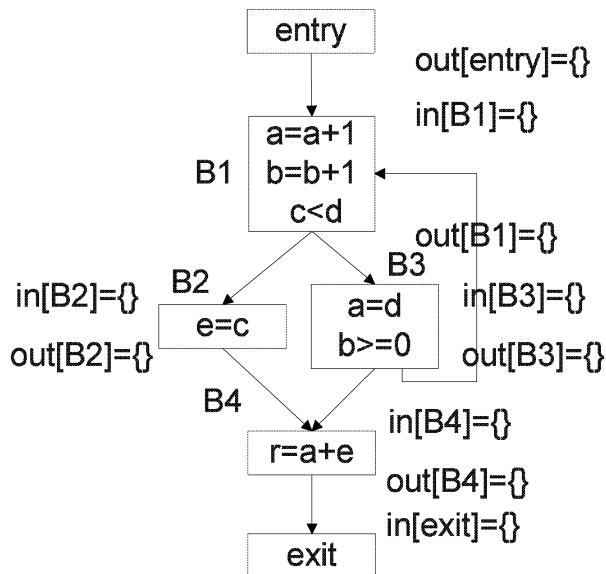
- Repeatedly visit all the nodes and update in and out
- Worklist algorithm?

```

for each block B do
  in [B] =  $\emptyset$  // or in[B] = Def[B]
  out[B] =  $\emptyset$ 
enddo
while changes to any in occur do
  for each block B do
    out[B] =  $\bigcup_{\text{succ. P of B}} \text{in}[P]$ 
    in[B] = Use[B]  $\cup$  (out[B] - Def[B])
  enddo
enddo
  
```



Example (Live Variable Analysis)

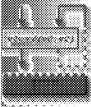


Def[B1] = {a,b}
Use[B1] = {a,b,c,d}

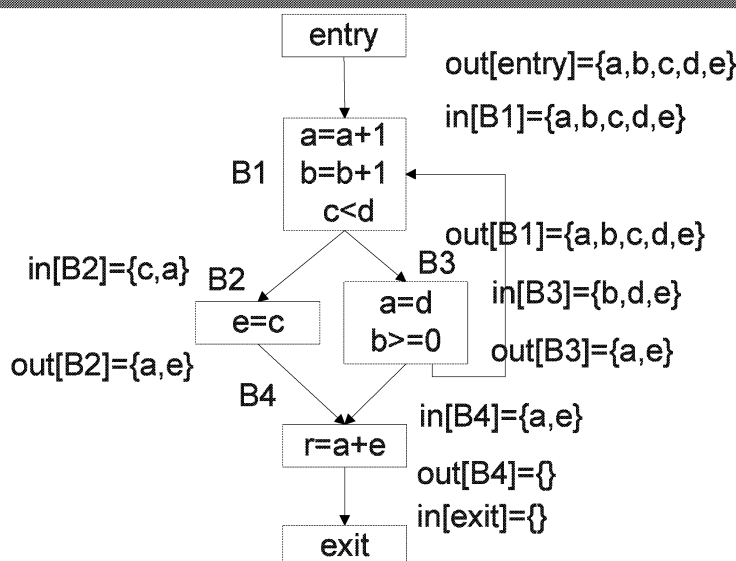
Def[B2] = {e}
Use[B2] = {c}

Def[B3] = {a}
Use[B3] = {b,d}

Def[B4] = {r}
Use[B4] = {a,e}



Example (Live Variable Analysis)



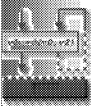
Def[B1] = {a,b}
Use[B1] = {a,b,c,d}

Def[B2] = {e}
Use[B2] = {c}

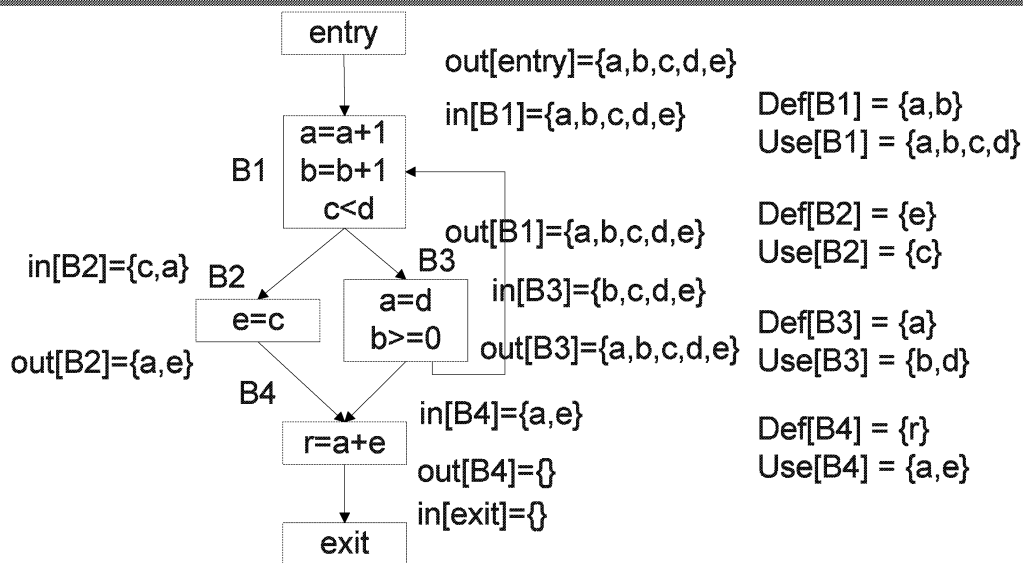
Def[B3] = {a}
Use[B3] = {b,d}

Def[B4] = {r}
Use[B4] = {a,e}

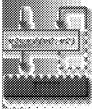
In the order of B4, B3, B2, B1



Example (Live Variable Analysis)

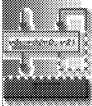


In the order of B4, B3, B2, B1



Some Notes on Iterative Solution

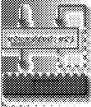
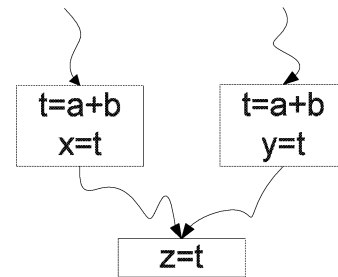
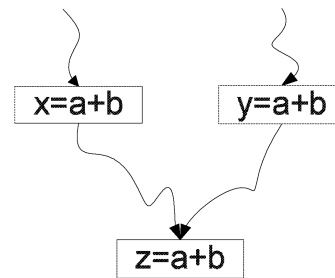
- The order of computation in an iterative method should follow the flow of information for speed
- Any solution to the dataflow equations is a conservative approximation
- The number of iterations:
 - By traversing in reverse postorder (in postorder, a node is not visited until all its depth-first spanning tree successors have been visited), the depth of the flow graph + 2
 - Flow graph depth: given a depth-first spanning tree for the graph, the largest number of retreating edges on any cycle-free path
 - Retreating edges: go from a node m to an ancestor of m in the depth-first spanning tree



Data Flow Problem #3

Available Expressions

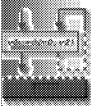
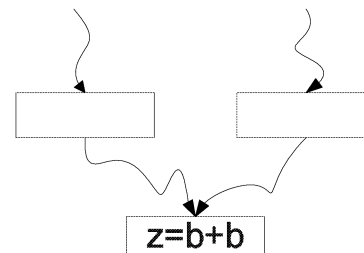
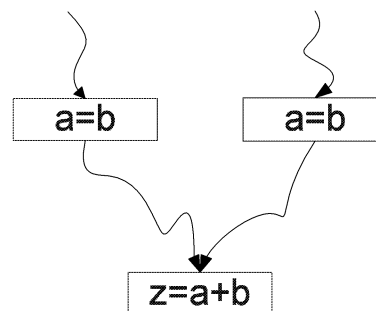
- An expression $x+y$ is available at a point p if every path from the initial node to p evaluates $x+y$, and after the last such evaluation prior to reaching p , there are no subsequent assignments to x or y
- Used for common subexpression elimination
- Data flow equations
 - Kill[B]: the set of expressions $a+b$ such that there is a definition of a or b in B
 - Gen[B]: the set of expressions $a+b$ computed in B at a position where neither a nor b are defined subsequently
 - $out[B] = Gen[B] \cup (in[B] - Kill[B])$
 - $In[B] = \bigcap_{pred. P \text{ of } B} out[P]$



Data Flow Problem #4

Copy Propagation

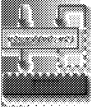
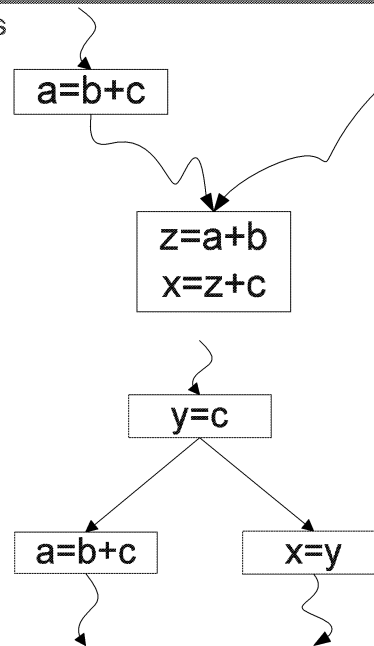
- A copy $a=b$ is available at point p if every path from the entry node to p has an occurrence of $a=b$, and there are no definitions of a or b after the last such occurrence
- Used for eliminating useless copy statements
- Data flow equations
 - Kill[B]: the set of copies $a=b$ such that either a or b is defined in B
 - Gen[B]: the set of copies $a=b$ appearing in B at a position where neither a nor b are redefined subsequently
 - $out[B] = Gen[B] \cup (in[B] - Kill[B])$
 - $In[B] = \bigcap_{pred. P \text{ of } B} out[P]$



Data Flow Problem #5

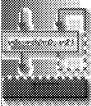
Upwards Exposed Uses

- A use of a variable is upwards exposed if there is some path in the CFG from the entry to the use which has no definition of the variable
- What uses of variables at particular points are reached by particular definitions
- The same as live variable analysis but the location at which the variables are used are recorded
 - Statement and use pairs: (s, x)
- Upwards exposed uses in B: uses of variables such that no prior definition of the variable occurs in B
- Data flow equations
 - $Kill[B]: \{ (s, x) \mid x \text{ is defined in } B \}$
 - $Gen[B]: \{ (s, x) \mid s \text{ is in } B \text{ and upwards exposed use of } x \text{ in } s \}$
 - $in[B] = Gen[B] \cup (out[B] - Kill[B])$
 - $out[B] = \bigcup_{succ. P \text{ of } B} in[P]$



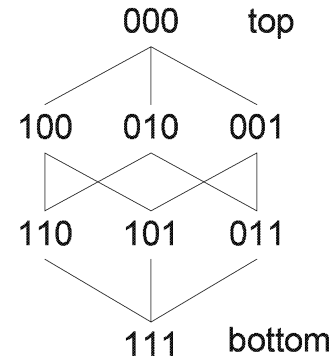
Transformations Using Data Flow Analysis

- Common subexpression elimination
- Constant propagation
- Copy propagation
- Dead code elimination
- Register allocation
- Partial redundancy elimination
 - bidirectional
- ...

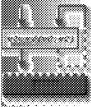


Data Flow Analysis Framework

- A data flow analysis is performed by operating on elements of a lattice
- Lattices
 - L : a set of elements
 - Meet (\sqcap) and join (\sqcup) operations
 - Closed under \sqcap and \sqcup
 - Commutative
 - Associative
 - \perp (bottom) element: for all $x \in L$, $x \sqcap \perp = \perp$
 - \top (top) element: for all $x \in L$, $x \sqcup \top = \top$
- \sqcap and \sqcup induce a partial order (\sqsubseteq) on the values
 - $x \sqsubseteq y$ iff $x \sqcap y = x$ or $x \sqsubseteq y$ iff $x \sqcup y = y$



Meet: and
Join: or



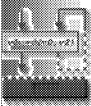
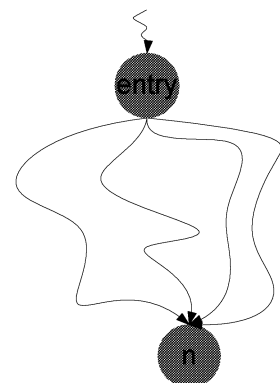
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

29



Meet-Over-all-Paths (MOP) Solutions

- We prefer the true, exact solution to the data-flow problem
 - However, there is no efficient way to tell exactly which paths are real and which are not
- MOP solution
 - $\text{Path}(B)$: the set of all paths from entry to a node B
 - P : any element of $\text{Path}(B)$
 - f_B : the transfer function of block B
 - f_P : the composition of the transfer functions encountered in following the path P
 - $f_P = f_{B_n} \circ f_{B_{n-1}} \circ \dots \circ f_{B_1}$
 - $B_1 = \text{entry}, B_2, \dots, B_n = B$ are the blocks in P
 - $\text{MOP}(B) = \sqcap_{P \in \text{Path}(B)} f_P(v_{\text{entry}})$



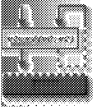
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

30



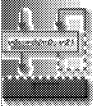
Maximal-Fixed-Point (MFP) Solutions

- Loops result in an infinite number of paths
 - impossible to get MOP
- A fixed point of a function $f:L \rightarrow L$ is an element $x \in L$ s.t. $f(x)=x$
- MFP solution: the solution to the data-flow equations that is maximal in the ordering
 - Compute meets early rather than at the end
 - Iterative algorithm



MFP and MOP

- A solution $S(B)$ is a conservative approximation if $S(B) \sqsubseteq \text{MOP}(B)$
- If all the transfer functions are monotone, the iterative algorithm produces the MFP solution but not necessarily the MOP solution
 - MFP solution \sqsubseteq MOP solution (legal)
 - $f:L \rightarrow L$, is monotone if for all x and y , $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$
 - Guarantee termination of the iterative algorithm
- If all the transfer functions are distributive,
 - MFP solution = MOP solution
 - $f:L \rightarrow L$, is distributive if for all x and y , $f(x \sqcap y) = f(x) \sqcap f(y)$
 - Distributivity implies monotonicity

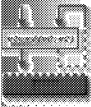


Iterative Data Flow Analysis Framework

■ Forward

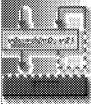
$$\begin{aligned} & \{ \text{Init } (\top \text{ or } \perp) \text{ for entry} \\ \text{in}[B] = & \{ \\ & \{ \bigcap_{P \in \text{Pred}(B)} \text{out}[P] \text{ otherwise} \\ \text{out}[B] = & f_B(\text{in}[B]) \end{aligned}$$

■ Backward

$$\begin{aligned} & \{ \text{Init } (\top \text{ or } \perp) \text{ for exit} \\ \text{out}(B) = & \{ \\ & \{ \bigcap_{P \in \text{Succ}(B)} \text{in}[P] \text{ otherwise} \\ \text{in}(B) = & f_B(\text{out}(B)) \end{aligned}$$


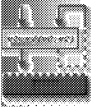
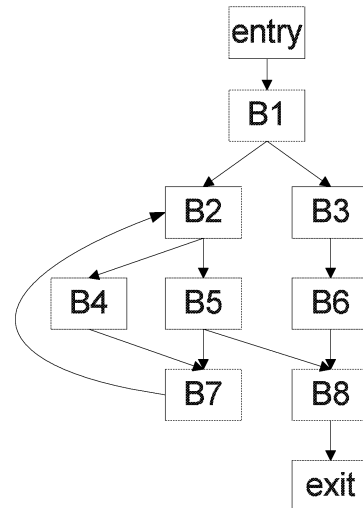
Control Flow Analysis

- Abstract syntax tree or other intermediate representations from the compiler front end provides relatively few hints about what the program does
- Discover hierarchical flow of control
 - Basic blocks
 - Control flow graph
 - Intervals
 - etc.
- Better data flow analyses are based on intervals



Dominators

- A node m dominates a node n ($m \text{ dom } n$), if every possible execution path from entry to n includes m
 - Reflexive: every node dominates itself
 - Antisymmetric: if $a \text{ dom } b$ and $b \text{ dom } a$, then $a=b$
 - Transitive: if $a \text{ dom } b$ and $b \text{ dom } c$, then $a \text{ dom } c$
 - $\text{dom}[m]$: the set of all dominators of m
- For $m \neq n$, m immediately dominates n iff $m \text{ dom } n$ and there does not exist a node p such that $p \neq m$ and $p \neq n$ for which $m \text{ dom } p$ and $p \text{ dom } n$

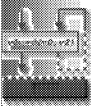


Computing Dominators

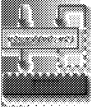
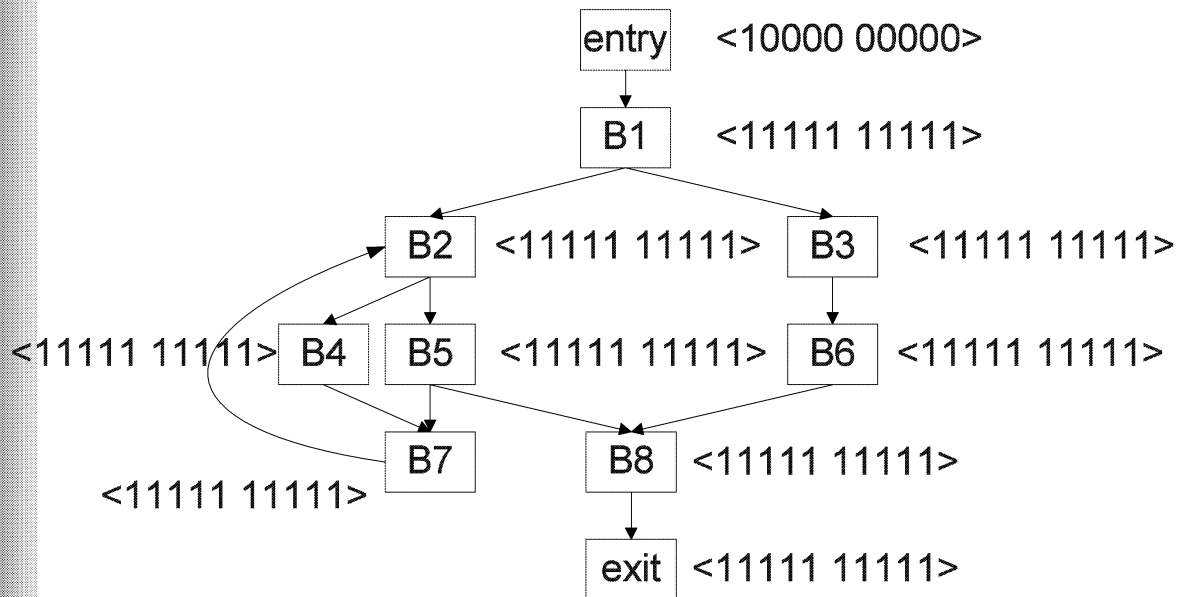
- A node m dominates a node n iff,
 - $m=n$ or
 - m is the unique immediate predecessor of n or
 - n has more than one immediate predecessor and for all immediate predecessors p of n , $p \neq m$ and $m \text{ dom } p$
 - $\text{dom}[n] = \{n\} \cup (\cap_{p \in \text{pred}[n]} \text{dom}[p])$
- Iterative algorithm


```

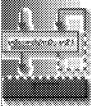
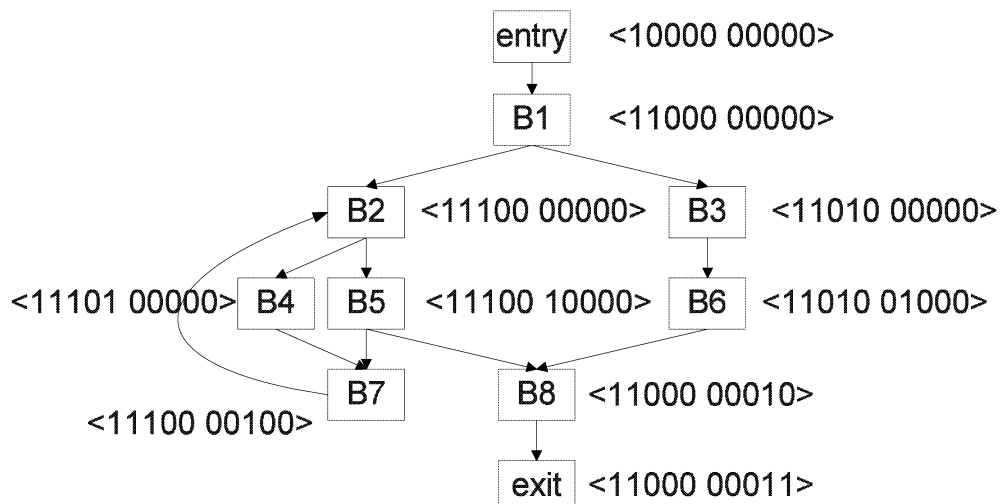
dom[entry] = {entry}
for each block B in N - {entry} do
    dom[B] = N
enddo
while changes to
    any dominators occur do
        for each block B in N - {entry} do
            for each p in pred[B] do
                dom[B] = dom[B] ∩ dom[p]
            enddo
            dom[B] = dom[B] ∪ {B}
        enddo
    enddo
enddo
      
```



