Machine-Level Representation of Programs: Data and Locality

Yipeng Huang

Rutgers University

April 2, 2024
Table of contents

Announcements

Procedures, function calls, and support for recursion
  Transfer of control to callee
  Transfer of data to callee
  Transfer of control back to caller
  Transfer of data back to caller and restoring caller state

Architecture support for recursive programming

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches
Announcements

Class session plan

- Tuesday, 4/2: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.
Table of contents
Announcements
Procedures, function calls, and support for recursion
  Transfer of control to callee
  Transfer of data to callee
  Transfer of control back to caller
  Transfer of data back to caller and restoring caller state
Architecture support for recursive programming
Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)
Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality
Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches
Procedures and function calls

To create the abstraction of functions, need to:

- Transfer control to function
- Transfer data to function (parameters)
- Transfer control back from function
- Transfer data back from function (return type)
- Restore caller context

Figure: Steps of a C function call. Image credit CS:APP
CPU and memory state in support of procedures and functions

Assembly/Machine Code View

Programmer-Visible State
- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Relevant state in CPU:
- `%rip` register / instruction pointer / program counter
- `%rsp` register / stack pointer

Relevant state in Memory:
- Stack
Procedures and function calls: Transferring data

For purposes of this class, the Bomb Lab, and the CS:APP textbook, we study the x86-64 Linux Application Binary Interface (ABI). Would be different on ARM or in Windows. So, don’t memorize this, but it is helpful for PA4 Lab.

Passing parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register / stack</th>
<th>Subset registers</th>
<th>Mnemonic(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>%rdi</td>
<td>%edi, %di</td>
<td>Diane’s</td>
</tr>
<tr>
<td>2nd</td>
<td>%rsi</td>
<td>%esi, %si</td>
<td>silk</td>
</tr>
<tr>
<td>3rd</td>
<td>%rdx</td>
<td>%edx, %dx, %dl</td>
<td>dress</td>
</tr>
<tr>
<td>4th</td>
<td>%rcx</td>
<td>%ecx, %cx, %cl</td>
<td>cost</td>
</tr>
<tr>
<td>5th</td>
<td>%r8</td>
<td>%r8d</td>
<td>$8</td>
</tr>
<tr>
<td>6th</td>
<td>%r9</td>
<td>%r9d</td>
<td>9</td>
</tr>
<tr>
<td>7th and beyond</td>
<td>Stack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^1\)http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.html
PA4 Defusing a Binary Bomb: `sscanf();`

```c
int sscanf (const char *str, // 1st arg, %rdi
            const char *format, // 2nd arg, %rsi
            ...)
```
Structure of stack for currently executing function Q()

- P() calls Q(). P() is the caller function. Q() is the callee function.
Stack instructions: **push** src and **pop** dest

![Stack diagram](image)

**Initially**

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

**pushq %rax**

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

**popq %rdx**

<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x123</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

**Increasing address**

- Stack "top" : 
- Stack "bottom" : 

**Figure:** x86-64 offers dedicated instructions to work with stack in memory. In addition to moving data, the updating of %rsp is implied. Image credit: CS:APP.
Procedure call and return: *call* and *ret*

<table>
<thead>
<tr>
<th>%rip</th>
<th>0x400563</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7ffffffffe840</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rip</th>
<th>0x400540</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7ffffffffe838</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rip</th>
<th>0x400568</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7ffffffffe840</td>
</tr>
</tbody>
</table>

(a) Executing *call*  
(b) After *call*  
(c) After *ret*

**Figure:** Effect of *call 0x400540* instruction and subsequent return. *call* and *ret* instructions update the instruction pointer, the stack pointer, and the stack to create the procedure / function call abstraction. Image credit: CS:APP.
Example in GDB

```c
#include <stdio.h>

int return_neg_one() {
    return -1;
}

int main() {
    int num = return_neg_one();
    printf("%d", num);
    return 0;
}
```

Compile, and then run it in GDB:

gdb return

In GDB, see evolution of %rip, %rsp, and stack:

- (gdb) layout split
- (gdb) break return_neg_one
- (gdb) info stack
- (gdb) print /a $rip
- (gdb) print /a $rsp
- (gdb) x /a $rsp

Step past return instruction, and inspect again:

- (gdb) stepi
- (gdb) info stack
Transfer of data back to caller

Passing function return data
Function return data is passed via:
- the 64-bit %rax register
- the 32-bit subset %eax register

Example from textbook slides on assembly procedures
Slides 33 through 38.
Restoring caller state

Structure of stack for currently executing function Q()

- P() calls Q(). P() is the caller function. Q() is the callee function.

Example from textbook slides on assembly procedures

Slides 40 through 44.
Table of contents

Announcements

Procedures, function calls, and support for recursion
- Transfer of control to callee
- Transfer of data to callee
- Transfer of control back to caller
- Transfer of data back to caller and restoring caller state

Architecture support for recursive programming

Cache, memory, storage, and network hierarchy trends
- Static random-access memory (registers, caches)
- Dynamic random-access memory (main memory)
- Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
- Spatial locality
- Temporal locality

Caches: motivation
- Hardware caches supports software locality
- Software locality exploits hardware caches
3_recursion.c: Putting it all together to support recursion

Discussion points

- Use info stack, info args in GDB to see recursion depth
- Difference between compiling with and without -g for debugging information.
- Memory costs of recursion.
- Compilers can recognize tail recursive calls to reduce memory use. Enabled with -foptimize-sibling-calls, -O2, -O3, and -Os.
Table of contents

Announcements

Procedures, function calls, and support for recursion
  Transfer of control to callee
  Transfer of data to callee
  Transfer of control back to caller
  Transfer of data back to caller and restoring caller state

Architecture support for recursive programming

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches
Assembly programming view of computer: CPU and memory.