Example (Computing Dominators)



Example (Computing Dominators)



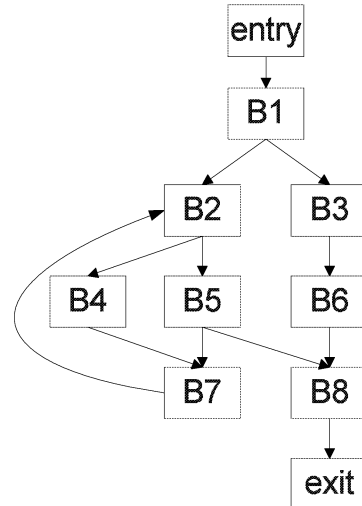
More on Dominators

■ Strict dominators

- M strictly dominates n if m dominates n and $m \neq n$

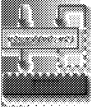
■ Postdominators

- A node p postdominates a node q if every possible execution path from q to exit includes p
- $p \text{ dom } q$ in the flow graph in which the direction of all the edges reversed and entry and exit nodes are interchanged



■ Back edges

- The head dominates its tail



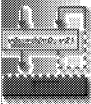
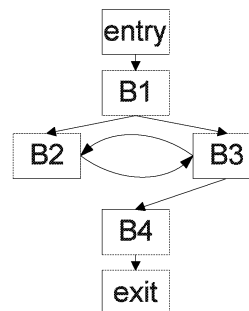
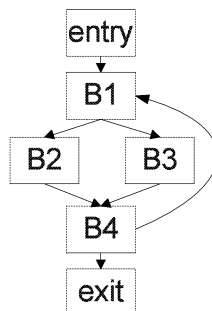
Natural Loops

■ Given a back edge (m,n) , the natural loop of (m,n) is the subgraph consisting of

- the set of nodes containing n and all the nodes from which m can be reached in the flow graph without going through n and
- the set of edges connecting all the nodes in its node set

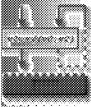
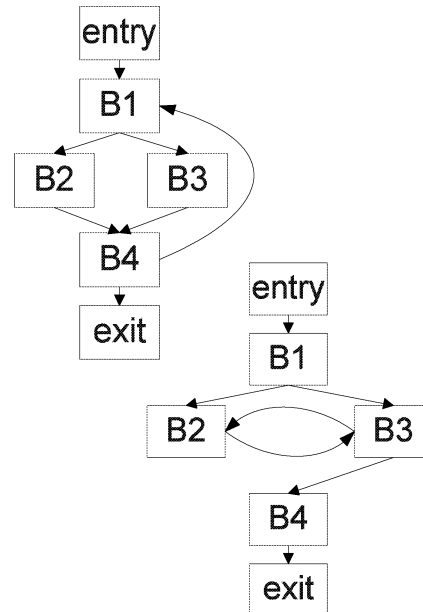
■ Unique single entry point (loop header)

■ There is at least one path back to the header from a node



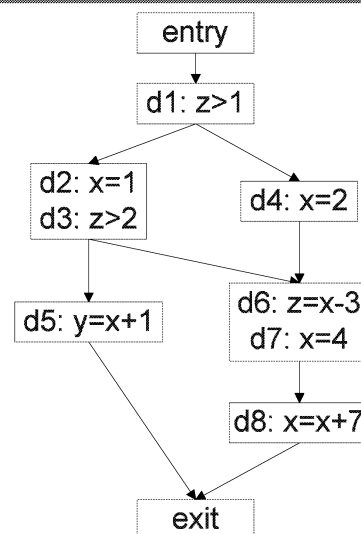
Reducible Flow Graphs

- A flow graph is reducible iff we can partition the edges into two disjoint groups:
 - The forward edges that form an acyclic graph in which every node can be reached from the entry node
 - The back edges
- Reducible \rightarrow all the loops are natural loops
- No jumps into the middle of the loops
- Structured flow-of-control statements produce reducible flow graphs
 - if-then-else, while-do, continue, break, etc.
- Non-reducible flow graphs are rare in most of programs

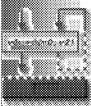


Du-Ud chains

- Du-chains
 - A du-chain for a variable connects a definition of the variable to all the uses to which it may flow
- Ud-chains
 - A ud-chain for a variable connects a use of the variable to all the definitions to which it may flow

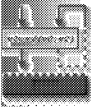


Du-chain of x: d2-d6,d2-d5,d4-d6,d7-d8
Ud-chain of x: d5-d2,d8-d7,d6-d4,d6-d2



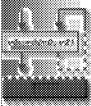
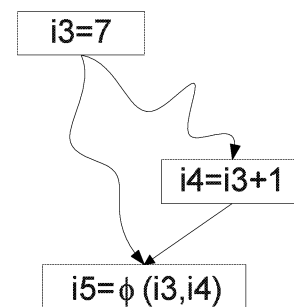
Static Single Assignment Form

- A procedure is in SSA form if every variable assigned a value in it occurs as the target of only one assignment
- Explicit Du-chains but compact
- Useful
 - Dead-code elimination
 - Constant propagation
 - Invariant code motion and removal
 - Strength reduction
 - etc.



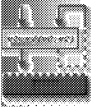
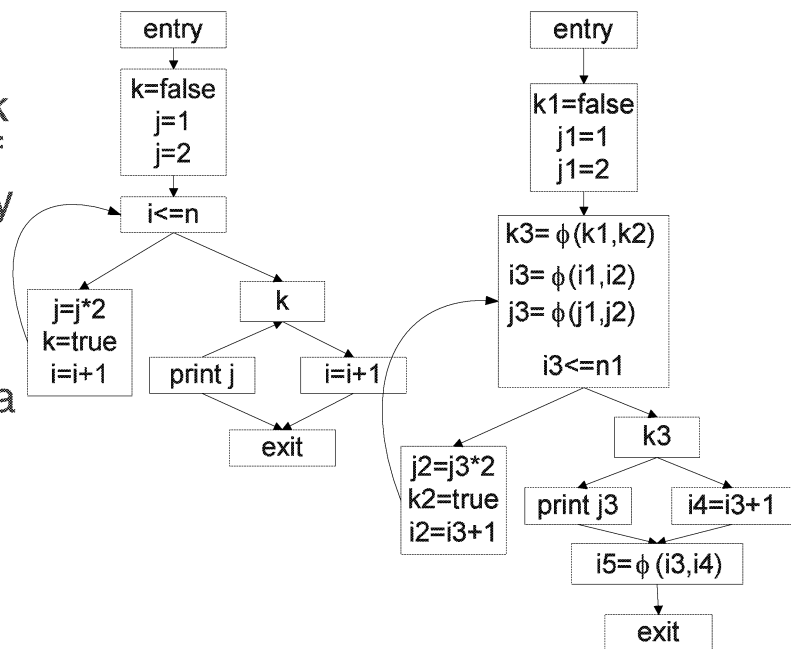
Φ -Functions

- Each Φ -function has as many argument positions as there are versions of the variable coming together at the join point
- Each argument position corresponds to a particular control-predecessor of the point
- Simply place Φ -functions at each join point in the program (but wasteful)



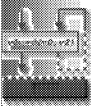
Dominance Property of the SSA form

- If variable x is used in a Φ -function in a block n , the definition of x dominates every predecessor of n
- If x is used in a statement that does not contain a Φ -function in a block n , the definition of x dominates n



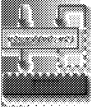
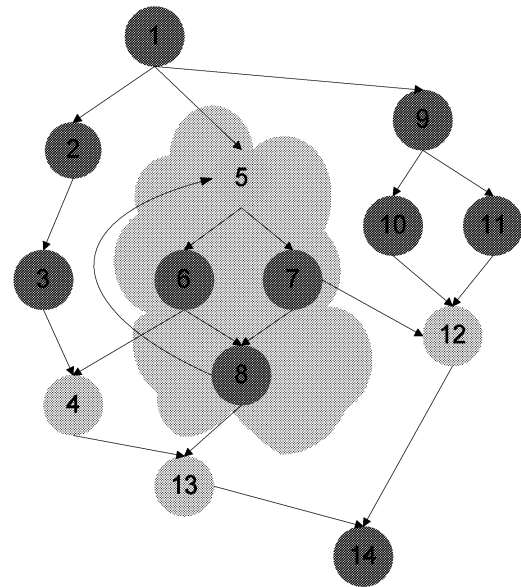
SSA Translation

- Translation (not minimal)
 1. Figure out at what join points to insert Φ -functions
 2. Insert trivial Φ -functions (i.e., $\Phi(x, x, \dots, x)$)
 3. Rename the definitions and uses of variables
- Inefficient, use iterated dominance frontiers!
 - Complicated, but efficient



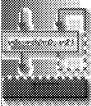
Dominance Frontiers

- The dominance frontier of node x is the set of all nodes y in the flow graph such that x dominates an immediate predecessor of y but does not strictly dominate y
 - The dominance frontier of a node is the border between dominated and undominated nodes



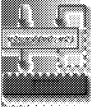
Data Dependence Analysis

- Determine the ordering relationship between statements
- For two given storage (register or memory) references, whether the locations of storage they access overlap
- For
 - Instruction scheduling
 - Cache optimization
 - Parallelization
 - etc.



Dependence Relations

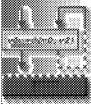
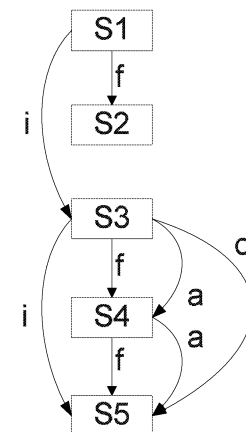
- If S1 precedes S2 in their given execution order, $S1 < S2$.
- A dependence between two statements in a program is a relation that constraints their execution order



Data Dependences

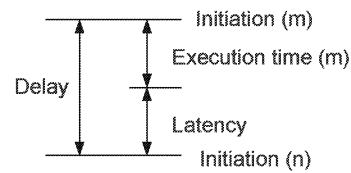
- A constraint that arises from the flow of data between statements
 - Write a < read a: true (flow) dependence ($S1 \delta^f S2$)
 - Read a < write a: antidependence ($S1 \delta^a S2$)
 - Write a < write a: output dependence ($S1 \delta^o S2$)
 - Read a < read a: input dependence ($S1 \delta^i S2$)
- Dependence graph
 - Nodes: statements
 - Edges: dependences

S1: $a = b + c$
S2: if $a > 10$ goto S5
S3: $d = b * e$
S4: $e = d + 1$
S5: $d = e / 2$



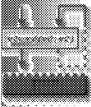
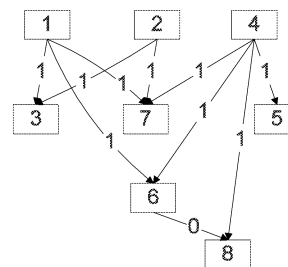
Basic Block Dependence DAGs

- A basic block's dependence graph is always a DAG
- Dependence DAG
 - Node: machine instructions
 - Edges: dependences between the instructions
 - Used in instruction scheduling
 - A node m is a predecessor of another node n in the dependence DAG if n must not execute until m has executed for some number of cycles
 - An edge between m and n is labeled with the required latency between m and n
- List scheduling



```

1: r3 = [r15](4)
2: r4 = [r15+4](4)
3: r2 = r3 - r4
4: r5 = [r12](4)
5: r12 = r13 + 4
6: r6 = r3 * r5
7: [r15+4](4) = r3
8: r5 = r6 + 2
    
```



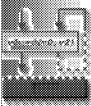
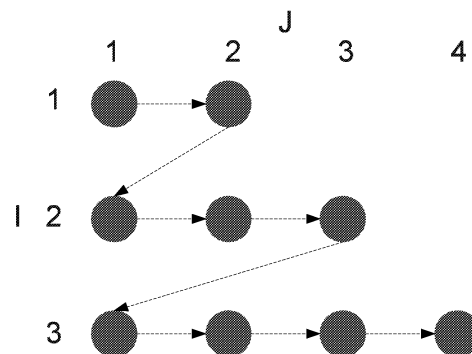
Dependences in Loops

- Iteration space: k -dimensional polyhedron consisting of all the k -tuples (index vectors) of values of the loop indices.


```

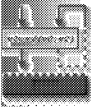
for i = 1 to 3 do
  for j = 1 to i+1 do
    S1  a(i,j) = b(i,j) + c(i,j)
    S2  b(i,j) = a(i,j-1)
  endfor
endfor
            
```

 - Dependence
 - $S1[i,j-1] \delta^l S2[i,j]$
 - $S1[i,j] \delta^a S2[i,j]$
- Lexicographic ordering of index vectors
 - $(i_1, i_2, \dots, i_n) < (j_1, j_2, \dots, j_n)$ iff $\exists k, 1 \leq k \leq n$, s.t., $i_1 = j_1, i_2 = j_2, \dots, i_{k-1} = j_{k-1}$ and $i_k < j_k$.
 - Iteration (i_1, i_2, \dots, i_n) of a loop nest precedes iteration (j_1, j_2, \dots, j_n) iff $(i_1, i_2, \dots, i_n) < (j_1, j_2, \dots, j_n)$



Loop-carried Dependences

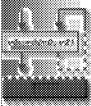
- Loop-carried dependence: caused by a loop surrounding it.
- Loop-independent dependence: independent of the loop surrounding



Dependence Testing

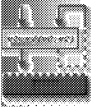
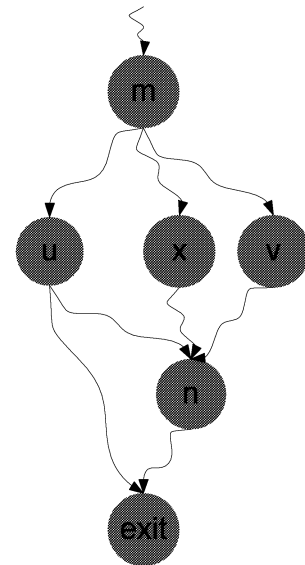
```
for i = 1 to 4 do
  b[i] = a[3*i-5] + 2.0
  a[2*i+1] = 1.0/i
endfor
```

- We need to see whether there is an integer i_1 and i_2 that satisfy the equation
$$2 * i_1 + 1 = 3 * i_2 - 5 \text{ and } 1 \leq i_1, i_2 \leq 4$$
- Dependence testing techniques for loops
 - Constrained Diophantine equations, Integer programming (NP-complete), GCD test, Extended GCD test, Strong and weak Single Index Variables tests, Delta test, Acyclic test, Power test, Simple Loop Residue test, Fourier-Motzkin test, Constraint-Matrix test, Omega test, etc.



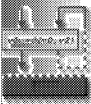
Control Dependences

- A constraint that arises from the control flow of the program
- Can node m directly control whether node n is executed?
- A node n is control-dependent on m iff
 - there exists a control-flow path from m to n such that every node in the path other than m is postdominated by n and
 - n does not postdominate m
- Control Dependence Graph
 - Edge from m to n whenever n is control-dependent on m
 - n is control dependent on m whenever m is in the dominance frontier of n in the reverse flow graph



Bibliography

- Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools", Addison Wesley, 1986.
- Andrew W. Appel. "Modern Compiler Implementation in Java", second edition, Cambridge University Press, 2002.
- Ron Cytron, Jean Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Program Dependence Graph", ACM Transactions of Programming Languages and Systems, Vol. 13, No. 4, October 1991, pp. 451-490.
- Steven S. Muchnick. "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997.



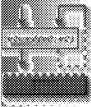
Lecture #2

Jaejin Lee

Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

jlee@cse.snu.ac.kr

<http://aces.snu.ac.kr/~jlee>



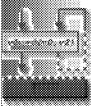
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University



1

Billion Transistor Architectures

- Microprocessors will have more than a billion transistors on a single chip by 2010
- The trends
 - Advanced wide issue superscalar processors
 - Issue 16 or 32 instructions per cycle
 - Simultaneous multithreaded (SMT) processors
 - Share an aggressive pipeline between multiple tasks when there is insufficient instruction-level parallelism
 - Chip multiprocessors (CMP)
 - Place a small number (2 to 16) of processors on a single chip
 - Processing in Memory (PIM) Architectures
 - Superscalar core with DRAM banks on a single chip to provide with sufficient bandwidth



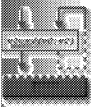
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University



2

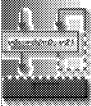
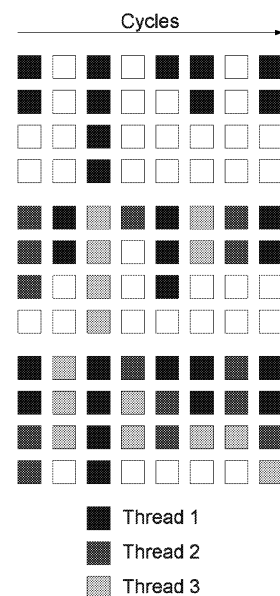
Threads

- Thread of control
- A thread consists of:
 - Program counter
 - Register set
 - Stack
- Share resources with other threads
 - Code or text section
 - Data section
 - OS resources: open files, signals, etc.



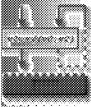
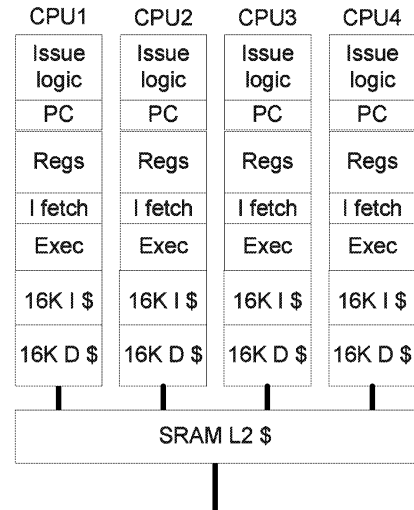
Simultaneous Multithreading Architectures

- Superscalar processors already have many HW mechanism to support multithreading
 - Adding a per thread renaming table and separate PCs
- Alpha 21464
- Conventional superscalar
 - Issue slots are wasted when there is not enough ILP in a thread (horizontal waste)
- Conventional multithreaded architecture (Tera)
 - Switching different thread contexts each cycle
 - Tolerate long latency operations (remove vertical waste)
 - Still waste unused issue slots (horizontal waste)
- SMT architecture
 - Selects instructions for execution from all threads each cycle
 - Remove both horizontal and vertical waste



Chip Multiprocessor Architectures

- A group of small identical processors in a single chip
 - E.g., eight 2-issue superscalar processors
 - Shared on chip L2 SRAM cache
- SUN dual-core UltraSparc 3 with SMT
- To exploit thread level and process level parallelism
- Similar to conventional Symmetric multiprocessors, but tightly coupled



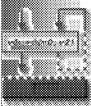
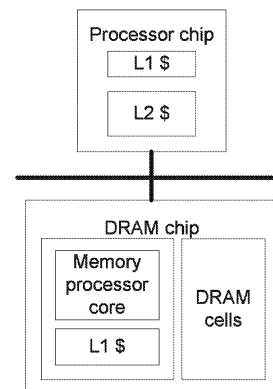
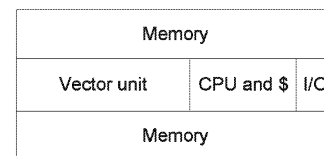
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

5



Processing-In-Memory Architectures

- The performance of applications is dominated by memory
 - To provide high bandwidth and low latency
- A simple processor core is embedded in the memory system
- PIM chips as main processors
 - IRAM, RAW, Execube, Smart Memories, ...
- PIM chips as helper processors (Intelligent Memory)
 - Heterogeneous processors
 - In main memory
 - FlexRAM, DIVA, Active Pages, ...
 - In memory controller
 - NVIDIA (graphics engine)



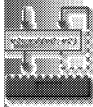
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

6



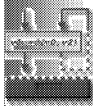
Embedded Systems Architectures

- Supercomputing technology
 - (about 10 years later) → desktop computing technology
 - (? years later) → embedded computing technology
- Require a device that does more things at one time
 - Multiple simultaneous deadlines
- Heterogeneous multiprocessor architectures
 - Philips Viper with MIPS and TriMedia cores
 - For set-top box and digital TV systems
 - MIPS RISC core
 - › High performance, running OS, controlling peripherals
 - TriMedia VLIW core
 - › High performance, audio and video processing
 - Intel IXP2xxx network processor architecture
 - XScale core
 - › Route table maintenance and system-level management
 - 16 programmable microengines
 - › Packet processing
 - Lots
- SMT architectures
 - MemoryLogix MLX1
 - Multithreaded 586 core



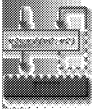
Programming These Systems?

- Conventional multithreading will not work
 - Loop level parallelism
- Heterogeneous multithreading
 - Main threads and helper threads
 - How to extract these threads?
 - Compiler's work
 - More difficult than exploiting conventional parallelism



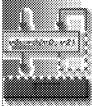
The Case of Intelligent Memory Architectures

- Co-execution
- Prefetching

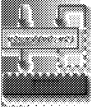
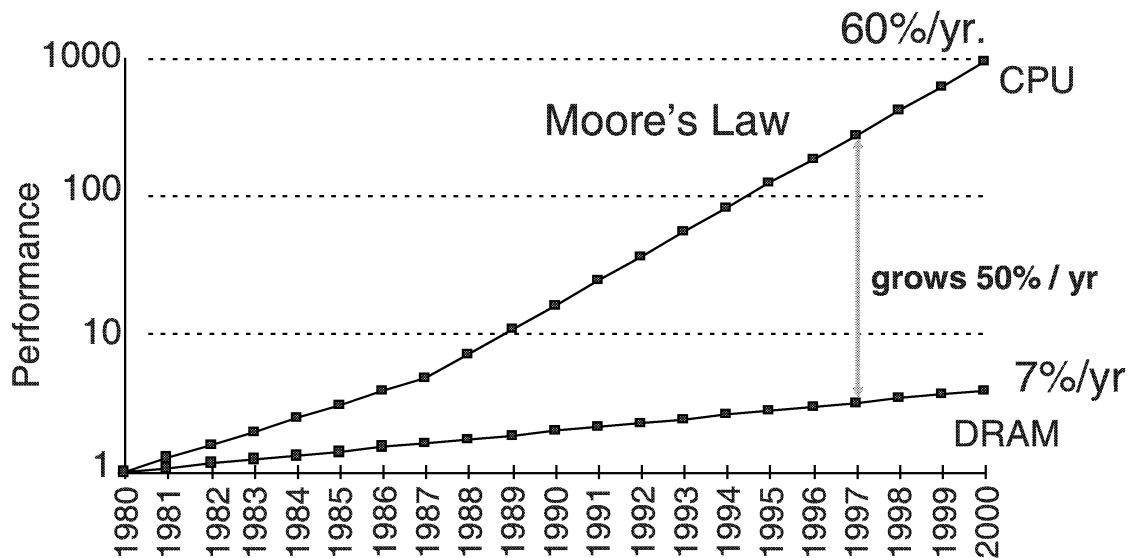


Memory Wall Problem

- Processor-Memory performance gap
 - Moore's law: microprocessor performance has been improving at a rate of 60% per year (2Ghz <) .
 - The latency of DRAM has been improving at a rate of less than 10% per year (< 533Mhz).



Processor-Memory performance gap



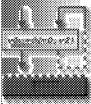
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

11



Memory Wall Problem (contd.)

- Performance is dominated by memory
- No optimal solutions yet
 - E.g., memory hierarchy
 - More than 2,000 papers on this topic
- One of the feasible hardware solutions:
Processing-In-Memory (PIM) architectures



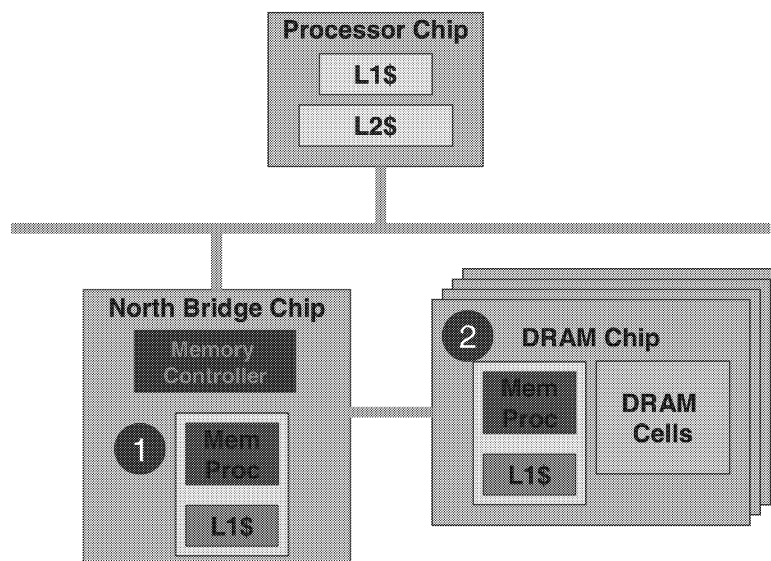
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

12



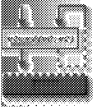


The Intelligent Memory Architecture



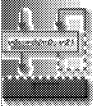
Heterogeneous Mix of Processors

- Host processor (P.host) and memory processor (P.mem)
 - P.host: a wide-issue superscalar with a deep cache hierarchy
 - P.mem: a simple, narrow-issue superscalar with only a small cache
- A user-level thread is running on the memory processor.
 - Flexible and adaptable



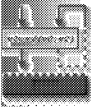
How to Exploit the Intelligent Memory?

- Co-execution of the memory thread with the main thread
- Prefetching using the memory thread for the main thread

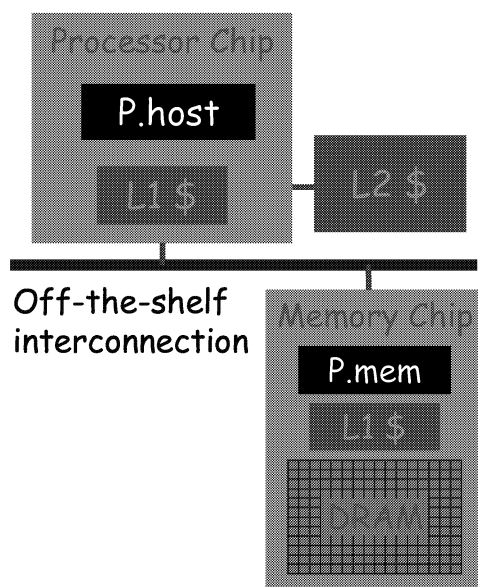


Co-execution

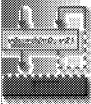
- Using a compiler,
 - Partition code into compute-/memory-intensive sections.
 - The memory-intensive sections are wrapped into a memory thread.
 - Statically/dynamically map the sections to the best processor.
 - Overlap the execution of the main thread and the memory thread.



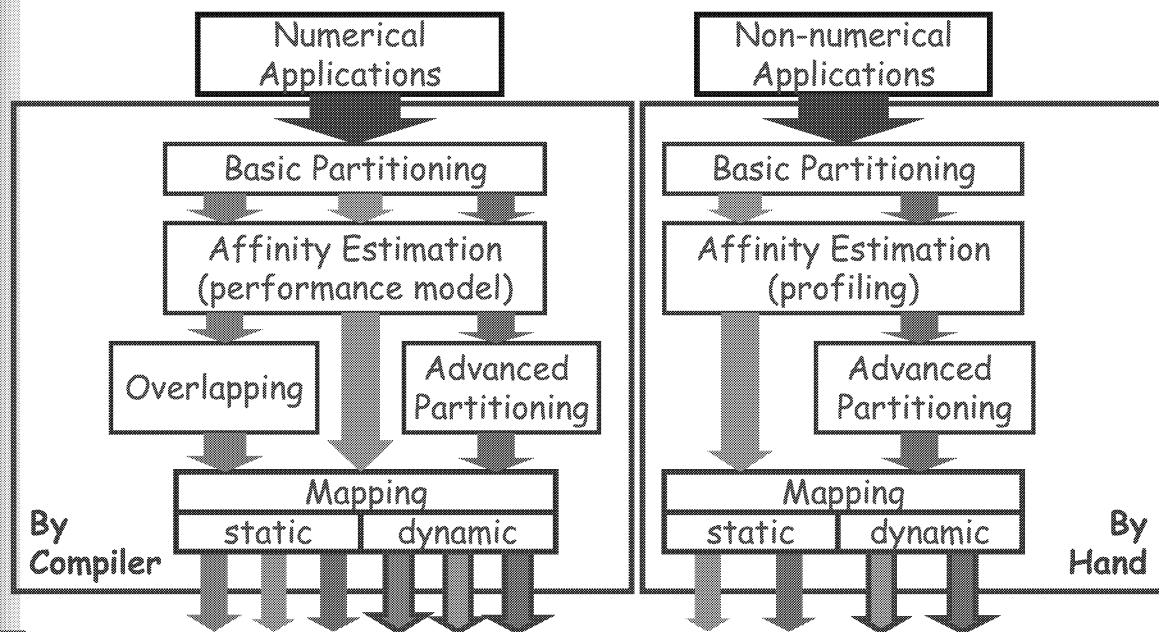
Cache Coherence



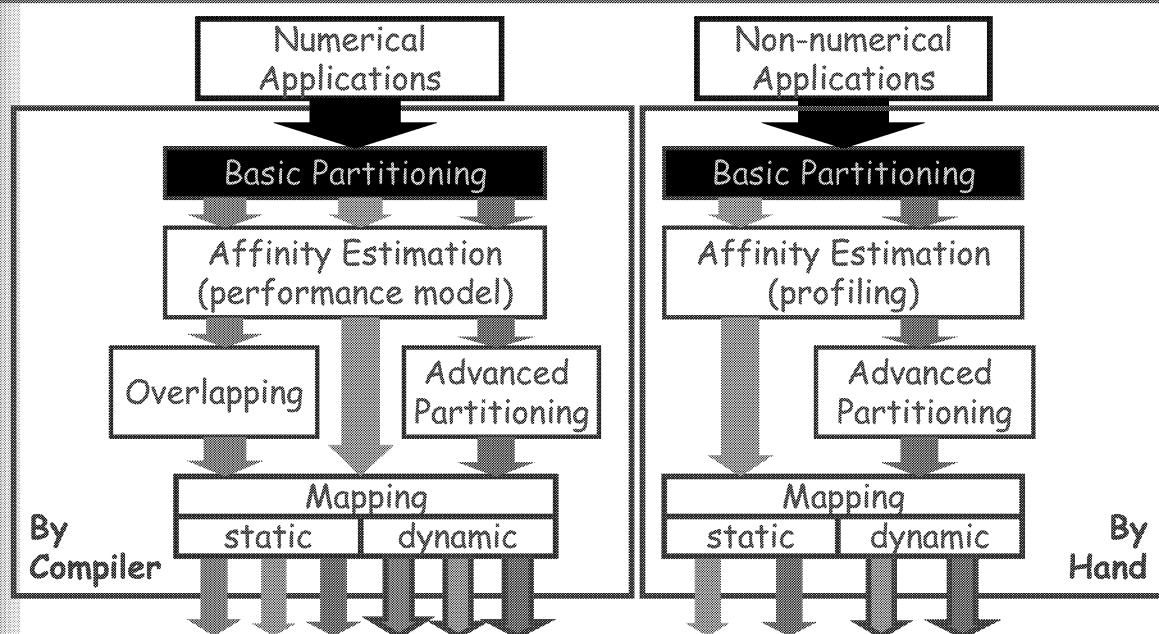
- Compiler controlled
 - Before P.mem starts execution, P.host write-back dirty lines that may be read by P.mem
 - Before P.host starts execution, it invalidates lines that may have been written by P.mem



Overview of the Co-execution Algorithm

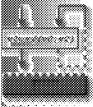


Overview of the Co-execution Algorithm



Basic Partitioning

- Finds code sections (basic modules) that are easy to extract and have,
 - Homogeneous computing and memory behaviors
 - Good locality of references
- A basic module is a loop nest, where
 - Each nesting level has only one loop
 - May span several subroutine levels

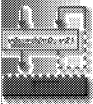


Basic Partitioning (example)

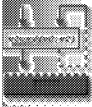
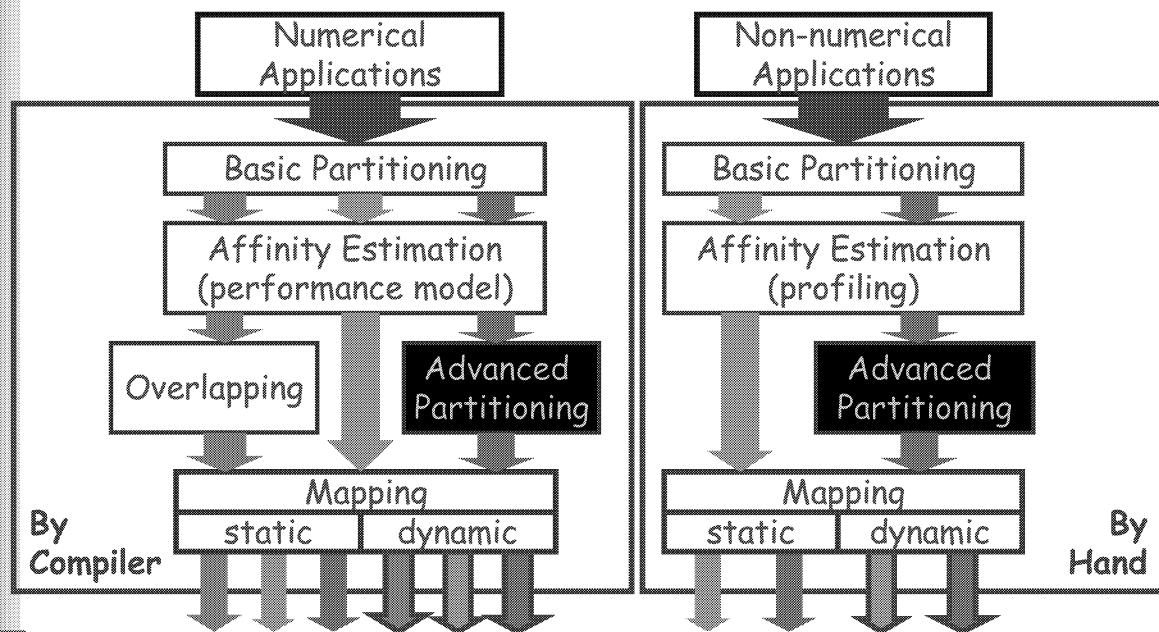
```
N1 = N*2
DO I=1, N1
  N2 = X * 4
  DO J = 1, N2
    X = ...
    A(J,I) = ...
  ENDDO
  IF (X .LT. 1.0) THEN
    X = ...
  ENDIF
ENDDO
C(N) = ...
DO K = 1, N-1
  B(K) = C(K+1)
ENDDO
```



```
N1 = N*2
DO I=1, N1
  N2 = X * 4
  DO J = 1, N2
    X = ...
    A(J,I) = ...
  ENDDO
  IF (X .LT. 1.0) THEN
    X = ...
  ENDIF
ENDDO
C(N) = ...
DO K = 1, N-1
  B(K) = C(K+1)
ENDDO
```

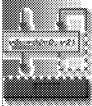


Overview of the Co-execution Algorithm



Advanced Partitioning

- Increases the grain size of the module, possibly reducing uniformity (compound modules)
- Decreases synchronization overhead
- Repeatedly applying expansion and combining steps



Advanced Partitioning (Expansion)

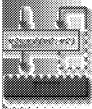
- Expansion: similar to the basic partitioning, but

```
if P then
...
else
  M
endif
```



```
if P then
...
else
  M
endif
```

New module M'



Advanced Partitioning (Combining)

- Combining: two adjacent modules with the same affinity are combined into a new module

```
...
M1
M2
...
```



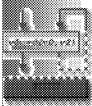
```
...
M1
M2
...
```

```
if P then
  M1
else
  M2
endif
```

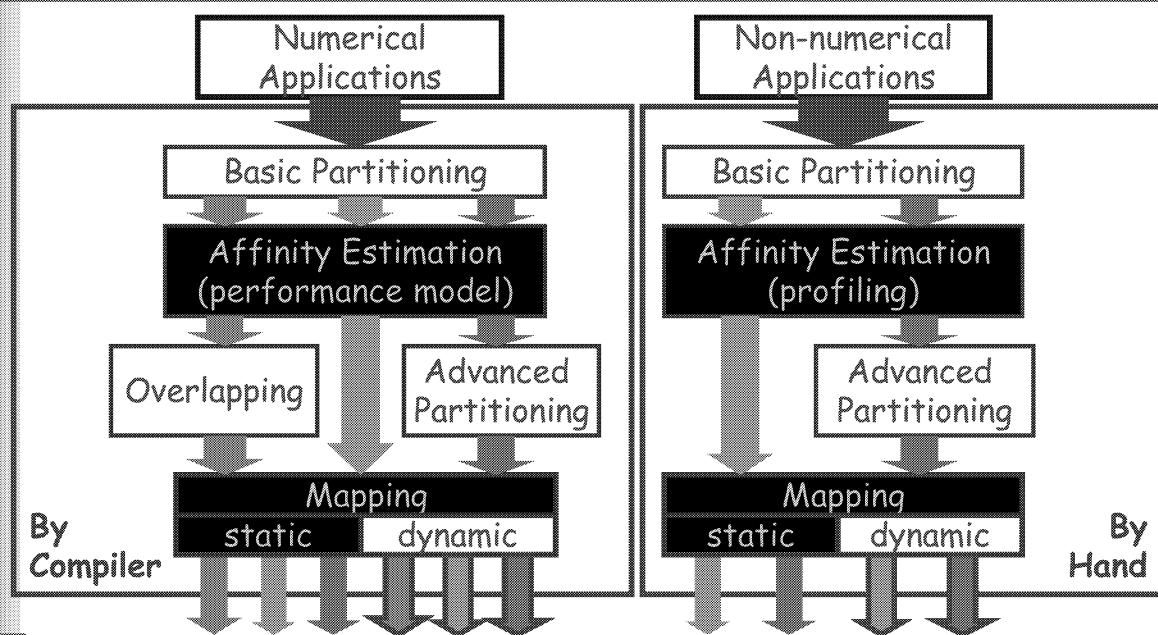


```
if P then
  M1
else
  M2
endif
```

New module M'



Overview of the Co-execution Algorithm



Static Mapping

■ Performance model (numerical apps)

- Execution time = $T_{\text{comp}} + T_{\text{memstall}}$
- Stack distance model for the number of misses

$$T_{\text{comp}} = \max\left(\frac{T_{\text{int}}}{N_{\text{int}}}, \frac{T_{\text{fp}}}{N_{\text{fp}}}, \frac{T_{\text{ldst}}}{N_{\text{ldst}}}\right) + T_{\text{other}}$$

$$T_{\text{memstall}} = \sum_{i \in \text{caches}} \text{miss}_i \cdot \text{penalty}_i$$

■ Profiling (non-numerical apps)

- Gather execution time and the number of invocations for all modules and subroutines





29



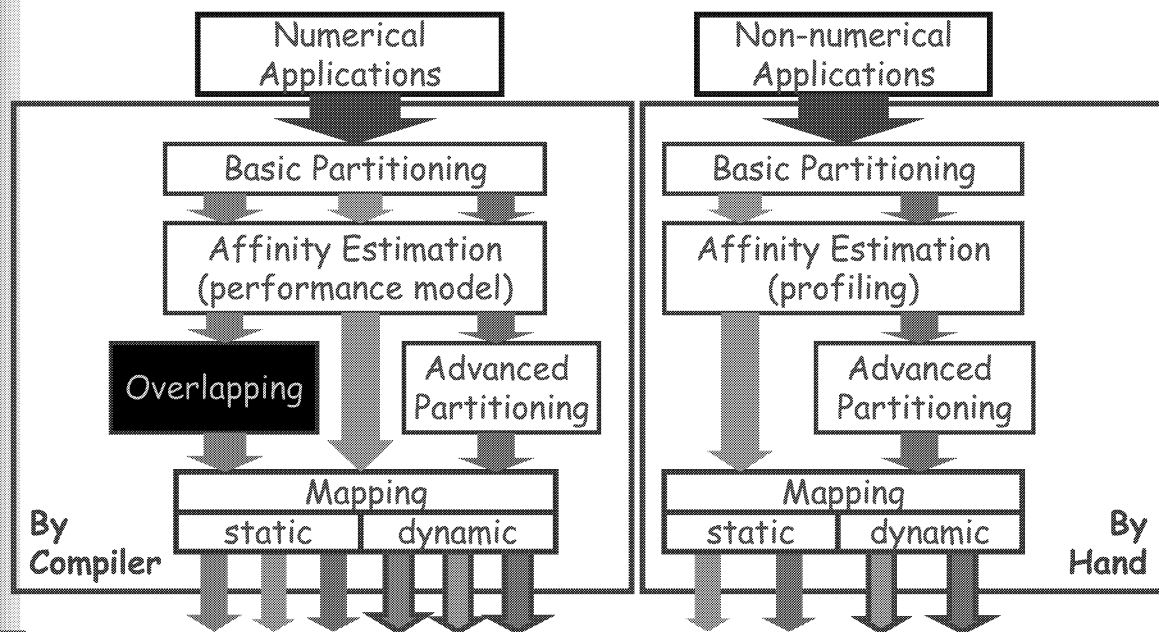
- | | Invocation | 1 | 2 | 3 | 4 | 5 | ... |
|----------------|------------|---|---|---|---|---|-----|
| Coarse | P.host | █ | | █ | █ | █ | |
| | P.mem | | █ | | | | |
| CoarseR | P.host | █ | | █ | █ | | |
| | P.mem | | █ | | | █ | |

- Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

30



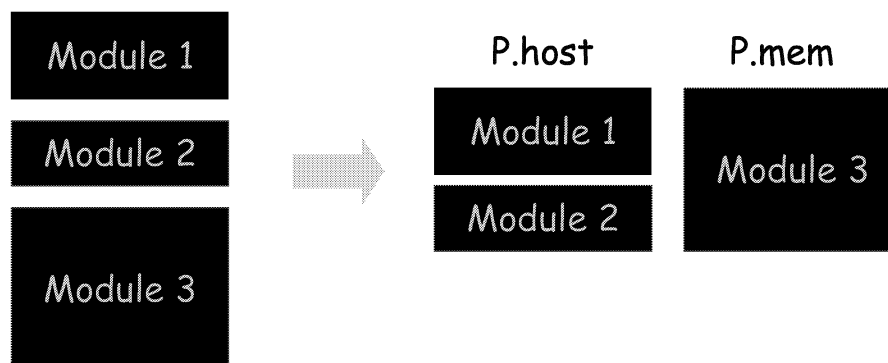
Overview of the Co-execution Algorithm



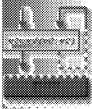
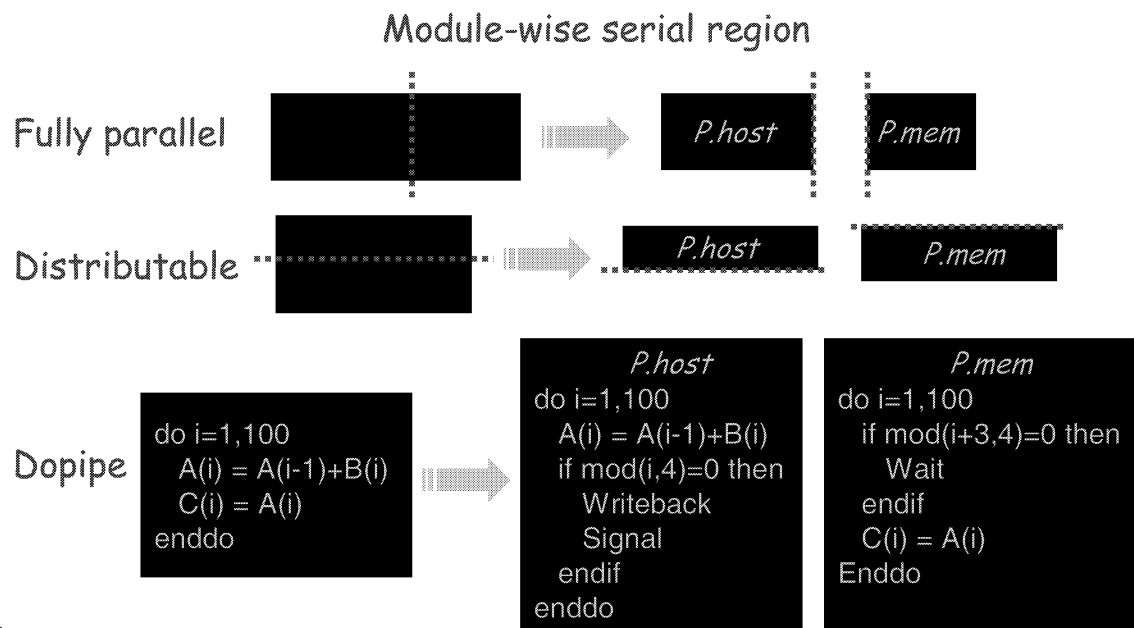
Overlapping Execution

- Module-wise parallel region and module-wise serial region

Module-wise parallel region



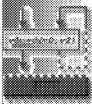
Overlapping Execution (contd.)



Evaluation Environment

- Applications:
 - Numerical: Swim, Tomcatv, LU, TFFT2, Mgrid
 - Non-numerical: Bzip2, Mcf, Go, M88ksim
- Simulation: Mint-based execution-driven

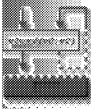
| Module | Parameter | Values |
|----------------|-------------------------|---|
| P.host::P.mem | Frequency | 800MHz :: 800MHz |
| | Issue width | Out-of-order 6 :: In-order 2 |
| | Functional Units | 4Int+4Fp+2Ld/St :: 2Int+2Fp+1Ld/St |
| P.host Caches | L1-Data | Write-through, 32KB , 2-cycle hit |
| | L2-Data | Write-back, 1MB (512KB for non-numerical apps.) , 10-cycle hit |
| | Write-back overhead | 5 + 1 × num_cache_lines (background) |
| | Invalidation overhead | 5 + 1 × num_cache_lines |
| P.mem Cache | L1-Data | Write-back, 16KB , 2-cycle hit |
| Memory and Bus | Memory Latency (cycles) | 160 from P.host, 21 from P.mem |
| | Bus Type | Split transaction, 16-B wide |



Average Characteristics of Basic Modules

- Different applications have a different distribution of module affinity.

| Averages | Numerical Applications | Non-numerical Applications |
|-------------------------------------|------------------------|----------------------------|
| Total Modules | 13.2 (99.1%) | 41.8 (63.0%) |
| P.host Affinity | 4.0 (37.0%) | 31.0 (38.9%) |
| P.mem Affinity | 9.2 (62.1%) | 10.8 (24.1%) |
| Parallel Modules | 11.4 (70.9%) | 8.8 (1.3%) |
| Serial Modules | 1.8 (28.2%) | 33.0 (61.7%) |
| Average Number of Invocations | 442.9 | 182,025 |
| Average Module Size (P.host cycles) | 4,570 K | 477 K |

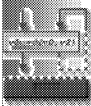
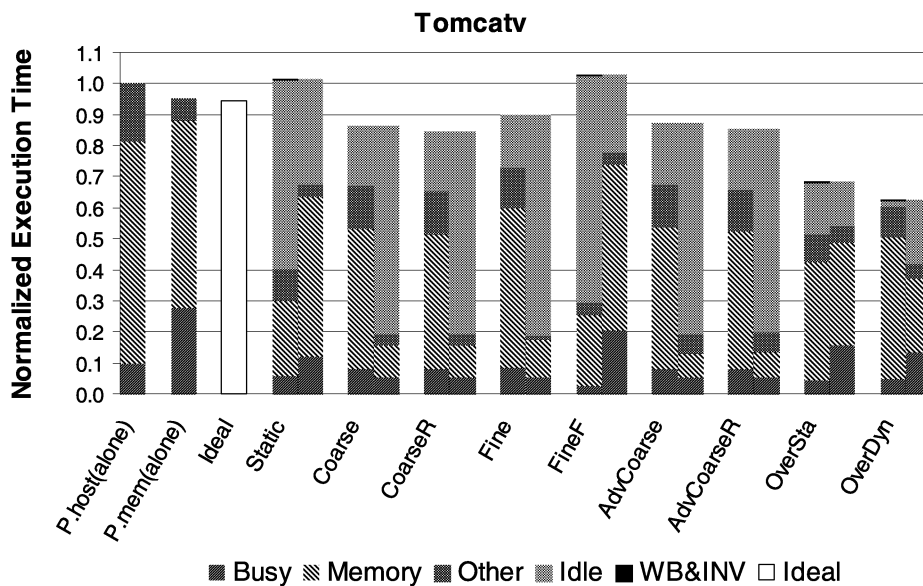


Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

35



Tomcatv (SPECfp95)

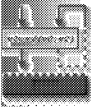
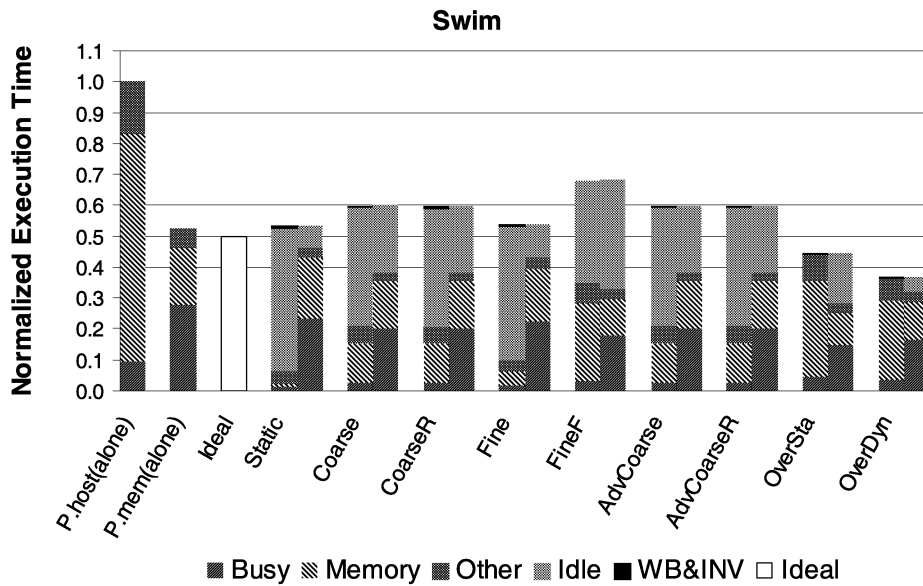


Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

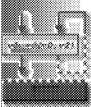
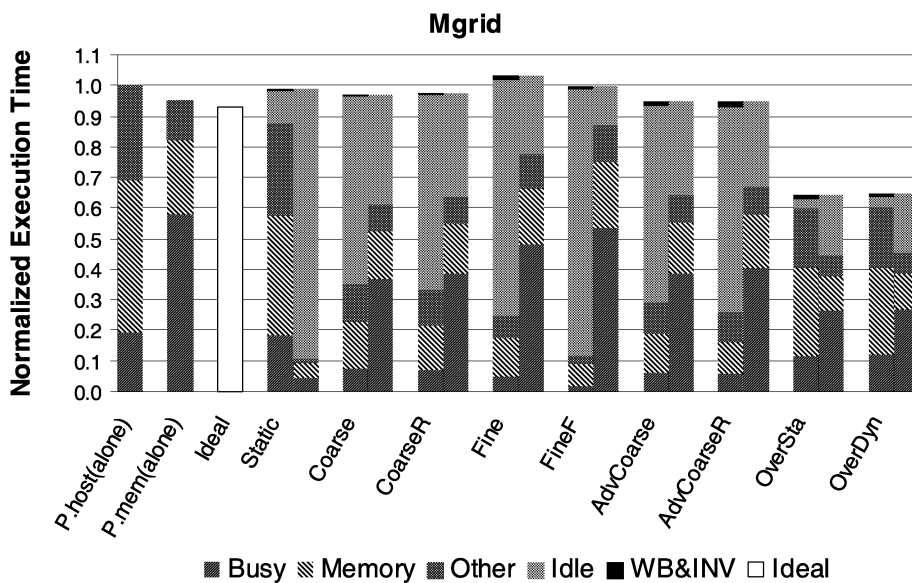
36



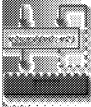
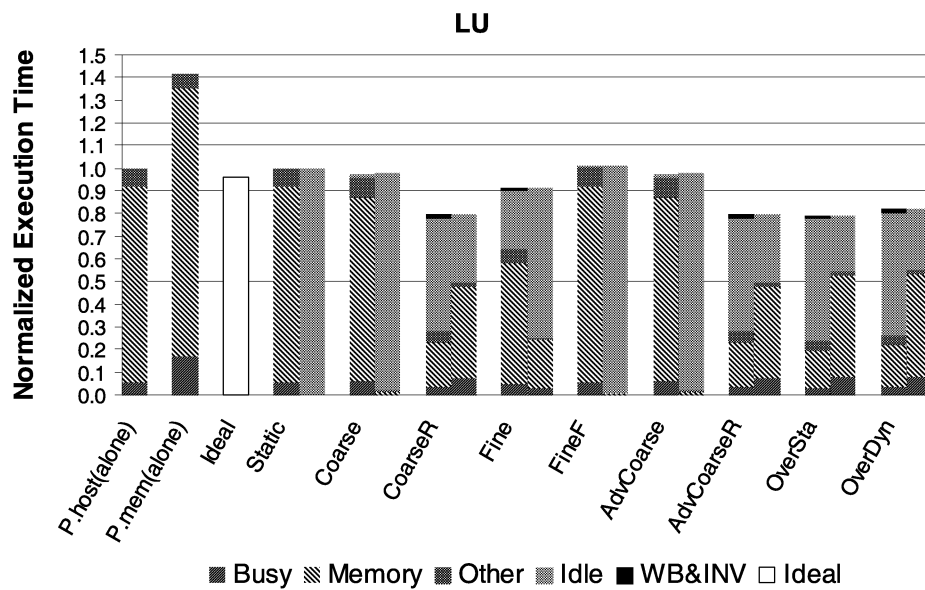
Swim (SPECfp2000)



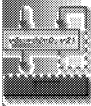
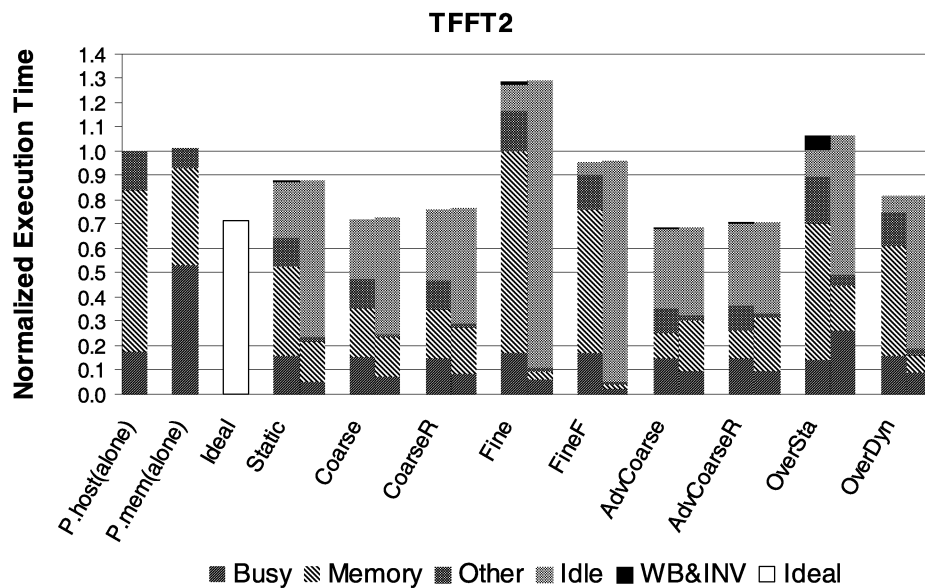
Mgrid (SPECfp2000)



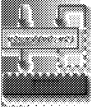
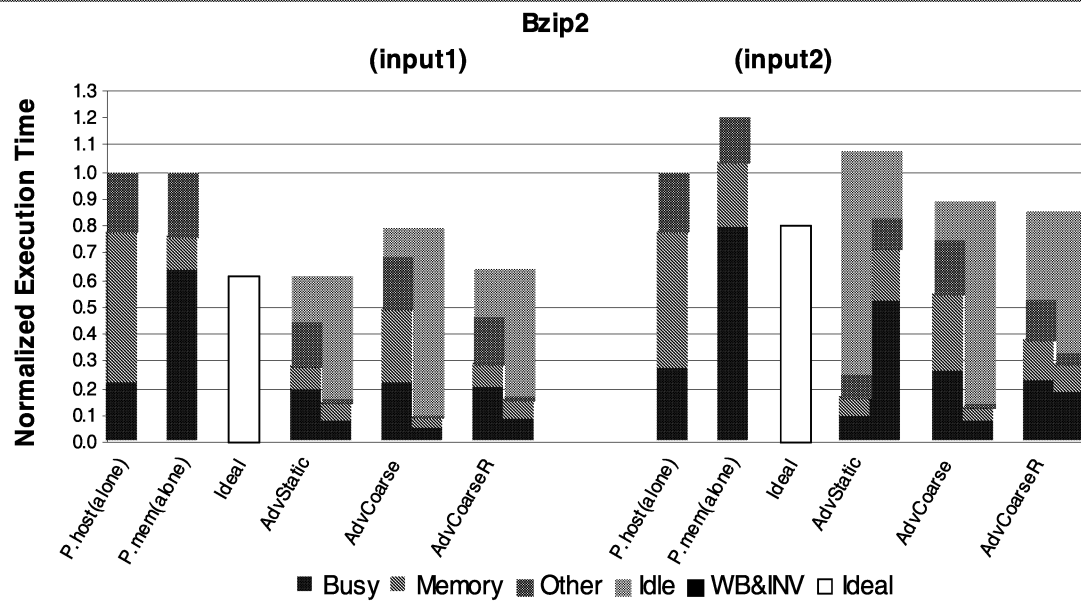
LU (NAS)



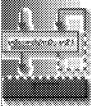
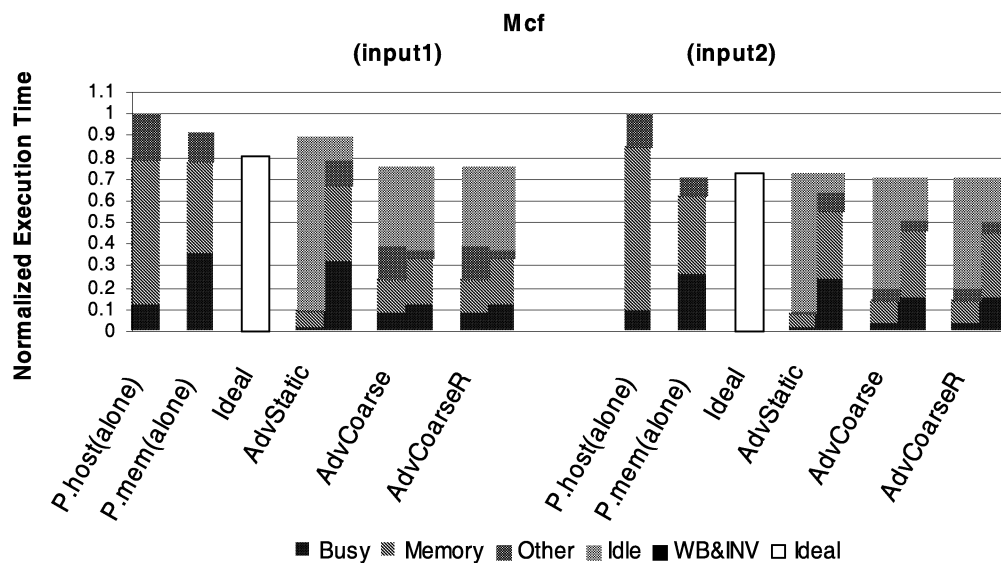
TFFFT2 (NAS)



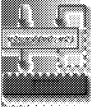
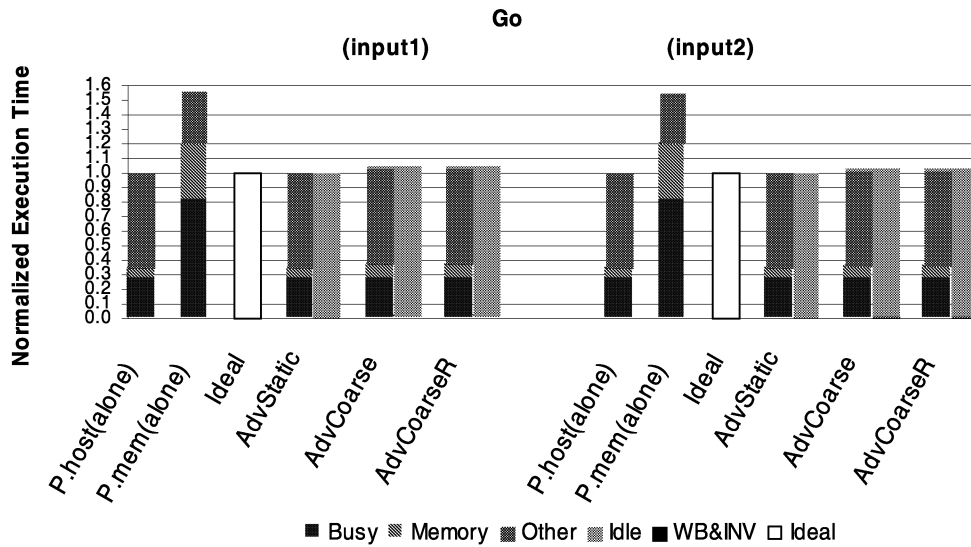
Bzip2 (SPECint2000)