Full view of computer architecture and systems: + caches, + storage, + network.
Cache, memory, storage, and network hierarchy trends

Topic of this chapter:
- Technology trends that drive CPU-memory gap.
- How to create illusion of fast access to capacious data.

Figure: Widening gap: CPU processing time vs. memory access time. Image credit CS:APP
Static random-access memory (registers, caches)

- SRAM is bistable logic
- Access time: 1 to 10 CPU clock cycles
- Implemented in the same transistor technology as CPUs, so improvement has matched pace.

Figure: SRAM operating principle. Image credit Wikimedia
Dynamic random-access memory (main memory)

- Needs refreshing every 10s of milliseconds
- 8GB typical in laptop; 1TB on each ilab machine
- Access time: 100 CPU clock cycles
- Memory gap: DRAM technological improvement slower relative to CPU/SRAM.

**Figure:** DRAM operating principle. Image credit ocw.mit.edu
CPU / DRAM main memory interface

Figure: Memory Bus. Image credit CS:APP

- DDR4 bus standard supports 25.6GB/s transfer rate

Figure: Intel 2020 Gulftown die shot. Image credit AnandTech
Solid state and hard disk drives (storage)

Technology

- SSD: flash nonvolatile memory stores data as charge.
- HDD: magnetic orientation.
- Access time: 100K CPU clock cycles

For in-depth on storage, file systems, and operating systems, take:

- CS214 Systems Programming
- CS416 Operating Systems Design

Since summer 2021, LCSR (admins of iLab) have moved the storage systems that supports everyone’s home directories to SSD. https://resources.cs.rutgers.edu/docs/file-storage/storage-technology-options/
I/O interfaces

Storage interfaces

- SATA 3.0 interface (6Gb/s transfer rate) typical
- PCIe (15.8 GB/s) becoming commonplace for SSD
- But interface data rate is rarely the bottleneck.

For in-depth on computer network layers, take:

- CS352 Internet Technology

Figure: I/O Bus. Image credit CS:APP
Cache, memory, storage, and network hierarchy trends

Topic of this chapter:

▶ Technology trends that drive CPU-memory gap.
▶ How to create illusion of fast access to capacious data.

Figure: Widening gap: CPU processing time vs. memory access time. Image credit CS:APP
Announcements

Procedures, function calls, and support for recursion
- Transfer of control to callee
- Transfer of data to callee
- Transfer of control back to caller
- Transfer of data back to caller and restoring caller state

Architecture support for recursive programming

Cache, memory, storage, and network hierarchy trends
- Static random-access memory (registers, caches)
- Dynamic random-access memory (main memory)
- Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
- Spatial locality
- Temporal locality

Caches: motivation
- Hardware caches supports software locality
- Software locality exploits hardware caches
Locality: How to create illusion of fast access to capacious data

From the perspective of memory hierarchy, locality is using the data in at any particular level more frequently than accessing storage at next slower level. 

First, let’s experience the puzzling effect of locality in `sumArray.c`

- `sumArrayRows()`
- `sumArrayCols()`

Well-written programs maximize locality

- Spatial locality
- Temporal locality
Spatial locality

```c
double dotProduct (  
double a[N],  
double b[N],  
) {  
double sum = 0.0;  
for(size_t i=0; i<N; i++){  
    sum += a[i] * b[i];  
}  
return sum;  
}
```

Spatial locality

- Programs tend to access adjacent data.
- Example: stride 1 memory access in `a` and `b`. 
Temporal locality

```c
double dotProduct (double a[N], double b[N]) {
  double sum = 0.0;
  for(size_t i=0; i<N; i++){
    sum += a[i] * b[i];
  }
  return sum;
}
```

Temporal locality

- Programs tend to access data over and over.
- Example: `sum` gets accessed $N$ times in iteration.
Table of contents

Announcements

Procedures, function calls, and support for recursion
  Transfer of control to callee
  Transfer of data to callee
  Transfer of control back to caller
  Transfer of data back to caller and restoring caller state

Architecture support for recursive programming

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches
CPU / cache / DRAM main memory interface

Figure: Cache resides on CPU chip close to register file. Image credit CS:APP

Figure: Intel 2020 Gulftown die shot. Image credit AnandTech

Figure: Cache stores a temporary copy from the slower main memory. Image credit CS:APP
CPU / cache / DRAM main memory interactions

Level k:

4  9  14  3

Level k+1:

0  1  2  3
4  5  6  7
8  9 10 11
12 13 14 15

Data are copied between levels in block-sized transfer units.

Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1.

Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.

When CPU loads (LD) from memory

- Cache read hit
- Cache read miss

When CPU stores (ST) to memory

- Cache write hit
- Cache write miss

Figure: Cache stores a temporary copy from the slower main memory. Image credit CS:APP