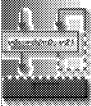
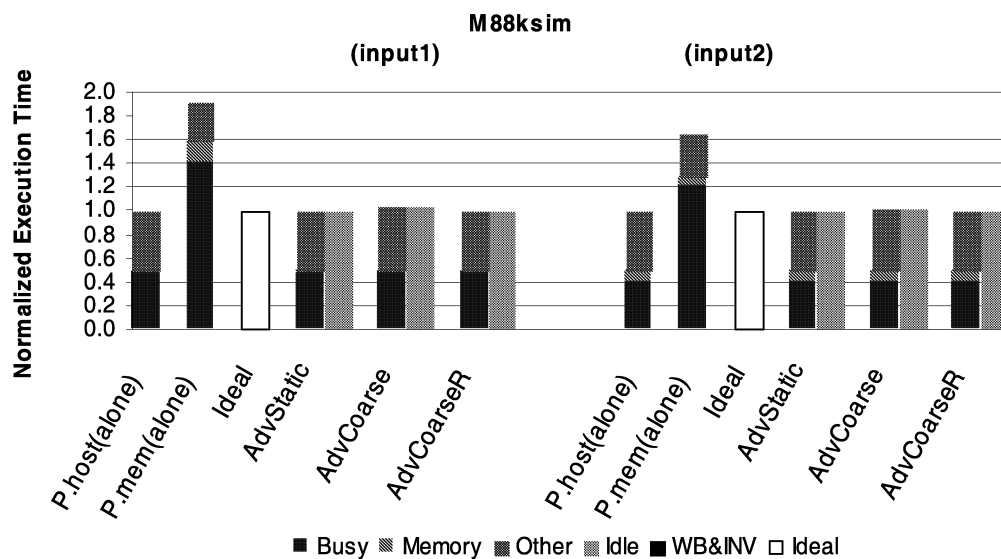
Mcf (SPECint2000)



Go (SPECint95)



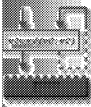
M88ksim (SPECint95)



Overall Speedups for Co-execution

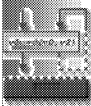
- Our co-execution algorithm delivers speedups that are comparable to the ideal speedup.

| Apps. | P.host(alone) /AdvCoarseR | P.host(alone) /OverDyn | Amdahl's 2 P.hosts | 2-processor SGI |
|---------|------------------------------|---------------------------|-----------------------|--------------------|
| Swim | 1.67 | 2.71 | 2.00 | 1.85 |
| Tomcatv | 1.17 | 1.60 | 1.67 | 1.44 |
| LU | 1.26 | 1.22 | 1.04 | 0.99 |
| TFFT2 | 1.42 | 1.22 | 1.91 | 0.80 |
| Mgrid | 1.05 | 1.55 | 1.94 | 1.47 |
| Average | 1.31 | 1.66 | 1.71 | 1.31 |
| Bzip2 | 1.37 | - | 1.01 | 0.99 |
| Mcf | 1.37 | - | 1.01 | 1.00 |
| Go | 0.97 | - | 1.01 | 0.57 |
| M88ksim | 1.01 | - | 1.03 | 1.00 |
| Average | 1.18 | - | 1.02 | 0.89 |



Prefetching

- New correlation prefetching in software using the memory thread
 - Widely applicable
 - Effective
 - Flexible
 - Inexpensive




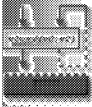
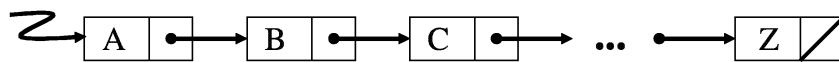
Background

■ Correlation Prefetching

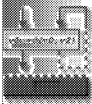
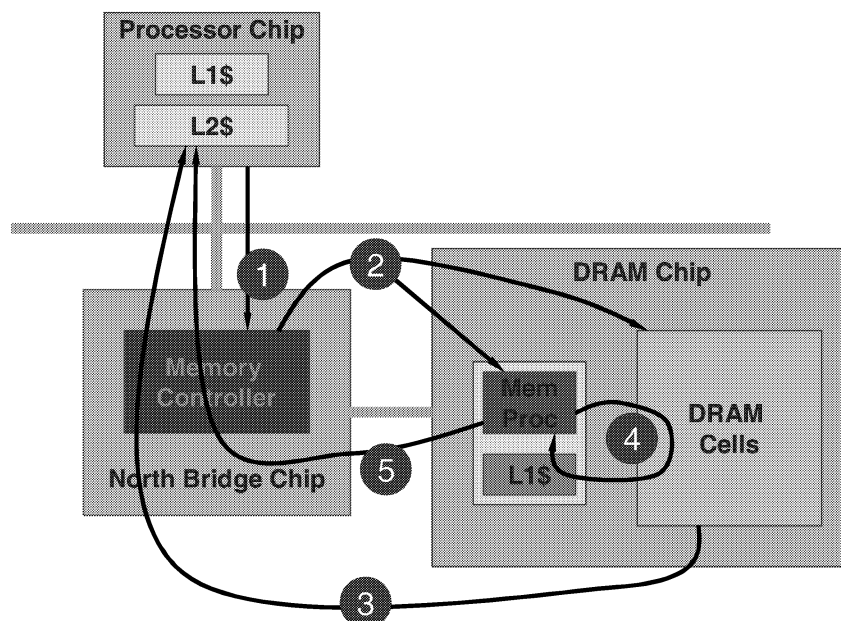
- Joseph and Grunwald [ISCA'97], Alexander and Kedem [HPCA'96], Lai, Fide, and Falsafi [ISCA,01]
- Records sequences of miss addresses in a correlation table.
- When the head of a sequence is seen, prefetch the rest.
- Effectively prefetches data when there are repeatable patterns.
- Hardware implementation (SRAM).

$a[4*(i++)]$ 

$a[\text{foo}(i)]$ 

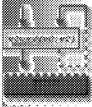


The Scheme



Comparison

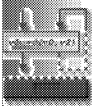
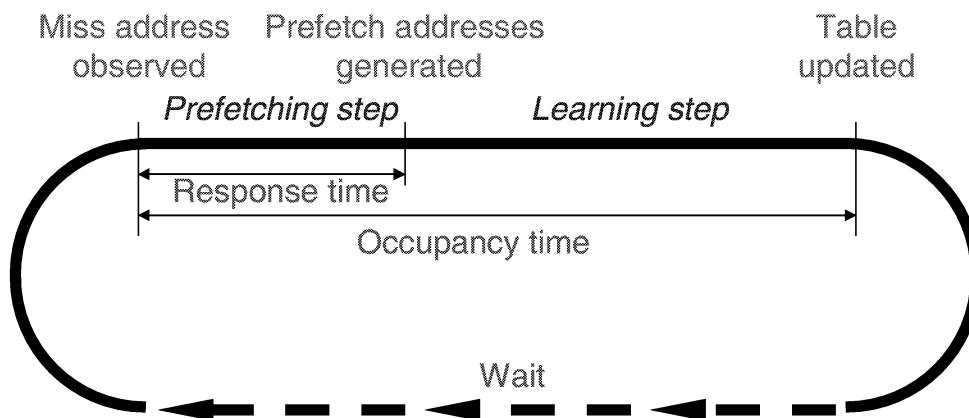
| | Previous work | Our approach |
|---------------------------|--|--|
| Prefetcher implementation | Custom hardware | User programmable general purpose core |
| Location | L1 prefetching | L2 prefetching |
| Table structure | Large (1-7.6MB SRAM) On-chip, dedicated | Small (DRAM) Dynamically allocated |
| Multiprogramming support | Cross pollution | One table per application |
| Customization support | No | Yes |



The Mechanism of the Memory Thread

■ Requirements:

- Low response time
- Occupancy time < miss distance



Correlation Table

Basic Organization
(Joseph & Grunwald)

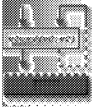
Addresses of immediate successors

| Tag | Succ Level 1 | |
|-----|--------------|--|
| | | |
| | | |
| | | |
| | | |

Advanced Organization

Addresses of next immediate successors

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|--|--------------|--|-----|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | ... |



Learning Step

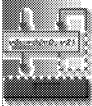
Basic Organization

| Tag | Succ Level 1 | |
|-----|--------------|--|
| A | | |
| | | |
| | | |
| | | |

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|--|--------------|--|-----|
| A | | | | | |
| | | | | | |
| | | | | | |
| | | | | | ... |

A, B, C, A, D, C, ...
↑
Current miss



Learning Step (contd.)

Basic Organization

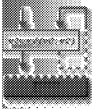
| Tag | Succ Level 1 | |
|-----|--------------|--|
| A | B | |
| B | | |
| | | |
| | | |

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|--|--------------|--|-----|
| A | B | | | | ... |
| B | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

A, B, C, A, D, C, ...

↑
Current miss



Learning Step (contd.)

Basic Organization

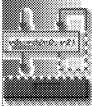
| Tag | Succ Level 1 | |
|-----|--------------|--|
| A | B | |
| B | C | |
| C | | |
| | | |

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|--|--------------|--|-----|
| A | B | | C | | ... |
| B | C | | | | |
| C | | | | | |
| | | | | | |
| | | | | | |

A, B, C, A, D, C, ...

↑
Current miss



Learning Step (contd.)

Basic Organization

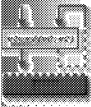
| Tag | Succ Level 1 | |
|-----|--------------|--|
| A | B | |
| B | C | |
| C | A | |
| | | |

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|--|--------------|--|-----|
| A | B | | C | | ... |
| B | C | | A | | |
| C | A | | | | |
| | | | | | |

A, B, C, A, D, C, ...

↑
Current miss



Learning Step (contd.)

Basic Organization

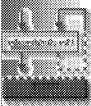
| Tag | Succ Level 1 | |
|-----|--------------|---|
| A | B | D |
| B | C | |
| C | A | |
| D | | |

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|---|--------------|--|-----|
| A | B | D | C | | ... |
| B | C | | A | | |
| C | A | | D | | |
| D | | | | | |

A, B, C, A, D, C, ...

↑
Current miss



Learning Step (contd.)

Basic Organization

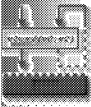
| Tag | Succ Level 1 | |
|-----|--------------|---|
| A | B | D |
| B | C | |
| C | A | |
| D | C | |

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|--------------|---|--------------|--|-----|
| A | B | D | C | | ... |
| B | C | | A | | |
| C | A | | D | | |
| D | C | | | | |

A, B, C, A, D, C, ...

↑
Current miss

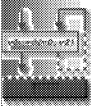


Prefetching Step

Basic Organization

| Tag | Succ Level 1 | |
|-----|--------------|---|
| A | B | D |
| B | C | |
| C | A | |
| D | C | |

- On miss A, B and D are prefetched
- Not far ahead prefetching
- Low coverage

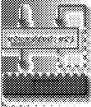


Prefetching Step (contd.)

Basic Organization + chaining

| Tag | Succ Level 1 | |
|-----|-----------------|---|
| A | B | D |
| B | C | |
| C | A | |
| D | C | |

- On miss A, B,D, and C are prefetched
- Pointer chasing
- High response time

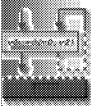


Prefetching Step (contd.)

Advanced Organization

| Tag | Succ Level 1 | | Succ Level 2 | | |
|-----|-----------------|---|-----------------|--|-----|
| A | B | D | C | | |
| B | C | | A | | |
| C | A | | D | | ... |
| D | C | | | | |

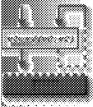
- On miss A, B,D, and C are prefetched
- Far ahead prefetching
- High coverage
- Timely prefetches
- Low response time



Evaluation Environment

■ Applications

- SPECint2000, SPECfp2000, NAS, Olden, SparseBench
- Numerical apps: CG, Equake, FT, Sparse, Tree
- Non-numerical apps: Gap, Mcf, MST, Parser
- Mostly irregular (except CG)



Evaluation Environment (contd.)

■ Main processor:

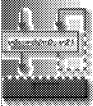
- 1.6 GHz, 6-issue out of order
- L1: 2-way 16 KB; L2: 4-way 512 KB
- Memory: 243 cycle round-trip (RT)

■ Memory processor:

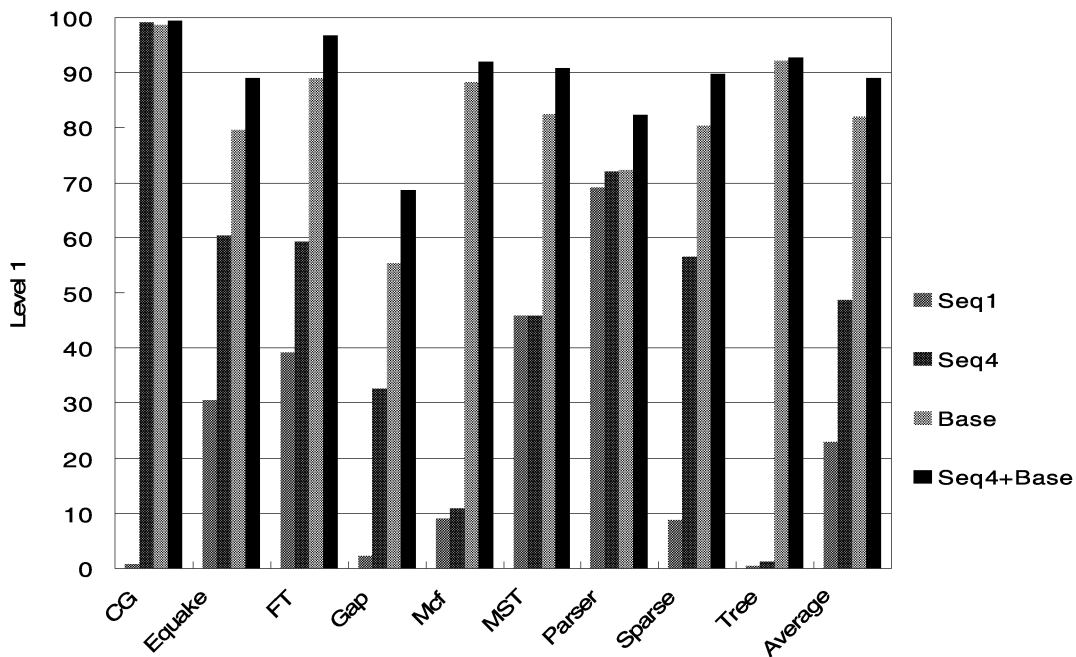
- 800 MHz, 2-issue out of order
- L1: 2-way 32 KB
- Memory: 100 cycle RT (in North Bridge chip), 56 cycle RT (in DRAM)

■ Correlation table:

- Application dependent, e.g. 64K entries, 3 levels, 2 successors



Predictability of Miss Sequences

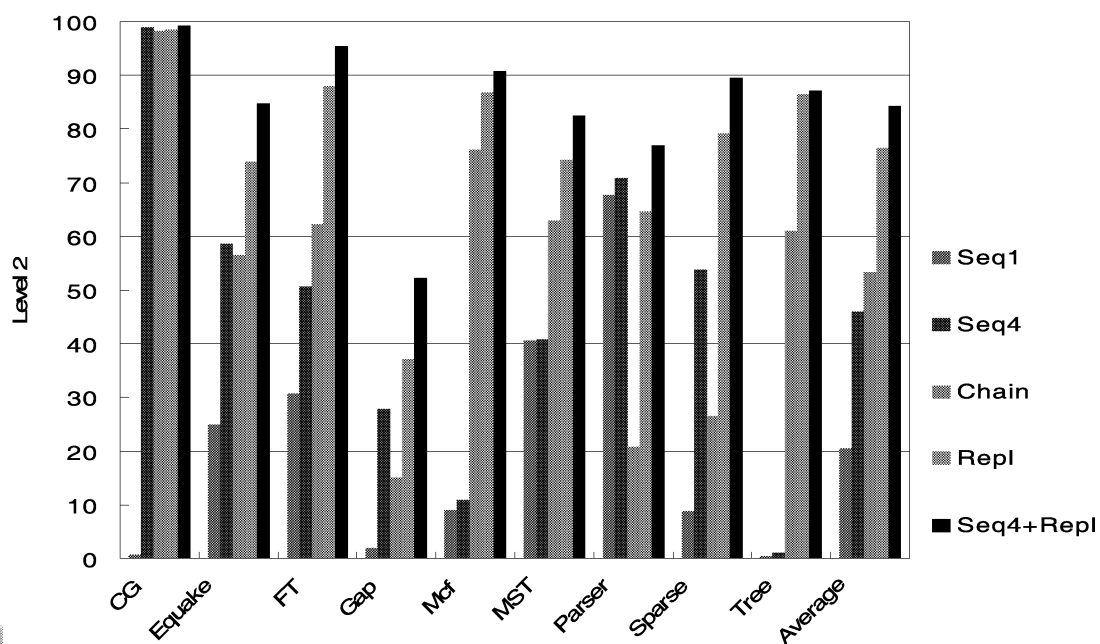


Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

63



Predictability of Miss Sequences (contd.)

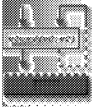
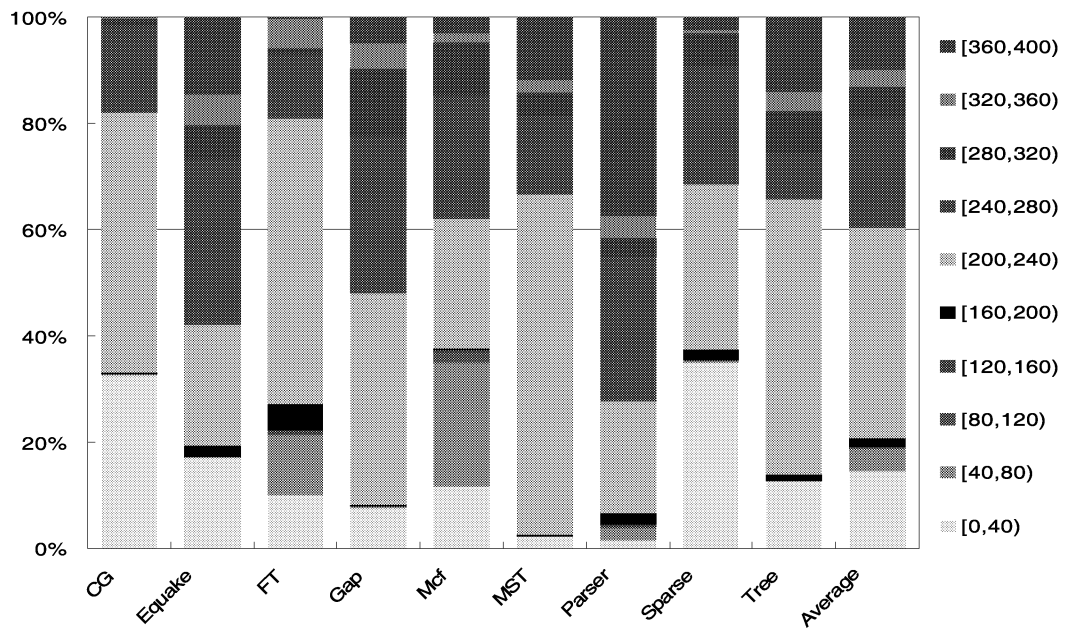


Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

64



Miss Distance

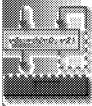
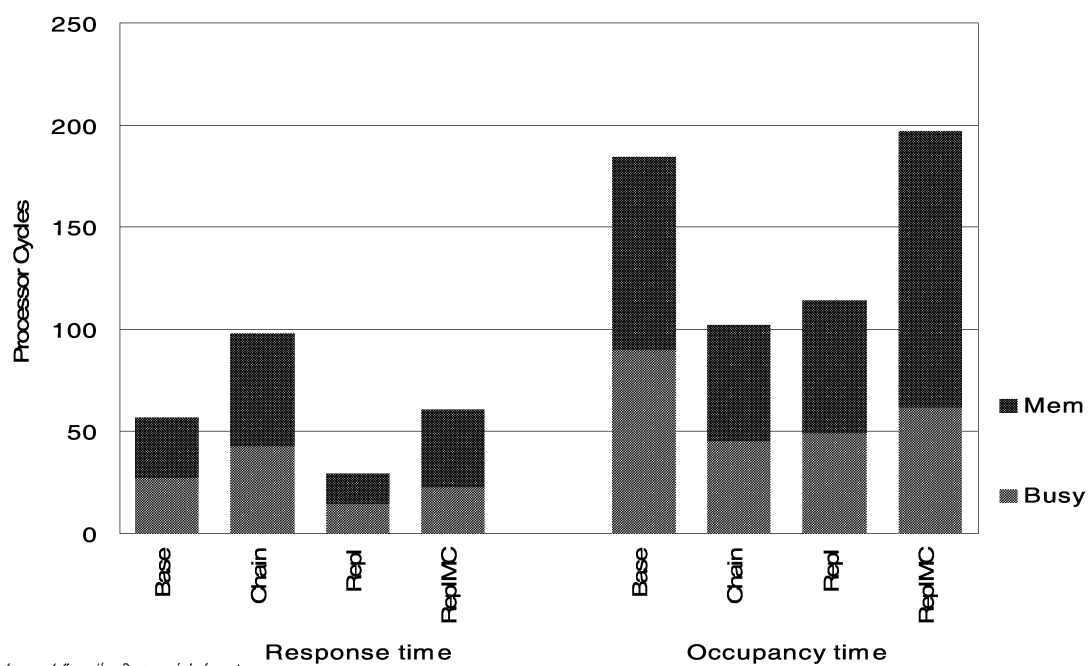


Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

65



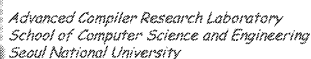
Response and Occupancy Time



Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University

66

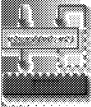
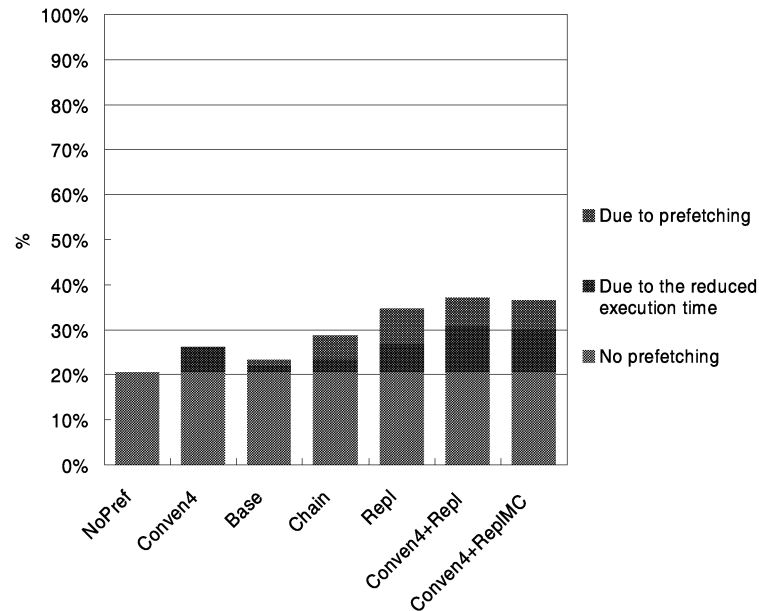




The diagram shows a laser beam incident on a sample. The sample is mounted on a stage. A detector is positioned to receive the signal from the sample. The signal is then processed by a computer, which is connected to the detector via a cable. The computer is labeled with the text 'v1=0, v2=1'.

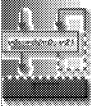


Bus Utilization



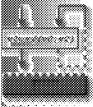
Summary (Co-execution and Prefetching)

- Co-execution
 - Static and dynamic algorithms implemented in a compiler
 - Exploiting the heterogeneity
 - The performance is better than a more expensive 2-SMP system
 - Non-numerical applications: average speedup of **1.2**
 - Numerical applications: average speedup of **1.7**
- Prefetching
 - Effectively prefetches for irregular applications
 - Average speedup of **1.53**
 - Little hardware cost
 - Very flexible

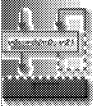


Other Helper Threads?

- Branch prediction
- Precomputation
- Speculation
- ...

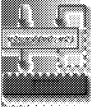


Link-Time Optimizations



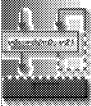
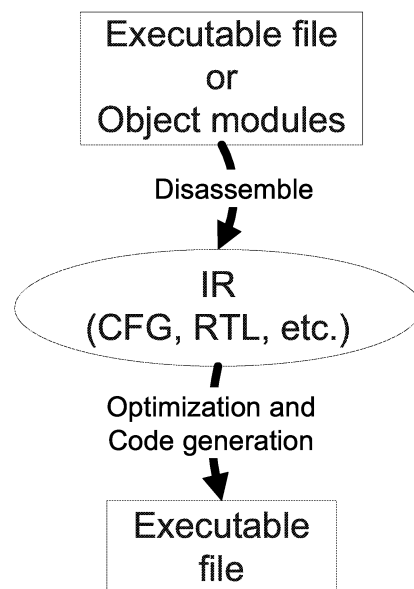
Limitation of Conventional Optimizing Compilers

- Only one module at a time
 - Compile one module without knowing much about its companion modules
 - Can't see library code
- Separate phases of compilation
 - Machine independent and dependent parts
 - Produce assembly code
- Interprocedural analysis?
 - Limited to code that is available at compile time



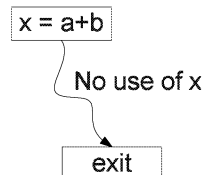
Optimization at Link Time

- Need interprocedural analysis
- Machine dependent
- Advantages
 - Whole program analysis is possible
 - Can do machine level optimizations
 - Independent from compilers and source languages
- Disadvantages
 - High-level information has been lost
 - E.g., type information
 - Much more difficult to find control-flow or data-flow information
 - E.g., jump tables for case or switch statements
 - Executable programs are significantly larger than the source programs



Dead Code Elimination

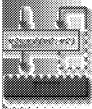
- An instruction is dead if it computes only values that are not used on any executable path leading from the instruction
- Liveness analysis needed
- Reduce the amount of code



```
foo:
store R13, 20(sp)
... def/use of r13
call bar
... R13 not used
... the result of bar not used
load R13,20(sp)
return
```

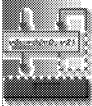
```
bar:
store R13, 20(sp)
...
... def/use of r13
...
R1 := result
load R13,20(sp)
return
```

R13: callee saved register
R1: reserved for return values



Unreachable Code Elimination

- Unreachable code: code that will never be executed
- Compile time
 - Debugging statements turned off
 - Result of other optimizations
- Link-time
 - Irrelevant library routines
 - Identified as unreachable due to the propagation of actual parameters
- Reduce the amount of code
- Reduce instruction cache pollution



Constant Propagation

- Given an assignment $x=c$ for a variable x and a constant c , replace later uses of x with uses of c as long as intervening assignments have not changed the value of x

- E.g., the value of gp is determined at link time

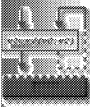
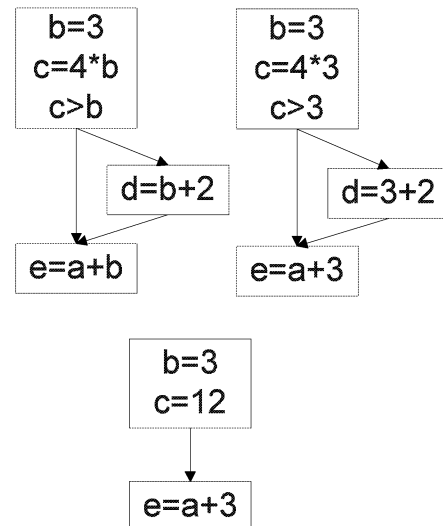
- Good for RISC

- Immediate operand

- Save registers

- Enable other optimizations

- Constant folding
- Induction variable optimizations
- Dependence analysis based optimizations



Constant Folding

- Evaluating expressions at compile or link time, whose operands are known to be constant

- e.g., the value of $R31$ is always 0 in a particular architecture, the value of gp is known at link time, read-only region

- Must preserve the semantics of exceptions

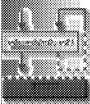
```
ldq R1, 16(gp)
ldq R2, 64(gp)
ldq R3, 48(gp)
ldq R4, 0(R1)
ldq R5, 0(R2)
addq R4, R5, R6
stq R6, 0(R3)
```

```
ldq R1, 16(gp)
lda R2, 8(R1)
lda R3, 16(R1)
ldq R4, 0(R1)
ldq R5, 8(R1)
addq R4, R5, R6
stq R6, 16(R1)
```

```
ldq R1, 16(gp)

ldq R4, 0(R1)
ldq R5, 8(R1)
addq R4, R5, R6
stq R6, 16(R1)
```

gp points to global address table whose area is read only



Loop Invariant Code Motion

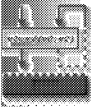
- Loop invariant code: computations in loops that produce the same value on every iteration of the loop and moves them out of the loop
 - e.g., address computations
- Need UD/DU-chains
- Interprocedural loop invariant code motion at link time

```
do i=1, 100
  L=i*(n+2)
  do j=1,100
    a(i,j)=100*n+10*L+j
  enddo
enddo
```

```
t1=10*(n+2)
t2=100*n
do i=1, 100
  t3=t2+i*t1
  do j=1,100
    a(i,j)=t3+j
  enddo
enddo
```

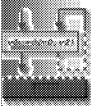
```
foo:
L1:
...
call bar
...
goto L1:
return
```

```
bar:
...
Invariant
code
...
return
```



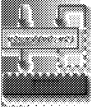
Other Link Time Optimizations

- Elimination of unnecessary memory operations
 - Escaping variables
 - Aliasing
 - Register pressure
- Inlining
 - Function call overhead
- Code layout change
 - Cache locality
 - Branch prediction
- Instruction scheduling
- ...



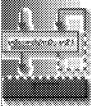
Bibliography

- Wayne Wolf. "How Many System Architectures", *IEEE Computer*, March 2003.
- Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. "Scalable Processors in the Billion-Transistor Era: IRAM", *IEEE Computer*, September 1997.
- Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. "A Single-Chip Multiprocessor", *IEEE Computer*, September 1997.
- Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. "Baring It All to Software: Raw Machines", *IEEE Computer*, September 1997.
- Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. "Simultaneous Multithreading: A Platform for Next-Generation Processors", *IEEE Micro*, September/October 1997.



Bibliography

- Yan Solihin, Jaejin Lee, and Josep Torrellas. "Correlation Prefetching with a User-Level Memory Thread", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 6, pp. 563-580, June 2003.
- Yan Solihin, Jaejin Lee, and Josep Torrellas. "Using a User-Level Memory Thread for Correlation Prefetching", In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 2002.
- Yan Solihin, Jaejin Lee, and Josep Torrellas. "Automatic Code Mapping on an Intelligent Memory Architecture", *IEEE Transactions on Computers*, Vol. 50, No 11, pp. 1248-1266, November 2001.
- Jaejin Lee, Yan Solihin, and Josep Torrellas. "Automatically Mapping Code in an Intelligent Memory Architecture", In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, January 2001.
- Amitabh Srivastava and David W. Wall. "Link-Time Optimization of Address Calculation on a 64-bit Architecture", In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1994.
- Amitabh Srivastava and David W. Wall. "A Practical System for Intermodule Code Optimization at Link-Time", *Journal of Programming Languages*, pp1-18, March 1993.
- Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. "Alto: A Link-Time Optimizer for the Compaq Alpha", *Software - Practice and Experience*, Vol. 31, pp 67-101, January 2001.

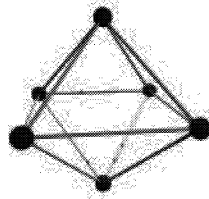


Dr. – Ing. Daniel Kästner

AbSint
Angewandte Informatik GmbH
kaestner@absint.com

Compilation for Embedded Processors

Advanced Techniques



Dr.-Ing Daniel Kästner
kaestner@absint.com

AbsInt
Angewandte Informatik GmbH

Acyclic and Cyclic Scheduling

- Scheduling of acyclic code:
 - List scheduling: basic blocks
 - Trace/Superblock scheduling: sequences of basic blocks
- In the presence of loops: cyclic scheduling
 - Unroll the loop n times and then schedule the body of the new loop. Drawbacks:
 - code growth
 - no overlapping across back edge
 - Scheduling loops for overlapping executions of several consecutive iterations: Software Pipelining.

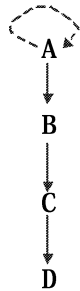
AbsInt
Angewandte Informatik



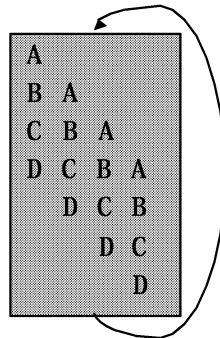
2

Loop Unrolling and Software Pipelining

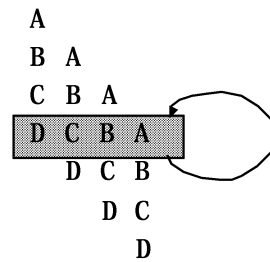
Data Dependence Graph:



After Loop Unrolling (4x)



After Software Pipelining



AbsInt
Angewandte Informatik



3

Terminology

- Operation: Machine operation, e.g. **add**, **load**, **store**. Names: a, b, c, ...
- Instruction: Set of machine operations scheduled at the same position. Names: A, B, C, ...
- Latency: Execution time of an operation.
- Delay: Distance between two consecutive dependent operations.
- Schedule: Mapping from operations to positions (cycles). Names: σ , ...

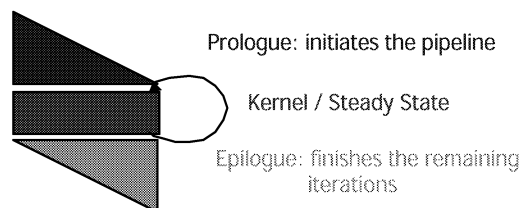
AbsInt
Angewandte Informatik



4

Goal of Software Pipelining

- Given:
 - loop with body L and λ iterations
 - p -times parallel architecture
- Wanted:
 - Efficient parallel schedule for L^λ .
 - L^λ is transformed into $\alpha\kappa\omega$ where κ is the body of a new loop, α the prologue (prelude) and ω the epilogue (postlude).



Constraints

- Precedence constraints (data dependences)
- Resource constraints
- All operations from the body L occur equally often in the kernel κ , p times.
- Width of $\kappa \leq p$

- Goal: $|\kappa|$ minimal.

Approaches

- Move-then-schedule: move code forwards/backwards over loop back edge to improve schedule.
Problems: Which operations to be moved, in which direction and how many times.
- Schedule-then-move: find a schedule and transform code accordingly.
 - Unroll-while-scheduling: Kernel recognition.
Problems: Complex information to be maintained, complex checks are required, no reliable implementation?
 - Modulo-Scheduling: Generate and solve a set of modulo constraints.
Problem: No control flow inside loops allowed (if conversion required).

Terminology and Properties of the Kernel

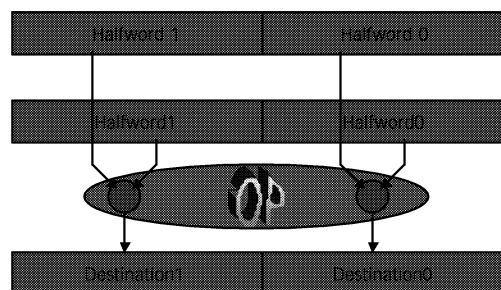
- Span μ of the kernel κ : number of consecutive iterations of L of which operations are contained in κ
(in general: different from ρ).
- Initiation interval $\Pi = |\kappa|$ is the distance between two successive iterations of the new loop.
- Observation:
 - Prelude starts $\mu-1$ iterations and postlude finishes $\mu-1$ iterations.
 - Number k of kernel iterations:

$$k = \frac{1-2(\mu-1)}{r}$$

Modulo Scheduling

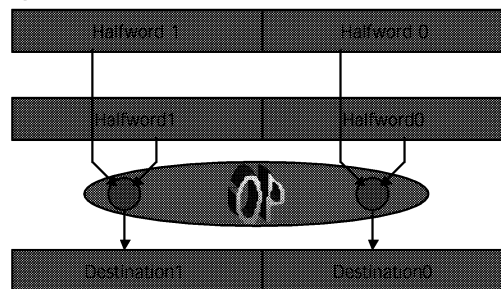
- Goal: Compute a schedule for one iteration of the loop so that when it is repeated at regular intervals, no intra-or inter-iteration dependences are violated and no resource conflicts arise.
- Basic steps:
 1. compute a lower bound for the initiation interval II
 2. find a schedule
 3. generate the kernel code
 4. generate prologue and epilogue code

SIMD Instructions and Superword-Level Parallelism



SIMD Instructions

- Developed originally because of increasing focus on multimedia applications.
- SIMD: Single Instruction Multiple Data.
- SIMD-Instruction: instruction operating concurrently on data that are packed in a single register or memory location.



SIMD Instructions

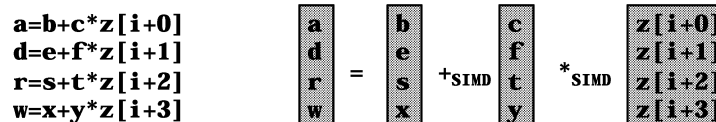
- Past: mostly used for small data types (8-bit, 16-bit)
- Future: 128-bit registers enabling simultaneous access to 4 32-bit words.
- SIMD execution units currently are appearing in desktop microprocessors for several reasons:
 - simple control
 - replicated functional units often already available
 - no heavily ported register file required⇒ simple, amenable to scaling.

Exploiting SIMD Instructions

1. Inline assembly
2. Specialized library calls
3. Vectorization techniques developed to parallelize scientific code for vector machines.
 - Pro: well-understood, many applications are vectorizable
 - Contra:
 - often very difficult since complex loop transformations might be necessary
 - incapable of locating SIMD-style parallelism with basic blocks
 - requires large amounts of parallelism to achieve speed-up
4. Algorithms for superword-level parallelism extraction

Superword-Level Parallelism

- Superword-Level Parallelism (SLP): Form of SIMD-parallelism in which source and result operands of a SIMD instruction are packed in larger storage words.
- Isomorphic statements: statements that contain the same operations in the same order.
- Detection of SLP: identification of independent isomorphic statements within a basic block. After statement packing they can be executed in parallel.



Cost and Benefit of SLP

- Result of packed computation is also packed
 - ⇒ Unpacking may be required
 - ⇒ Performance benefit of statement packing = speedup from parallelization *minus* cost of packing and unpacking
- Performance degradation is possible e.g. when packing and unpacking costs are high relative to ALU operations.

Goals of SLP Detection

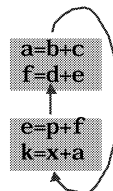
- Minimize packing and unpacking:
 - look for cases where packed data produced as a result in one computation can be directly used in another computation.
- Locate adjacent memory references:
 - if packed statements contain adjacent references among corresponding operations, operands are effectively packed in memory ⇒ no reshuffling within a register is needed
 - one address calculation followed by a load/store is required instead of one individual calculations and loads and stores per element.

Algorithm for Exploiting SLP

1. Loop unrolling: transform vector parallelism into SLP
2. Alignment analysis: determine address alignment of each load/store (for architectures not supporting unaligned memory access)
3. Transformation of IR into a low-level IR
4. Application of standard compiler optimizations to remove redundant computations and spurious data dependences.
5. Core of algorithm:
 1. Locate statements with adjacent memory references and pack them into groups of size 2.
 2. Discover more groups based on the active set of packed data.
 3. Merge all groups into larger clusters of a size consistent with the superword datapath width.
 4. Produce a new schedule for each basic block where groups of packed statements are replaced by SIMD instructions.

Scheduling

- Statements within a group must be independent and can be executed in parallel.
- Executing two groups can cause dependence violations:
 - A dependence edge is drawn from group g_1 to g_2 if a statement in g_2 is dependent from a statement in g_1 .
 - A (non-trivial) cycle in this graph indicates that the set of chosen groups is invalid and at least one group has to be eliminated. This can be expected to occur seldomly in practice.



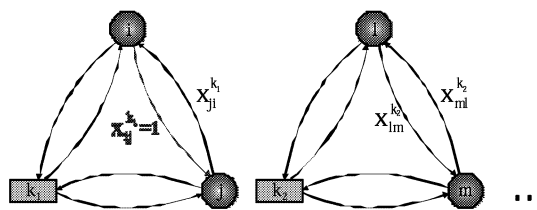
Scheduling

- When a group of packed statements is scheduled, a new SIMD instruction is emitted. If this new instruction requires operand packing, reshuffling the necessary operands is performed first. Similarly, if any statements require unpacking of source operands.
- The algorithm assumes that all data are in unpacked configuration upon entry to the block
 - ⇒ all variables that are alive on exit are unpacked at the end of the block
 - ⇒ improvement by dedicated analysis possible.

Architectural Limitations to SLP

- Complex CISC-like instructions that are hard to exploit by automatic code generation methods at all.
- Often SLP is only viewed as multimedia extension and only implemented for a subset of the instruction set, e.g. not for floating-point operations.
- Restricted interconnectivity between register banks limit possible data moves.
- Bad support for packing and unpacking limits the achievable speedup.

Phase Coupling Problems and Code Optimization by Integer Linear Programming



The Phase Coupling Problem during Code Generation

- Main subtasks of compiler backends:
 - code selection: mapping of IR statements to machine instructions of the target processor
 - register allocation: map variables and expressions to registers in order to minimize the number of memory references during program execution
 - register assignment: determine the physical register used to store a value that has been previously selected to reside in a register
 - instruction scheduling: reorder an instruction sequence in order to exploit instruction-level parallelism and minimize pipeline stalls
 - resource allocation: assign functional units and buses to operations
- Most of these tasks are interdependent, i.e. decisions made in one phase may turn out to be suboptimal in terms of overall code quality since they impose restrictions to other phases.

The Phase Coupling Problem in Code Generation

- Solution in a perfect world: Address all problems simultaneously in a single phase. BUT:
 - How to formulate this?
 - Code selection, register allocation/register assignment and instruction scheduling in general are NP-hard problems.
 - This means that in general there is no chance to optimally solve even one single of these tasks separately.

The Phase Coupling Problem in Code Generation

- Classical approaches: isolated solution by heuristic methods (list scheduling, trace scheduling, graph coloring register allocation, etc).
 - ⇒ Problem: interdependence of code generation phases.
 - ⇒ Suboptimal combination of suboptimal partial results.
 - ⇒ Inefficient code.
- Purely heuristic phase coupling (mutation scheduling, etc.) :
 - Code quality depends on the chosen heuristics to a large degree.
 - Efficiency of the heuristics depends on the target architecture.
 - ⇒ Goal conflict between easy retargetability and high code quality.

Code Selection and Register Allocation

- The goal of code selection is to determine the cheapest instruction sequence for a subgraph of the IR. However, the code selector does not know what the real overall costs of the instruction sequence will be; it can use only estimations.
- Register allocation usually is done after code selection so the code selector typically has to assume an infinite number of registers (virtual registers).
- In consequence when estimating the cost of an instruction sequence the code selector will mostly assume register references.
- Register allocation has to cope with a finite number of registers. If there are too few registers, spill code is generated.
- However, the cost of this spill code has not been considered during code selection, so the chosen operation sequence may in fact be a bad choice since another one might have worked without spill code.

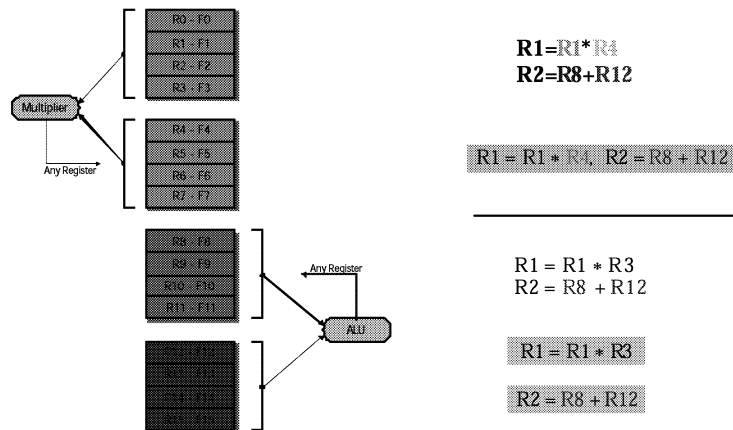
Register Allocation and Instruction Scheduling

- Goal of register allocation: minimize number of registers.
 - Consequence: false dependences caused by register reuse, limiting instruction-level parallelism.
- Goal of instruction scheduling: maximize instruction-level parallelism.
 - Consequence: parallelization often forbids register reuse, possibly triggering generation of spill code during register allocation.
- Whichever task is executed first: it can make decisions which are globally suboptimal due to restrictions they impose to the second task.

(Example for phase-coupling problem between code selection and instruction scheduling: later.)

Instruction Scheduling and Register Assignment

Analog Devices SHARC: Restricted Parallelism between ALU and multiplier.



AbsInt
Angewandte Informatik



27

Code Selection and CISC Instructions

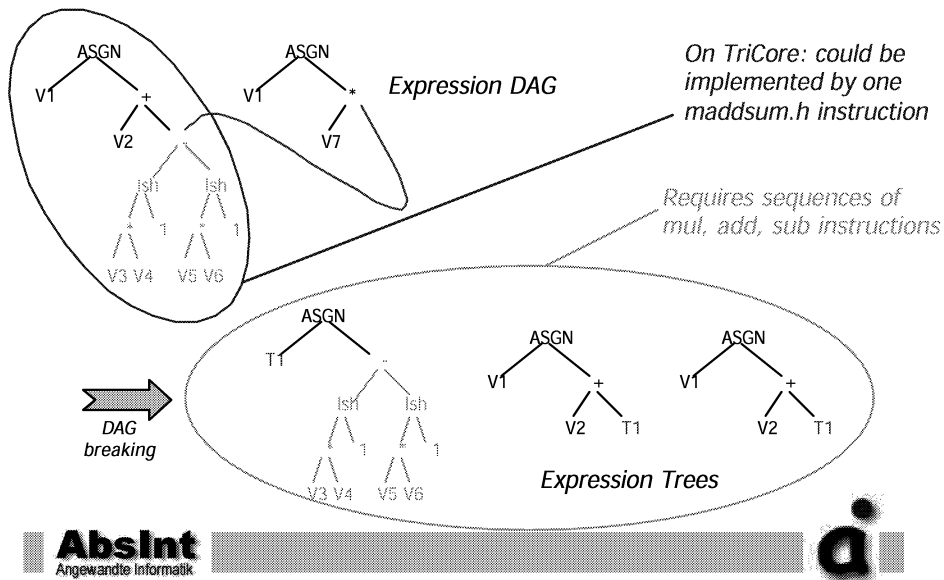
- Code selection usually is done by tree parsing (tree pattern matching) and dynamic programming.
- Usual the IR however takes the form of directed acyclic graphs, e.g. due to common subexpressions.
- Before code selection proper DAGs must be broken into trees for the code selection algorithm to work.
- Breaking the trees is done heuristically. Thus the resulting trees and the resulting use of temporary storage locations may destroy the opportunity of generating complex instructions which would correspond to larger expression trees.

AbsInt
Angewandte Informatik



28

Code Selection and CISC Instructions



29

Code Selection and Complex Data Routes

- Complex data paths can have the consequence that moving a value from one register to another takes place across a route of register sets.
 - Code selection: Breaking the expression DAG into trees can lead to choosing a temporary storage location for one tree which leads to a sequence of register moves in other trees using this value.
 - Other consequence: the cost of one subtree depends on the code selected for other subtrees. This violates the precondition for dynamic programming which is commonly used to implement code selection.

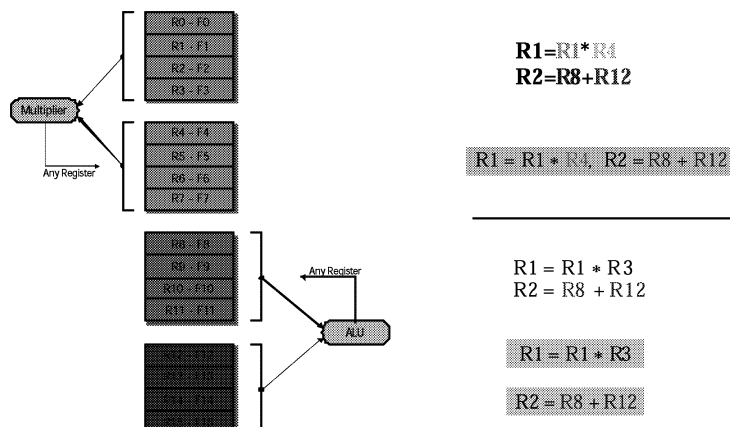
30

Code Selection and Instruction-Level Parallelism

- For architectures exhibiting instruction-level parallelism operations from different expression trees can be executed in parallel if no dependences are violated.
- This cannot be taken into account during code selection, since the code of different subtrees is not independent any more. Again this violates the independence precondition of dynamic programming. A code sequence that locally seems clumsy may be the best one since it might allow many parallelizations with operations from other expression trees.
- Consequence: the quality of code selection can be severely affected due to the phase-coupling problem between instruction scheduling and code selection.

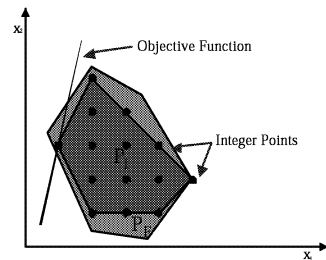
Heterogeneous register sets

Phase coupling problem between instruction scheduling and register assignment.



Integer Linear Programming

- Integer linear programming is NP-complete, thus computing a provably optimal solution requires a structured formulation.
- $P_F = P_I$: Optimal integer solution can be computed in polynomial time.
- ILP allows to incorporate several code generation phases, e.g. instruction scheduling, register assignment and resource allocation (functional unit allocation) in a homogeneous problem description and to solve them optimally.
- Optimal phase-coupled code generation can be done for hot code sequences, ILP-based approximations provide high-quality approximative solutions.



AbsInt
Angewandte Informatik



33

ILP-Structure

$$\min M_{\text{steps}}$$

$$t_j \leq M_{\text{steps}} \quad \forall j$$

Precedence Constraints

Assignment Constraints

Resource Constraints

AbsInt
Angewandte Informatik



34

ILP-Modelling Styles

- ILP-formulations in the area of code generation:
 - Time-indexed formulations: Choice of decision variables is based on the points of time the modeled events are assigned to.
 - $x_{jn}^k = 1$: operation j is started in clock cycle n by resource type k .
 - Order-indexed formulations: Decision variables reflect the ordering of the modeled events.
 - $x_{ij}^k = 1$: operation j is executed after i by resource type k .

Code Generation By Integer Linear Programming

- Well-structured ILP formulations:
 - SILP (Scheduling and Allocating with Integer Linear Programming) [Zhang,96],[DK00]: order-indexed.
 - OASIC (Optimal Architectural Synthesis with Interface Constraints) [Gebotys,Elmasry,93],[DK00]: time-indexed.

SILP

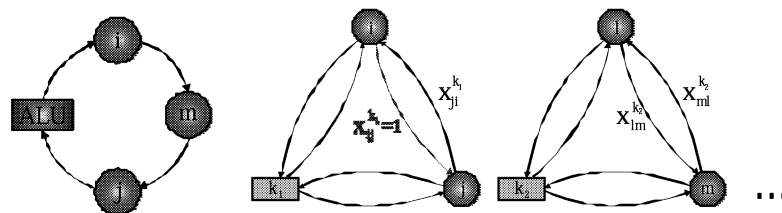
Instruction scheduling
Register assignment
Resource allocation

OASIC

Instruction scheduling
Resource allocation
Register assignment

The SILP Formulation (1)

- Order-based formulation.
- The main decision variables describe the flow of the hardware resources through the operations of the program: $x_{ij}^k \in \{0,1\}$
- The resource flow graph:



AbsInt
Angewandte Informatik



37

The SILP Formulation (2)

$$\min M_{\text{steps}}$$

$$t_j \leq M_{\text{steps}} \quad \forall j$$

$$t_j - t_i \geq w_i \quad \forall (i, j) \in E_D^i$$

$$t_j - t_i \geq w_i - w_j + 1 \quad \forall (i, j) \in E_D^o$$

$$t_i \leq t_j + w_j - 1 \quad \forall (i, j) \in E_D^a$$

Data Dependences

$$\sum_{(i,j) \in E_P^k} x_{ij}^k - \sum_{(j,m) \in E_P^k} x_{jm}^k = 0 \quad \forall j \forall k$$

$$\sum_k \sum_{(i,j) \in E_P^k} x_{ij}^k = 1 \quad \forall j$$

$$\sum_{(k,j) \in E_P^k} x_{kj}^k \leq R_k \quad \forall k$$

Flow Modelling

$$t_j - t_i \geq z_i + \left(\sum_k x_{ij}^k - 1 \right) * U \quad \forall (i, j) \in E_P^k$$

- Number of constraints: $O(n^2)$
- Number of variables: $O(n^2)$

AbsInt
Angewandte Informatik



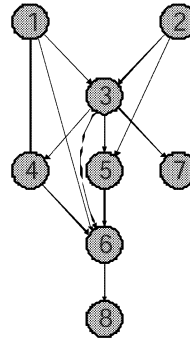
38

Example (1)

```

1  r1 = M[...];
2  r2 = M[...];
3  r3 = r1 + r2;
4  r4 = r3 * r1;
5  r5 = r2 + r3;
6  r1 = r4 + r5;
7  M[...] = r3;
8  M[...] = r1;

```



Data Dependence Graph

Functional Units:

- ALU:1
- MUL:1
- MEM:1

Execution Time: 1 clock cycle

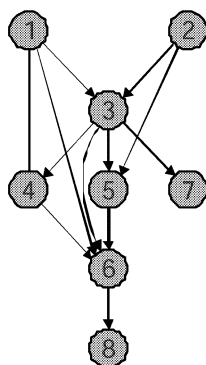
Latency: 1 clock cycle

AbsInt
Angewandte Informatik



39

Precedence Constraints



$$t_j - t_i \geq w_i \quad \forall \text{ true dependences } (i, j)^t$$

$$t_i \leq t_j + w_j - 1 \quad \forall \text{ anti dependences } (i, j)^a$$

$$t_j - t_i \geq w_i - w_j + 1 \quad \forall \text{ output dependences } (i, j)^o$$

$$t_3 - t_1 \geq 1$$

$$t_3 - t_2 \geq 1$$

$$t_4 - t_3 \geq 1$$

$$t_5 - t_3 \geq 1$$

$$t_7 - t_3 \geq 1$$

$$t_6 - t_4 \geq 1$$

$$t_6 - t_5 \geq 1$$

$$t_6 - t_1 \geq 1$$

$$t_4 \leq t_6$$

$$t_3 \leq t_6$$

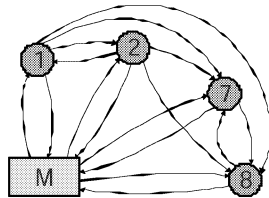
$$t_8 - t_6 \geq 1$$

AbsInt
Angewandte Informatik



40

Flow Conservation Constraints



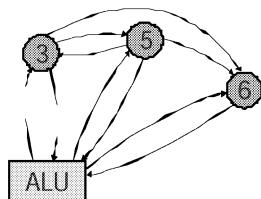
$$x_{21}^M + x_{M1}^M - x_{12}^M - x_{17}^M - x_{18}^M - x_{1M}^M = 0$$

$$x_{12}^M + x_{M2}^M - x_{21}^M - x_{27}^M - x_{28}^M - x_{2M}^M = 0$$

$$x_{17}^M + x_{27}^M + x_{87}^M + x_{M7}^M - x_{78}^M - x_{7M}^M = 0$$

$$x_{18}^M + x_{28}^M + x_{78}^M + x_{M8}^M - x_{87}^M - x_{8M}^M = 0$$

...

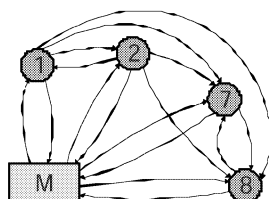


AbsInt
Angewandte Informatik



41

Assignment Constraints



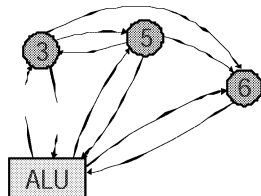
$$x_{21}^M + x_{M1}^M = 1$$

$$x_{12}^M + x_{M2}^M = 1$$

$$x_{17}^M + x_{27}^M + x_{87}^M + x_{M7}^M = 1$$

$$x_{18}^M + x_{28}^M + x_{78}^M + x_{M8}^M = 1$$

...

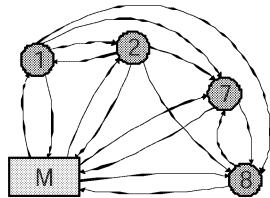


AbsInt
Angewandte Informatik



42

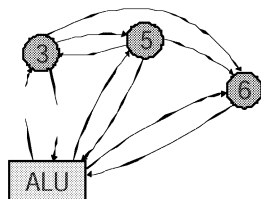
Resource Constraints



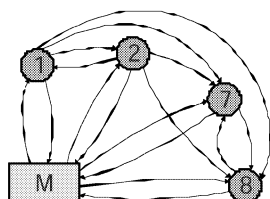
$$x_{M1}^M + x_{M2}^M + x_{M7}^M + x_{M8}^M \leq 1$$

$$x_{Alu3}^{Alu} + x_{Alu5}^{Alu} + x_{Alu6}^{Alu} \leq 1$$

$$x_{Mul4}^{Mul} \leq 1$$



Serial Constraints

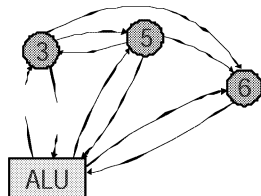


$$t_2 - t_1 \geq 1 + (x_{12}^M - 1) \cdot 8$$

$$t_1 - t_2 \geq 1 + (x_{21}^M - 1) \cdot 8$$

$$t_7 - t_1 \geq 1 + (x_{17}^M - 1) \cdot 8$$

...



The OASIC Formulation (1)

- Time-based formulation.
- The main decision variables describe the assignment of an operation's starting time to a control step and a functional unit type.
- Main decision variables: x_{jn}^k where $x_{jn}^k = 1$ means that the execution of operation j is started in control step n on an instance of functional unit type k .

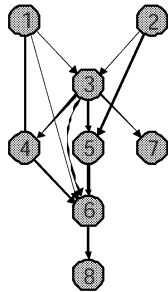
The OASIC Formulation (2)

$$\min M_{\text{steps}}$$

$$t_j = \sum_k \sum_{n=\text{asap}(j)}^{\text{alap}(j)} n \cdot x_{jn}^k \leq M_{\text{steps}} \quad \forall j$$

- Precedence Constraints
- Assignment Constraints
- Resource Constraints
- Number of constraints: $O(n^3)$
- Number of variables: $O(n^2)$

Time & Precedence Constraints



| | ASAP | ALAP |
|---|------|------|
| 1 | 1 | 4 |
| 2 | 1 | 4 |
| 3 | 2 | 5 |
| 4 | 3 | 6 |
| 5 | 3 | 6 |
| 6 | 4 | 7 |
| 7 | 3 | 8 |
| 8 | 5 | 8 |

$$t_1 = 1 \cdot x_{11}^M + 2 \cdot x_{12}^M + 3 \cdot x_{13}^M + 4 \cdot x_{14}^M \leq M_{steps}$$

$$t_2 = 1 \cdot x_{21}^M + 2 \cdot x_{22}^M + 3 \cdot x_{23}^M + 4 \cdot x_{24}^M \leq M_{steps}$$

$$t_3 = 2 \cdot x_{32}^M + 3 \cdot x_{33}^M + 4 \cdot x_{34}^M + 5 \cdot x_{35}^M \leq M_{steps}$$

...

$$x_{12}^M + x_{32}^M \leq 1$$

$$x_{13}^M + x_{32}^M \leq 1$$

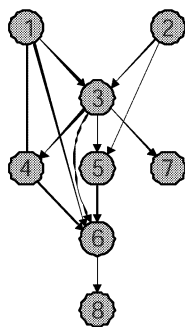
$$x_{13}^M + x_{33}^M \leq 1$$

$$x_{14}^M + x_{32}^M \leq 1$$

$$x_{14}^M + x_{33}^M \leq 1$$

$$x_{14}^M + x_{34}^M \leq 1$$

Assignment & Resource Constraints



| | ASAP | ALAP |
|---|------|------|
| 1 | 1 | 4 |
| 2 | 1 | 4 |
| 3 | 2 | 5 |
| 4 | 3 | 6 |
| 5 | 3 | 6 |
| 6 | 4 | 7 |
| 7 | 3 | 8 |
| 8 | 5 | 8 |

$$x_{11}^M + x_{12}^M + x_{13}^M + x_{14}^M = 1$$

$$x_{21}^M + x_{22}^M + x_{23}^M + x_{24}^M = 1$$

$$x_{32}^M + x_{33}^M + x_{34}^M + x_{35}^M = 1$$

...

$$x_{11}^M + x_{21}^M \leq 1$$

$$x_{12}^M + x_{22}^M \leq 1$$

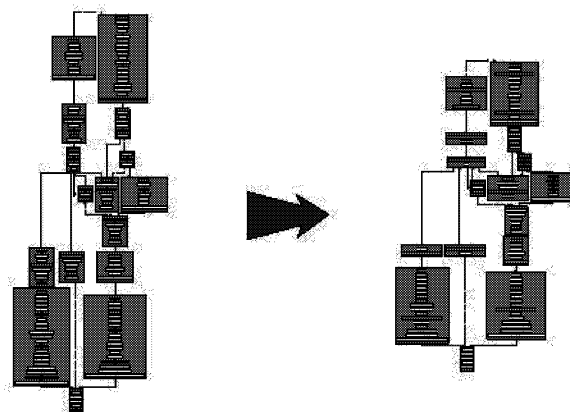
$$x_{13}^M + x_{23}^M + x_{73}^M \leq 1$$

...

Extensions

- Incorporating architectural irregularities
 - Assumption: All specified resource types can work in parallel.
 - Many processors exhibit restrictions of instruction-level parallelism and of resource usage.
 - PROPAN approach [DK03]: specifying architectural irregularities in the hardware specification (TDL) via logical formula which are automatically transformed into integer linear constraints.
- Reducing computation time:
 - Coupling with standard heuristical techniques, the ILP techniques focussing on the hotspots of the program.
 - ILP-based approximations [DK00]: Iteratively solve partial relaxations of the original problem (some integer variables may take non-integral values). In each iteration: fix some variables with an integer value to their current value. The computation time can be significantly reduced, yet high-quality solutions are obtained.

Postpass Code Compaction: The aiPop Framework



Why Code Compaction?

- Embedded systems
 - Memory is expensive
 - Price
 - Space
 - Weight
 - Power consumption
 - μ C architecture is limited, e.g.
 - 16 MBytes
 - 64 KBytes OTP

Postpass Optimizations

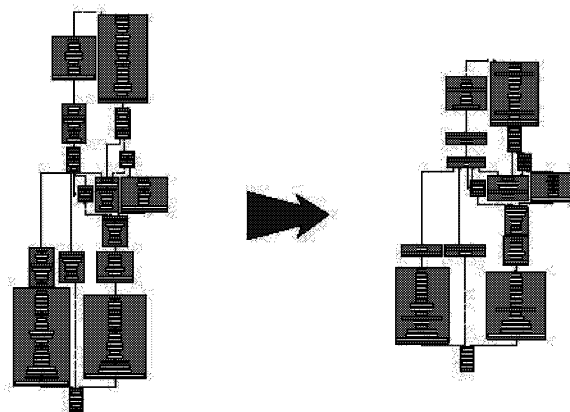
- Larger scope
 - Procedure
 - Module
 - Entire application
- Lower level
 - Standard optimizations
 - Target-architecture-dependent optimizations

Real World Requirements

- Debugging
- Speed
- Minimal interference
with the existing development process

aiPop166 - Code Compaction for C166

The aiPop166 optimizer suite was developed to reduce code size and to improve code quality of assembly files produced by a C-compiler.



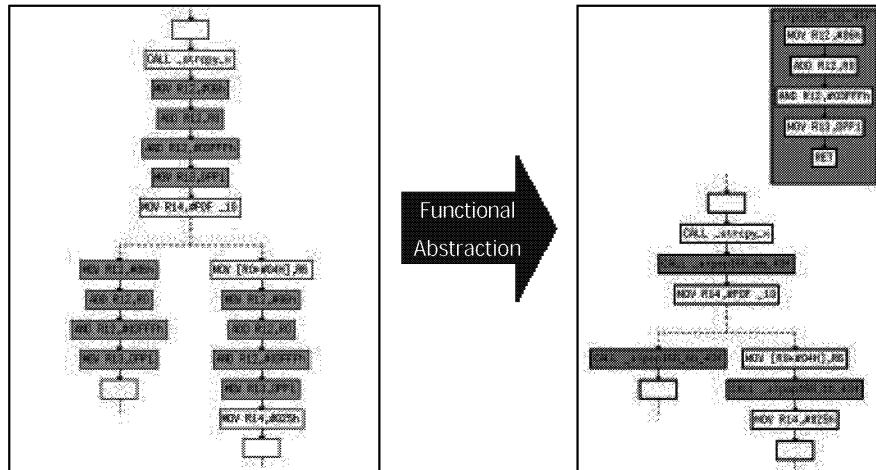
aiPop166 Features

- Functional abstraction for common basic blocks
- Tail merging for procedures
 - Optimizations based on data dependency analysis
- Interprocedural constant propagation
- Optimizations based on live variable analysis
- EXTP optimizations in memory model **LARGE**
- Peephole optimizations

Functional Abstraction, Tail Merging

1. Find multiple occurrences of instruction sequences
2. Make one representative sequence
3. Replace the occurrences by
 - function calls: functional abstraction
 - jumps between procedures: tail merging

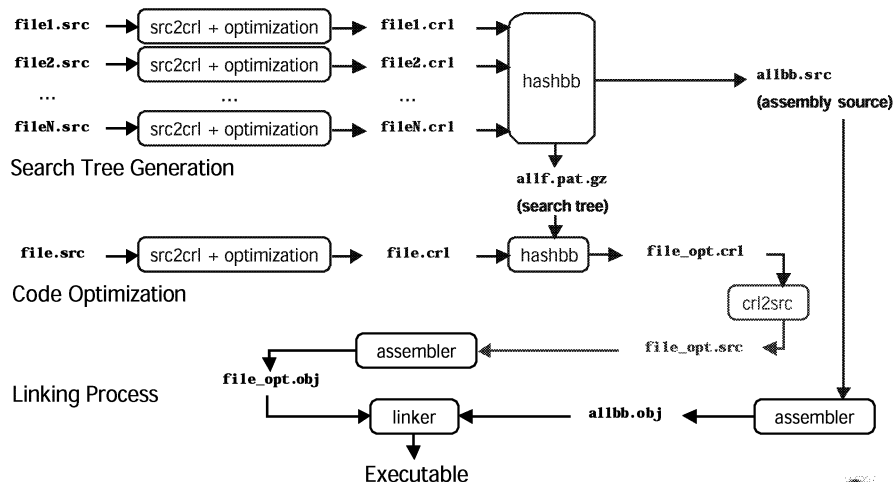
Example: Functional Abstraction



AbsInt
Angewandte Informatik

57

Integration into the Development Cycle



AbsInt
Angewandte Informatik

58

Interprocedural Constant Propagation

On C16x (large) constant values are expensive!

Example:

```
...
MOV R7,#FFFFh
ADD R3,R7
MOV R5,#FFFFh
...
```



Transformed to:

```
...
MOV R7,#FFFFh
ADD R3,R7
; MOV R5,#FFFFh ; -aipop166: --opt-coreg 1
MOV R5,R7 ; +aipop166: --opt-coreg 1
...
```

MOV R5,R7 uses only
two bytes

AbsInt
Angewandte Informatik



59

Optimizations Based on DDA

Redundant Cyclic Moves

Example:

```
i:  ...
    MOV Ra, Rb
    ...
j:  ...
    MOV Rb, Ra
    ...
```



Transformed to:

```
i:  ...
    MOV Ra, Rb
    ...
j:  ; MOV Rb, Ra ; -aipop166: --opt-cymo
    ...
```

j is eliminated if:

- **i** is the only definition of **Ra** reaching **j**
- **Rb** reaching the second operation **j** is uniquely defined and identical to the definition of **Rb** reaching **i**
- No flags set by **j** are used as an input to other operations

AbsInt
Angewandte Informatik



60

Combining Assignments

Typical struct assignment:

```

...
b.xpos= a.xpos;
b.ypos= a.ypos;
b.count= a.count;
...
MOV RL1, _a
MOV _b, RL1
MOV RL2, (_a+8)
MOV (_b+8), RL2
MOV RL3, (_a+2)
MOV (_b+2), RL3
...

```



```

...
MOV RL1, _a
MOV _b, RL1
MOV RL2, (_a+1)
MOV (_b+1), RL2
MOV RL3, (_a+2)
MOV (_b+2), RL3
...

```

Needs DDA information



```

...
MOV R1, WORD PTR _a
MOV WORD PTR _b, R1
MOV R3, WORD PTR (_a+2)
MOV WORD PTR (_b+2), R3
...

```

Needs free registers



```

...
MOV R1, WORD PTR _a
MOV R2, WORD PTR _b
MOV [R1], [R2+]
ADD R1, #2
MOV [R1], [R2+]
ADD R1, #2
...

```

Four bytes less per assignment

AbsInt
Angewandte Informatik



61

Compaction Rates

- 20.39% (large mobile phone application)
 - over 25% more functionality can be packed into a flash memory of the same size
- 5.3% (small application featuring highly hand-optimized C code)
- 66% on specific modules

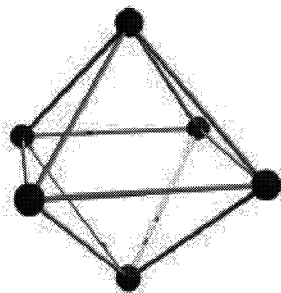
AbsInt
Angewandte Informatik



62

Tool Demonstration aiPop

Program Analysis and Abstract Interpretation



Program Analysis Goals

- Automatically computing information about a program
 - Checking for possible program optimizations
= replacing the program with a "better" one that does "the same"
 - Generating "intelligent" error messages
 - Validating other aspects (e.g. timing)

Other Examples

- Elimination of useless assignments
- Loop invariant code motion
- Cache analysis
- Pipeline analysis
- Stack analysis
- Worst-Case Execution Time Analysis
- Other safety-critical analyses
- ...

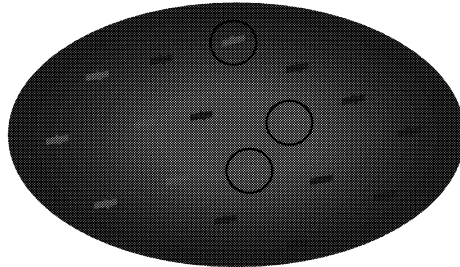
Dynamic PA vs Static PA

- **Dynamic Program Analysis:**
collecting information by means of testing
 - Disadvantage:
generated code still must cover all possible inputs
- **Static Program Analysis:**
generating information without executing the program
 - Optimizations hold for any given input

Static Program Analysis

- Determination of runtime properties at compile time
- Most of the (interesting) properties are undecidable:
The results cannot always be correct and exact => approximations
- An approximate program analysis is safe, if its results can always be depended on. Results are allowed to be imprecise as long as they are on the safe side
- Quality of the results (precision) should be as good as possible

Exact Answers



Property
holds

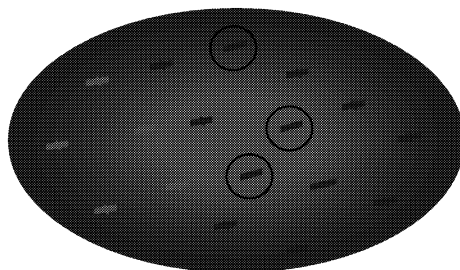
Property
does not hold

AbsInt
Angewandte Informatik



69

Approximations



Property
definitely holds

Property
might not hold

→ Erring on the safe side

AbsInt
Angewandte Informatik

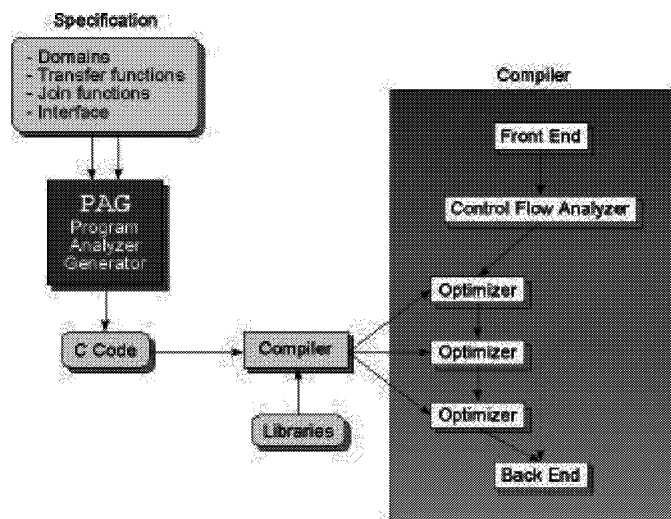


70

Abstract Interpretation

- Semantics-based methodology for program analyses; introduced 1977 by Patrick Cousot and Radhia Cousot.
- Basic idea: Perform the program's computations using value descriptions or abstract values in place of the concrete values.
 - to ensure that analysis results are obtained in finite time
 - to obtain results that describe the result of computations on a set of possible inputs
- Supports systematic derivation of program analyses
- Supports correctness proofs
- Tool support (e.g. PAG, the Program Analyser Generator)

PAG: The Program Analyser Generator



Cache Memories and Real-Time Systems

- On hardware with caches: worst case assumption every access is a cache miss
- Worst case timings are far away from realistic timings. This leads to a waste of hardware resources.
- The degree of success of timing validations depends on precise predictions.
- Software monitoring, dual loop benchmark, direct measurement with logic analyser, hardware simulation are no longer generally applicable.
- Choosing the fastest available processor, praying, or crossing fingers is not a true alternative.

Cache Analysis by Abstract Interpretation

the semantics $\xrightarrow{\text{determines}}$ set of all cache states for each program point



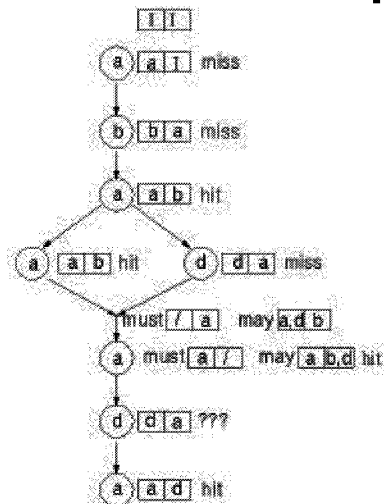
"cache" semantics $\xrightarrow{\text{determines}}$ set of all cache states for each program point



conic

abstract semantics $\xrightarrow[\text{PAG}]{\text{determines}}$ abstract cache states for each program point

Cache Analysis by Abstract Interpretation



Example:
Fully Associative Cache (2
Elements)

Result of the Analyses

Categorization of memory references

| Category | Abb. | Meaning |
|----------------|-----------|---|
| always hit | ah | The memory reference will always result in a cache hit. |
| always miss | am | The memory reference will always result in a cache miss. |
| not classified | nc | The memory reference could neither be classified as ah nor am . |

Tool Demonstration aiT WCET Analyzer

Bibliography

- **Compiler Design:**
 - [ASU86] Aho, Sethi, Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.
 - [WiMa95] Wilhelm, Maurer. Compiler Design. Addison-Wesley, 1995.
 - [M97] Muchnick. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, 1997.
 - [SS03] The compiler design handbook: Optimizations and Machine Code Generation. Ed. Srikant, Shankar. CRC Press, 2003.

Bibliography

■ Code Selection:

- [AJ76] Aho, Johnson. Optimal Code Generation for Expression Trees. Journal of the ACM, vol 23, no 3, 1976.
- [GG78] Glanville, Graham. A new Method for Compiler Code Generation. Proceedings of the 5th ACM Symposium on Principles of Programming Languages, 1978.
- [AG85] Aho Ganapathi. Efficient Tree Pattern Matching: An Aid to Code Generation. Proceedings of the 12th ACM Symposium on Principles of Programming Languages, 1985.
- [FSW94] Ferdinand, Seidl, Wilhelm. Tree Automata for Code Selection. Acta Informatica, vol 31, 1994.

Bibliography

■ Register Allocation:

- [Cha82] Chaitin. Register Allocation and Spilling via Graph Coloring. Proceedings of the SIGPLAN'82 Symp. on Compiler Construction. SIGPLAN Notices, vol 17, no 6, 1982.
- [CH84] Chow, Hennessy. The Priority-Based Coloring Approach to Register Allocation. ACM Transactions on Programming Languages and Systems, vol. 12, no 4, 1990.
- [Bri92] Briggs. Register Allocation via Graph Coloring. Phd Thesis, Rice University, 1992.
- [BCT94] Briggs, Cooper, Torczon. Improvements to Graph Coloring Register Allocation. ACM Transactions on Programming Languages and Systems, vol 16, no 3, 1994.

Bibliography

■ Instruction Scheduling and Parallelization:

- [LDSM80] Landskov, Davidson, Shriver. Local Microcode Compaction Techniques. ACM Computing Surveys, vol 12, no 3, 1980.
- [Fis81] Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, vol 20, no 7, 1981.
- [AJLA95] Allan, Jones, Lee, Allan. Software Pipelining. ACM Computing Surveys, 1995.
- [R96] Rau. Iterative Modulo Scheduling. International Journal of Parallel Processing, vol 24, 1996.
- [LA00] Larsen, Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. ACM SIGPLAN Notices, 2000.

Bibliography

■ Retargetable Compilation and Optimization:

- [FraHan91b] Fraser, Hanson. A Retargetable Compiler for ANSI C. SIGPLAN Notices, vol 26, no 10, 1991.
- [FraHan95] Fraser, Hanson. A Retargetable C Compiler: Design And Implementation. Benjamin/Cummings Publishing Company, Inc., 1995.
- [DaFra80] Davidson, Fraser. The Design and Application of a Retargetable Peephole Optimizer. ACM Transactions on Programming Languages and Systems, vol 2, no 2, 1980.
- [DaFra84] Davidson, Fraser. Code Selection through Object Code Optimization. ACM Transactions on Programming Languages and Systems, vol 6, no 4, 1984.
- [BeDa88] Benitez, Davidson. A Portable Global Optimizer and Linker. Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, in SIGPLAN Notices, vol 23, no 7, 1988.
- [BeDa94] Benitez, Davidson. Target-Specific Global Code Improvement: Principles and Applications. Department of Computer Science, University of Virginia, 1994.
- [Sta98] Stallman. Using and Porting GNU CC. Free Software Foundation, 1988.
- [Tri98] TRIMARAN: An Infrastructure for Research in Instruction-Level Parallelism. <http://www.trimaran.org>.
- [COSY] Ace Associated Computer Experts. <http://www.ace.nl/products/cosy.htm>

Bibliography

- Retargetable Compilation and Optimization for Embedded Processors:
 - [CHESS95] Lanneer, Van Praet, Kifli, Schoofs, Geurts, Thoen, Goossens. CHESS: Retargetable Code Generation For Embedded DSP Processors. In [MaGo95]. Kluwer Academic Publishers, 1995.
 - [MaGo95] Marwedel, Goossens, G. Code Generation for Embedded Processors. Kluwer Academic Publishers, 1995.
 - [Le97] Leupers. Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997.
 - [HTGDN99] Halambi, Tomiyama, Gruen, Dutt Nicolau. Automatic Software Toolkit Generation for Embedded Systems-on-Chip. Proceedings of the 1999 International Conference on VLSI and CAD (ICVC99), 1999.
 - [DK00] Kästner. Retargetable Postpass Optimisation by Integer Linear Programming. Saarland University, 2000.
 - [DK01] Kästner. ILP-based Approximations for Retargetable Code Optimization. Proceedings of the 5th International Conference on Optimization: Techniques and Applications, Hong Kong, 2001.

Bibliography

- Architecture Description Languages:
 - [Emm89] Emmelmann. BEG -- a Back End Generator. GMD Forschungsstelle an der Universitaet Karlsruhe, 1989.
 - [LSU93] Lipsett, Schaefer, Ussery. VHDL: Hardware Description and Design. Kluwer Academic Publishers, 1993.
 - [TM95] Thomas, Moorby. The Verilog Hardware Description Language. Kluwer Academic Publishers, 1995.
 - [FaPraFre95] Fauth, Van Praet, Freericks. Describing Instruction Set Processors Using nML. Proceedings of the European Design and Test Conference. IEEE, 1995.
 - [BCRS97] Bodin, Chamski, Rohou, Sezec. Functional Specification of SALTO. A Retargetable System for Assembly Language Transformation and Optimization, rev. 1.00 beta, INRIA, 1997.
 - [RaFe97] Ramsey, Fernandez. Specifying Representations of Machine Instructions. ACM Transactions on Programming Languages and Systems, vol 19, no 3, 1997.
 - [DaRa98] Davidson, Ramsey. Machine Descriptions to Build Tools for Embedded Systems. Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems. Springer LNCS, Volume 1474, 1998.
 - [DK03] Kaestner. TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses. Proceedings of the Second ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, 2003.

Bibliography

■ Postpass Optimizations

- [DK00a] Kästner. A Retargetable System for Postpass Optimisations and Analyses. Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools, Montreal, 2000.
- [CF00] Ferdinand. Post Pass Code Compaction at the Assembly Level for C16x. Contact, vol 3, no 9, 2000.
- [BKCP03] De Bus, Kaestner, Chanet, Van Put, De Sutter. Post-Pass Compaction Techniques. Communications of the ACM, vol 46, no 8, 08/2003.

Bibliography

■ Program Analysis

- [CC79] Cousot, Cousot. Systematic Design of Program Analysis Frameworks. Proceedings of the 6th ACM Symposium on Principles of Programming Languages POPL, 1979.
- [F97] Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD thesis, Saarland University, 1997.
- [NNH99] Nielson, Nielson, Hankin. Principles of Program Analysis. Springer, 1999.
- [M99] Florian Martin. Generation of Program Analyzers. PhD thesis, Saarland University, 1999.

Sponsored by:

