Lesson learned on Cell/B.E. for Hetero Programming Model, and alignments tweaks on RISC-V for Network speeds

Akira Tsukamoto
RISC-V Ambassador, RISC-V International
Third International workshop on RISC-V for HPC
HPC Asia 2024, January 25, 2024

Topics on
Lessen learned on Cell/B.E. for Hetero Programming Model

- Who am I
- Story in 2006: The performance issue on Cell/B.E., it was not a hardware issue
- The method to improve in programing
- Developing library without the need of manual optimization
- Providing feedback to the OpenCL specification
- Cell/B.E. played a role as a pioneer of modern GPGPU programing

Introduction: Akira Tsukamoto

- My background related on HPC
  - Lead developing library on Cell/B.E. for PS3 to provide GPGPU style programming
    - MARS (Multi Core Application Runtime System) project
  - Developed HPC of 168 IBM Cell/B.E. clustering servers
    - Prototype system for RoadRunner which became the first one petaflops supercomputer
    - Server Cluster Management System (DIM), Cluster File System (GPFS)
- Expertise
  - Cyber Physical and Supply Chain Security, Realtime communication protocol
  - Collaborative engineering developments with Communities
- Main community activities
  - Linux Foundation, Internet Engineering Task Force (IETF)

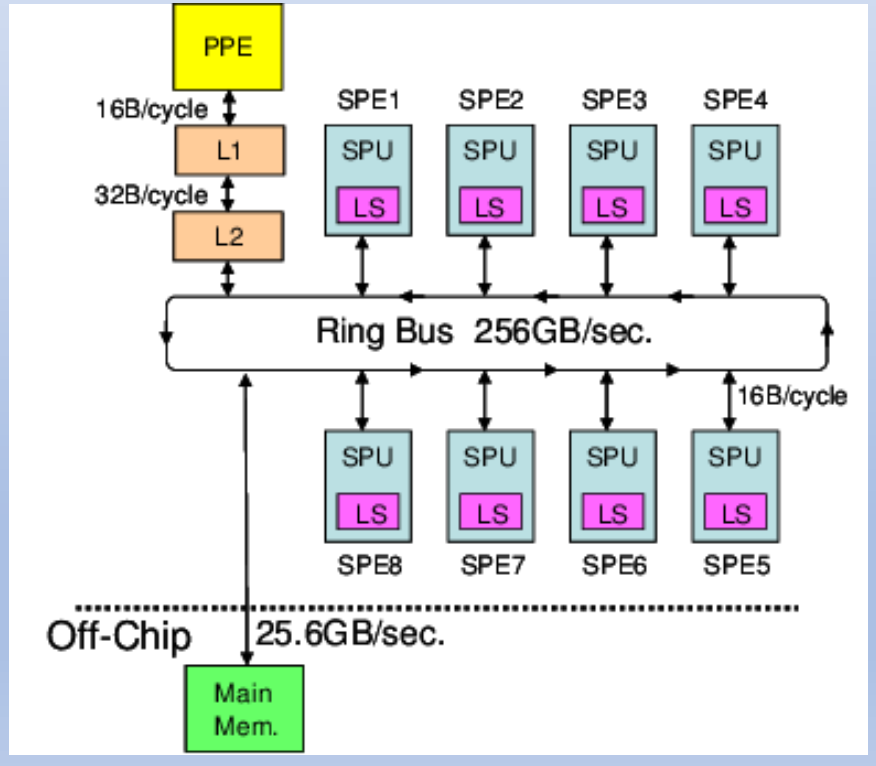
RISC-V Ambassador

- Hosting a Japanese RISC-V Technical Study meeting starting in 2019
  - A casual event for who wants to exchange or chat engineering knowledge
  - Typically, around 80 registration and 60 participants

https://risc-v.connpass.com/

Limitation of Unix style SMP programing on Cell/B.E.

- Historical Symmetric multiprocessing (SMP) programing
  - Unix Multi-Processes Programming
    - fork(), wait(), etc.
  - Unix Multi-threads Programing
    - pthread in C, std::string, boost::thread in C++
- The Multi-Process, Multi-thread only utilize PPE (PowerPC core)
- Effective utilization of SPE was the key to improve performance on Cell/B.E.
- SPE is highly optimized on SIMD instructions
  25.6 GFLOPS / SPE <- much faster
  6.4 GFLOPS / PPE (both on single precision)
- SPE only has fast access to Local Storage memory
  - Reading data from PPE's main memory is very slow
  - Locality of the date for SPE is extremely important
- Only has 8 SPE, no room to waste SPE



Manually optimizing SPE

- Few talented programmers started to manually optimize programs to improve SPE utilization, keeping SPE utilization above 80%
  - In programming, separate calculation task for SPE and PPE
  - Using conventional Multi-threading programing on PPE
  - Making dedicated SIMD friendly calculation function code for SPE
  - (1) Push the calculation function code to SPE
  - (2) Push required calculation data with using DMA for calculation in SPE
  - (3) Read the data with using DMA when calculation is finished in SPE
  - (4) If calculating the same algorithms then goto (2)
  - (5) If prefer calculating the different algorithms then goto (1)
  - Be careful with alignment, other than bellow had significant performance degradation, 32bits (4bytes) for program code, 128bits (16 bytes) for data
  - Use DMA effectively, initiate early, do something else during DMA, predict when DMA is finished and start calculation immediately
  - All must be done manually because of locality requirement of the data and code for SPE

Any of mistakes in the list, lead dramatical performance degradation

this is not for all programmers

Designing library for conventional programmers

- There was CUDA, GPGPU core supported only fixed algorithms until around 2008
  - Focusing data parallelism was fine for achieving high performance
- On Cell/B.E., optimizing parallelism on both calculation code and data was the best to achieve the performance
  - Necessity of providing programing environment which is programmer friendly way of programing (1) to (5) in the previous page
  - The key is not to waste SPE time with SPE waiting for calculation code or data
- Main optimization aspects would like to hide from programmers for designing library
  - DMA handling, detecting SPE is requesting data, detecting SPE have finished data, manually scheduling the calculation function code between PPE and SPE

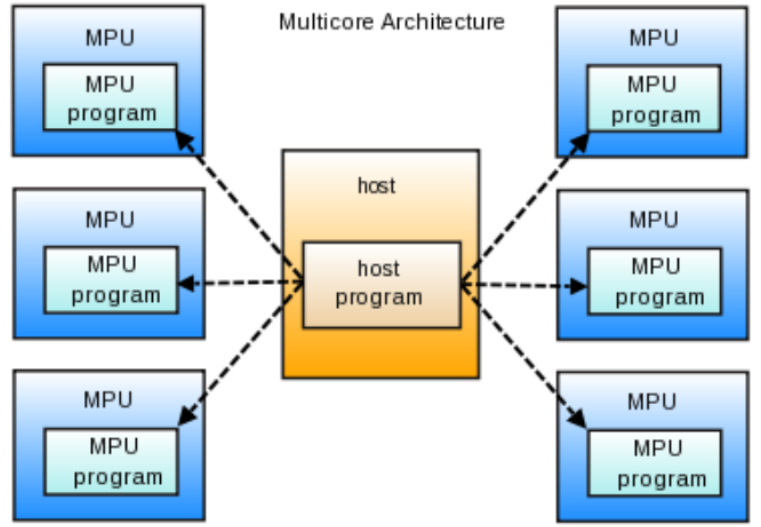
MARS - Multi-core Application Runtime System

Main Page | Modules | Data Structures | Files

MARS - Multicore Application Runtime System

Additional Resources

- Future releases and other information for MARS:
  - ftp://ftp.infradead.org/pub/Sony-PS3/mars/
- Source repository for MARS:
  - http://git.infradead.org/ps3/mars-src.git
  - git://git.infradead.org/ps3/mars-src.git



1.1 Host Processor (host)

The host processor (host) is the processor on which the host program will be run.

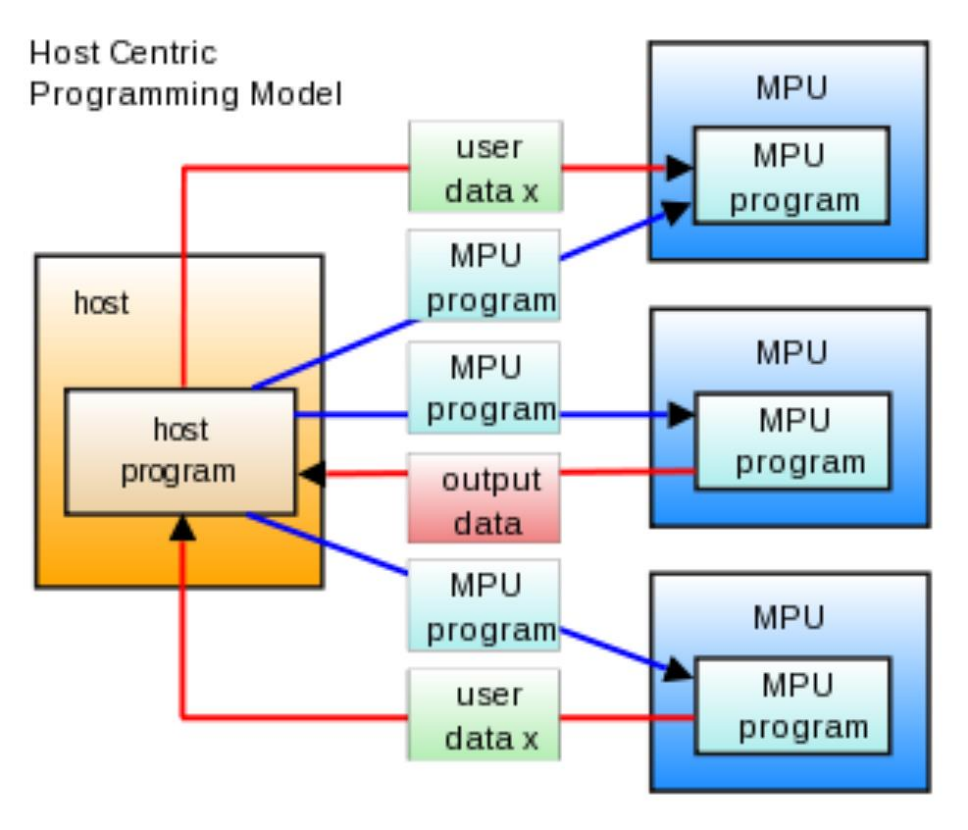
Originally hosted by Fedora project
ftp://ftp.infradead.org/pub/Sony-PS3/mars/latest/mars-docs-1.1.5/html/index.html
Backup copy by Akira Tsukamoto
https://www.akiratec.com/archive/Sony-PS3/mars/1.1.5/mars-docs-1.1.5/html/

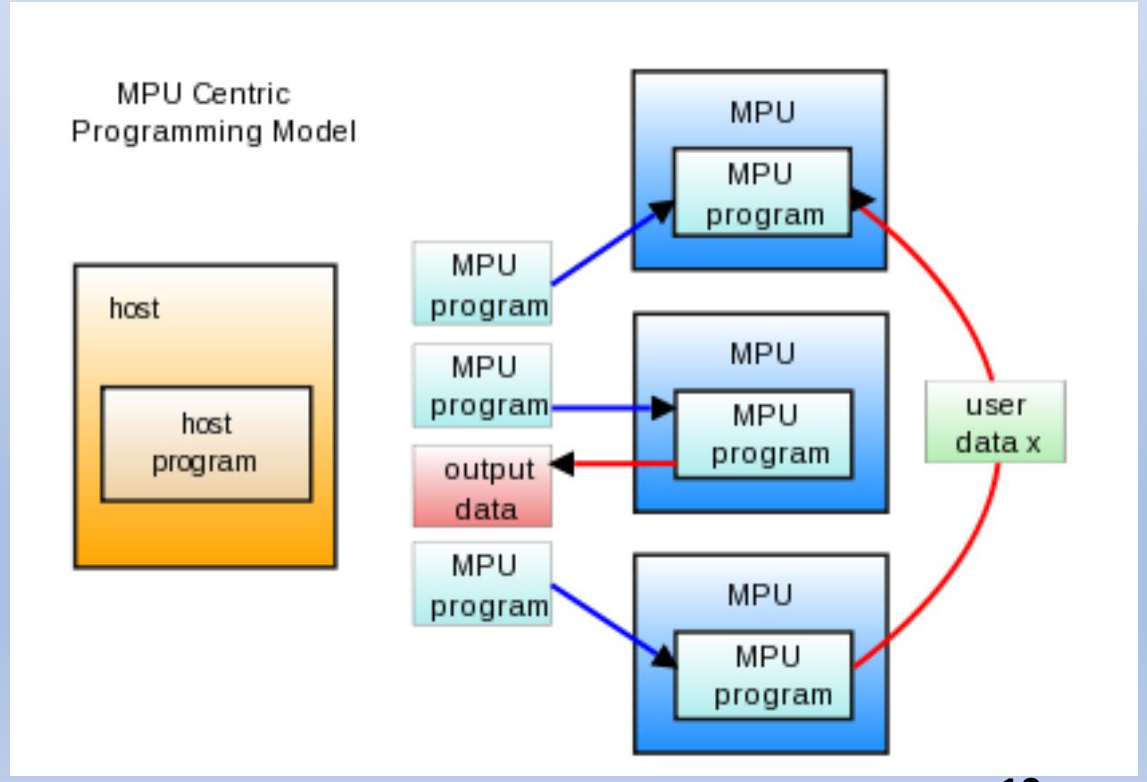
MARS programing style (1/6)

- Some naming conventions in the MARS documentation
  - Host Processor (host) -> PPE
  - Host Storage -> main memory connected on PPE
  - Microprocessing Unit (MPU) -> SPE
  - MPU storage -> SPU local storage
  - Workload/Task -> calculation code and/or data

MARS programing style (2/6)

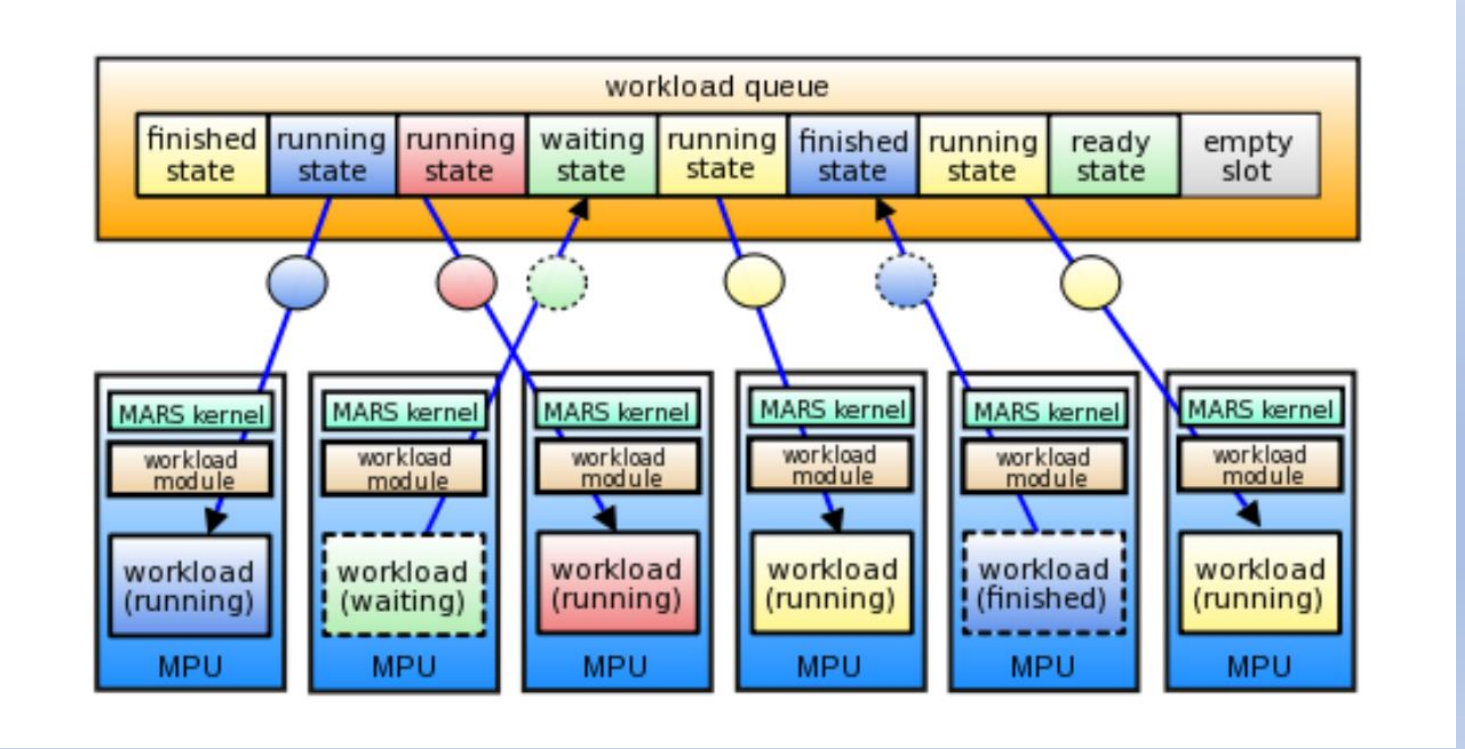
- To prevent from wasting SPE time by waiting for Workload (code and/or data)
- SPE will fetch workload from SPE, instead of PPE pushing them to SPE





MARS programing style (3/6)

- Many workloads are inserted in the queue at PPE
- Every time when the SPE detects stalls (waiting or finished), SPE will replace the stalled Workloads with other Workloads on demand



MARS programing style (4/6)

- Many synchronization APIs
  - mars_task_semaphore_*()
  - mars_task_barrier_*()
  - mars_task_event_flag_*()
  - mars_task_queue_*()
  - mars_task_signal_send/wait/try_wait()

MARS programing style (5/6)

- Hiding manual DMA data transfer handling between PPE main memory to SPE local storage

```
1   #include <mars/task.h>
2
3   int mars_task_main(const struct mars_task_args *task_args)
4   {
5           uint64_t semaphore_ea = task_args->type.u64[0];
6           uint64_t shared_resource_ea = task_args->type.u64[1];
7           uint32_t shared_resource __attribute__((aligned(16)));
8
9           mars_task_semaphore_acquire(semaphore_ea);
10
11          get(&shared_resource, shared_resource_ea, sizeof(uint32_t));
12
13          shared_resource++;
14
15          put(&shared_resource, shared_resource_ea, sizeof(uint32_t));
16
17          mars_task_semaphore_release(semaphore_ea);
18
19          return 0;
20  }
```

DMA: from PPE main to SPE Local Storage

Calculation code in between
Best to use SIMD capability here
See later page for details

DMA: from SPE local storage to PPE main

MARS programing style (6/6)

- MARS APIs relationship with OpenCL

MARS

```
1   #include <mars/task.h>
2
3   int mars_task_main(const struct mars_task_args *task_args)
4   {
5           uint64_t semaphore_ea = task_args->type.u64[0];
6           uint64_t shared_resource_ea = task_args->type.u64[1];
7           uint32_t shared_resource __attribute__((aligned(16)));
8
9           mars_task_semaphore_acquire(semaphore_ea);
10
11          get(&shared_resource, shared_resource_ea, sizeof(uint32_t));
12
13          shared_resource++;
14
15          put(&shared_resource, shared_resource_ea, sizeof(uint32_t));
16
17          mars_task_semaphore_release(semaphore_ea);
18
19          return 0;
20  }
```

From main to local storage

From local storage to main

The **OpenCL** Specification

Version: 1.0

Document Revision: 43

Acknowledgements

John Bates, Sony

OpenCL

```
1
2   __kernel square(
3       __global float *g_input,
4       __global float *g_output,
5       __local float *local,
6       const unsigned int count)
7   {
        int gid = get_global_id(0);
        int lid = get_local_id(0);
10
11      local[lid] = g_input[gid];
12      barrier(CLK_LOCAL_MEM_FENCE);
13
14      local[lid]++;
15
16      g_output[gid] = local[lid];
```

The Cell/B.E. played a pioneering role on modern AI/GPGPU programming

Effective SIMD optimizing coding

- Instead of using “shared _ resource ++” or “local[lid]++” in previous pages

vector type extension on spu-gcc

Vector Type	Data
_vector unsigned char	Sixteen unsigned 8-bit data
_vector signed char	Sixteen signed 8-bit data
_vector unsigned short	Eight unsigned 16-bit data
_vector signed short	Eight signed 16-bit data
_vector unsigned int	Four unsigned 32-bit data
_vector signed int	Four signed 32-bit data
_vector unsigned long long	Two unsigned 64-bit data
_vector signed long long	Two signed 64-bit data
_vector float	Four single-precision floating-point data
_vector double	Two double-precision floating-point data

SIMD programming

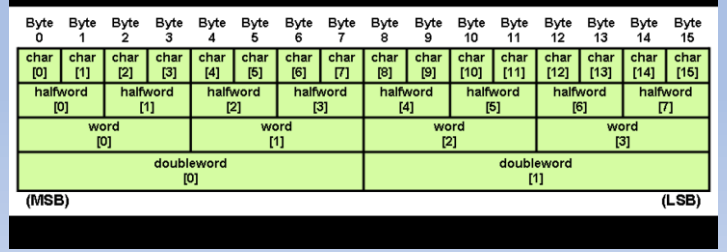
```
float a[4], b[4], c[4];

for (i = 0; i < 4; i++) {
   c[i] = a[i] * b[i];
}
```

```
__vector float va, vb, vc;
vc = spu_mul(va, vb);
```

Other SIMD Built-in Functions

Applicable Instructions	VMX	SPU SIMD	Description
Arithmetic Instructions	vec_add(a,b)	spu_add(a,b)	Adds the elements of vectors *a* and *b*.
	vec_sub(a,b)	spu _sub(a,b)	Performs subtractions between the elements of vectors *a* and *b*.
	vec_madd(a,b,c)	spu_madd(a,b,c)	Multiplies the elements of vector *a* by the elements of vector *b* and adds the elements of vector *c*.
	vec_re(a,b)	spu_re(a,b)	Calculates the reciprocals of the elements of vector a.
	vec_rsqrte(a)	spu_rsqrte(a)	Calculates the square roots of the reciprocals of the elements of vector *a*
Logical Instructions	vec_and(a,b)	spu_and(a,b)	Finds the bitwise logical products (AND) between vectors *a* and *b*.
	vec_or(a,b)	spu_or(a,b)	Finds the bitwise logical sums (OR) between vectors *a* and *b*.

vector type extension on spu-gcc



See details in the slide

Originally hosted by Fedora project
ftp://ftp.infradead.org/pub/mars/presentations/CellBE-best-programming-20091211.pdf

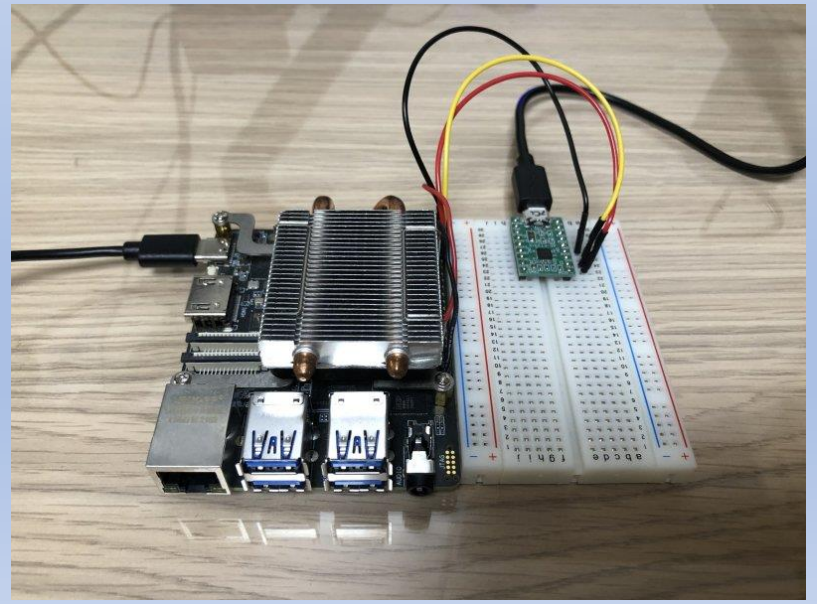
Backup copy by Akira Tsukamoto
https://www.akiratec.com/archive/Sony-PS3/mars/presentations/CellBE-best-programming-20091211.pdf

Topics
Alignments tweaks on RISC-V for Network speeds

- What was observed on RISC-V RV64 boards
- Two type of copy functions inside kernel
- When the copy functions are used with large size and what is related with network performance
- Why the performance improves compared to historical copy function
- Relation with “Computer Architecture: A Quantitative Approach” and the optimizing copy functions
- Which items of CPU design impact performance of software
- What’s next?

What was observed on RISC-V RV64 boards

- CPU usage was extremely high at 34.69% (memcpy), 33.84% (copy_to_user) for a total of 68.53%, which caused slow network speeds.
  - Starlight Dev Board
    - SoC: StarFive JH7100
    - Core: SiFive U74 (Dual core)



- Workload of Network benchmark
  - iperf3 -u -b 1000M --length 6500 -c 192.168.1.112
- On starlight, similar on Unmatched
  - perf top -Ue task-clock

```
Samples: 35K of event 'task-clock', 4000 Hz, Event count (
Overhead  Shared O  Symbol
  34.69%  [kernel]  [k] memcpy
  33.84%  [kernel]  [k] __asm_copy_to_user
   2.28%  [kernel]  [k] sifive_l2_flush64_range
   1.58%  [kernel]  [k] dev_gro_receive
   1.41%  [kernel]  [k] skb_gro_receive
   1.28%  [kernel]  [k] memset
   0.97%  [kernel]  [k] _raw_spin_unlock_irqrestore
   0.86%  [kernel]  [k] page_pool_put_page
   0.75%  [kernel]  [k] finish_task_switch.isra.0
   0.66%  [kernel]  [k] __skb_datagram_iter
   0.63%  [kernel]  [k] inet_gro_receive
   0.60%  [kernel]  [k] enh_desc_get_rx_status
```

Where to optimize? (1/2)

- aligned access and unaligned access, aligned address for RV64
  - U74 (RV64) has load and store instructions on main memory access for 64bit/8byte boundary only aligned 64bit memory access and unaligned 8bit, 16bit, 32bit boundary access examples

```
0xE200:0000      64 bit / 8 byte aligned address for U74, able to divide by 8
0xE200:0001       8 bit / 1 byte boundary
0xE200:0002      16 bit / 2 byte boundary
0xE200:0003       8 bit / 1 byte boundary
0xE200:0004      32 bit / 4 byte boundary
0xE200:0005       8 bit / 1 byte boundary
0xE200:0006      16 bit / 2 byte boundary
0xE200:0007       8 bit / 1 byte boundary
0xE200:0008      64 bit / 8 byte aligned address for U74, able to divide by 8
```

  - The 8bit, 16bit, 32bit boundary access are not implemented in U74 and every 8bit, 16bit, 32bit boundary access were trapped as illegal instruction, and OpeSBI in M-mode is handling the 8bit, 16bit, 32bit boundary access operation

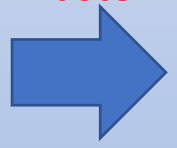
  - Overhead for each unaligned memory access:
    - illegal instruction trap + switching S-mode (kernel) and M-mode (OpenSBI)

Where to optimize? (2/2)

- Typical copy functions inside kernel are used with small copying size
  - Less than 64 bytes
- The size becomes large for network packets.
  - MTU (Maximum Transmission Unit) is 1500 bytes, most of the network application are optimized to use maximum MTU size to reduce number of calling socket APIs instead of calling every bytes

```
if (I = 0; I = 1500; i++) {
    send(sock, *buf, 1, 0); /* one byte */
}
```

faster



```
send(sock, *buf, 1500, 0); /* 1500 byte */
```

Reduce switching between u-mode (app) and s-mode (kernel)

  - The MTU size is getting larger, many started to use jumbo MTU 9000 bytes
  - Resulting to copying in large size

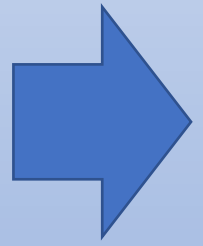
    memcpy(*dst, *src, 1500) or memcpy(*dst, *src, 9000)

    copy_to_user(*u_dst, k_src, 1500) or copy_to_user(*u_dst, k_src, 9000)

Optimizing on in-order CPU core

- Only software could avoid data hazard on in-order core
  - U74: In-order, 5-6stage, 4 cycle load use
- Known to improve performance with loop unrolling to reduce read after write data hazard (RAW)

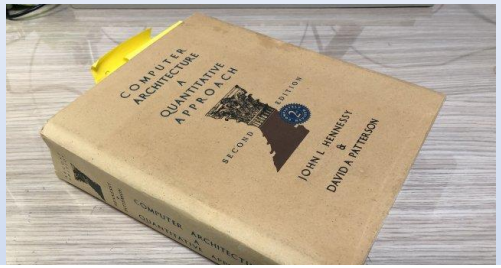
```
for (i = 0; i < size; i++) {
    *dst = *src;
    dst++, src++;
}
```

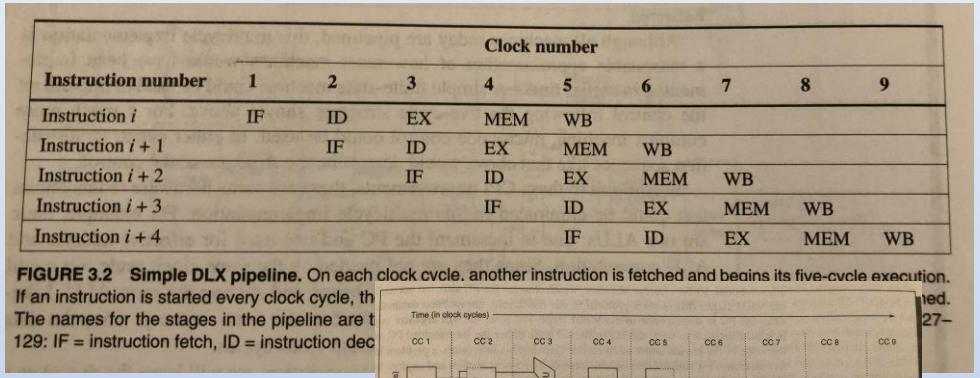


```
for (i = 0; i < size; i + 4) {
        *dst      = *src;
        *dst + 1 = *src + 1 ;
        *dst + 2 = *src + 2 ;
        *dst + 3 = *src + 3 ;
        dst = dst + 4, src = src + 4;
}
```

But why?

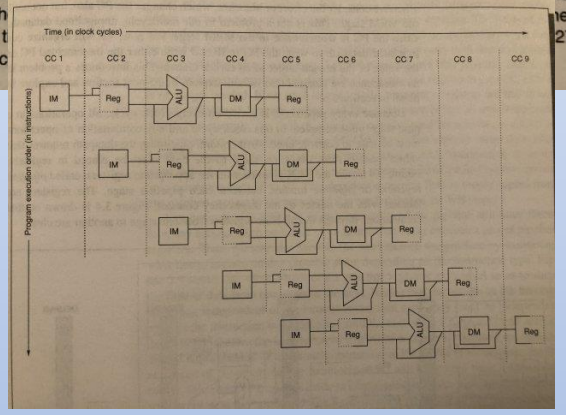
“Computer Architecture: A Quantitative Approach”

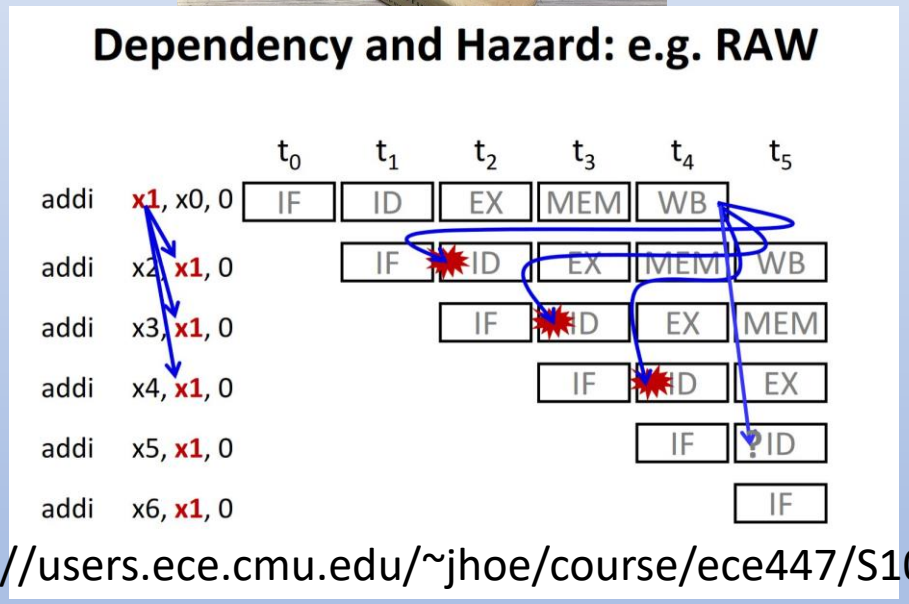




Instruction number	Clock number 1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

FIGURE 3.2 Simple DLX pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, th ...ed. The names for the stages in the pipeline are t ...27–129: IF = instruction fetch, ID = instruction dec





https://users.ece.cmu.edu/~jhoe/course/ece447/S10handouts/L08.pdf

Describes condition of read after write data hazard (RAW)

Able to predict U74 would likely to have 3 pipeline stall for RAW from 4 cycle load use

Good to study on good book!

Two type of copy functions in kernel

- Both types had impact on the performance

1) Used for source codes inside kernel, similar usage as in-kernel library
   memcpy(), mommove(), memset()
   The page fault is handled in kernel, able to write completely in C only
   arch/riscv/lib/string.c
   Matteo Croce have sent the optimization patches to mailing
   https://www.spinics.net/lists/kernel/msg4094278.html

2) Used for system-calls for copying data between user space and kernel space
   copy_to_user(), copy_from_user()
   Require adding page fault handler manually, require to write at least partially in assembler
   I sent the optimization patches
   https://lkml.org/lkml/2021/6/19/131
   https://lkml.org/lkml/2021/7/20/180
   Merged in v5.14-rc3

The code, putting all together

The location of the source of copy_to_user(), copy_from_user().
https://elixir.bootlin.com/linux/latest/source/arch/riscv/lib/uaccess.S

```
        /*
         * Register allocation for code below:
         * a0 - start of uncopied dst
         * a1 - start of uncopied src
         * a2 - size
         * t0 - end of uncopied dst
         */
        add     t0, a0, a2

        /*
         * Use byte copy only if too small.
         * SZREG holds 4 for RV32 and 8 for RV64
         */
        li      a3, 9*SZR
        bltu    a2, a3, .

        /*
         * Copy first byt
         * a0 - start of
         * t1 - start of
         */
        addi    t1, a0, S
        andi    t1, t1, ~
        /* dst is already
        beq     a0, t1, .
1:
        /* a5 - one byte for copying data */
        fixup lb      a5, 0(a1), 10f
        addi    a1, a1, 1       /* src */
        fixup sb      a5, 0(a0), 10f
        addi    a0, a0, 1       /* dst */
        bltu    a0, t1, 1b      /* t1 - start of aligned dst */

.Lskip_align_dst:
        /*
         * Now dst is aligned.
         * Use shift-copy if src is misaligned.
         * Use word-copy if both src and dst are aligned because
         * can not use shift-copy which do not require shifting
         */
        /* a1 - start of src */
        andi    a3, a1, SZREG-1
        bnez    a3, .Lshift_copy
```

Checking src and dst are 8 byte aligned address or not, if not, copy until dst is aligned

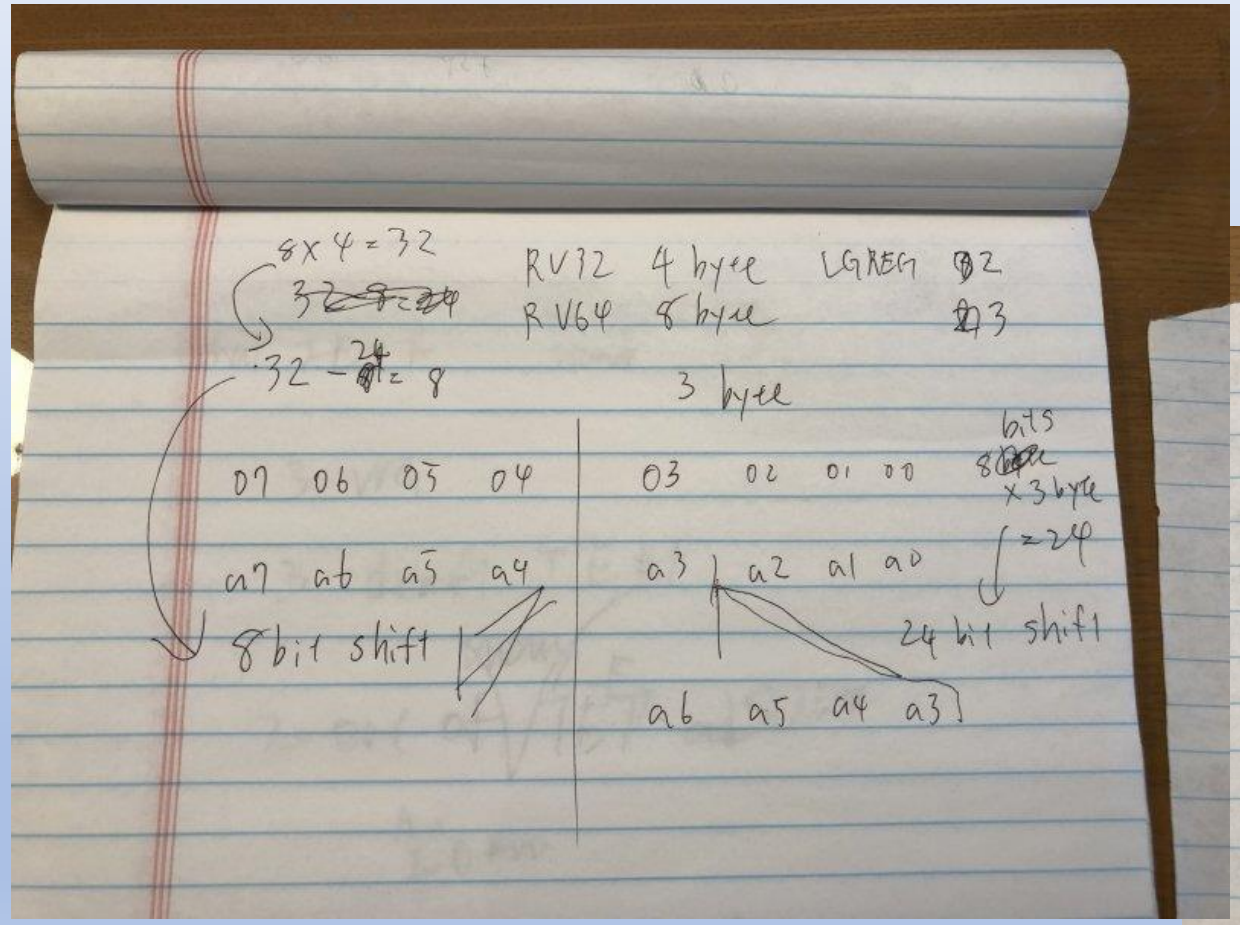
```
.Lshift_copy:

        /*
         * Word copy with shifting.
         * For misaligned copy we still perform aligned word copy, but
         * we need to use the value fetched from the previous iteration and
         * do some shifts.
         * This is safe because reading is less than a word size.
         *
         * a0 - start of aligned dst
         * a1 - start of src
         * a3 - a1 & mask:(SZREG-1)
         * t0 - end of uncopied dst
         * t1 - end of aligne
         */
        /* calculating aligne
        andi    t1, t0, ~(SZR
        /* Converting unalign
        andi    a1, a1, ~(SZR

        /*
         * Calculate shifts
         * t3 - prev shift
         * t4 - current shift
         */
        slli    t3, a3, 3 /*
        li      a5, SZREG*8
        sub     t4, a5, t3

        /* Load the first wor
        fixup REG_L   a5, 0(a
3:
        /* Main shifting copy
         *
         * a0 - start of alig
         * a1 - start of alig
         * t1 - end of aligne
         */

        /* At least one itera
        srl     a4, a5, t3
        fixup REG_L   a5, SZR
        addi    a1, a1, SZREG
        sll     a2, a5, t4
        or      a2, a2, a4
        fixup REG_S   a2, 0(a
        addi    a0, a0, SZREG
        bltu    a0, t1, 3b

        /* Revert src to original unaligned value  */
        add     a1, a1, a3
```

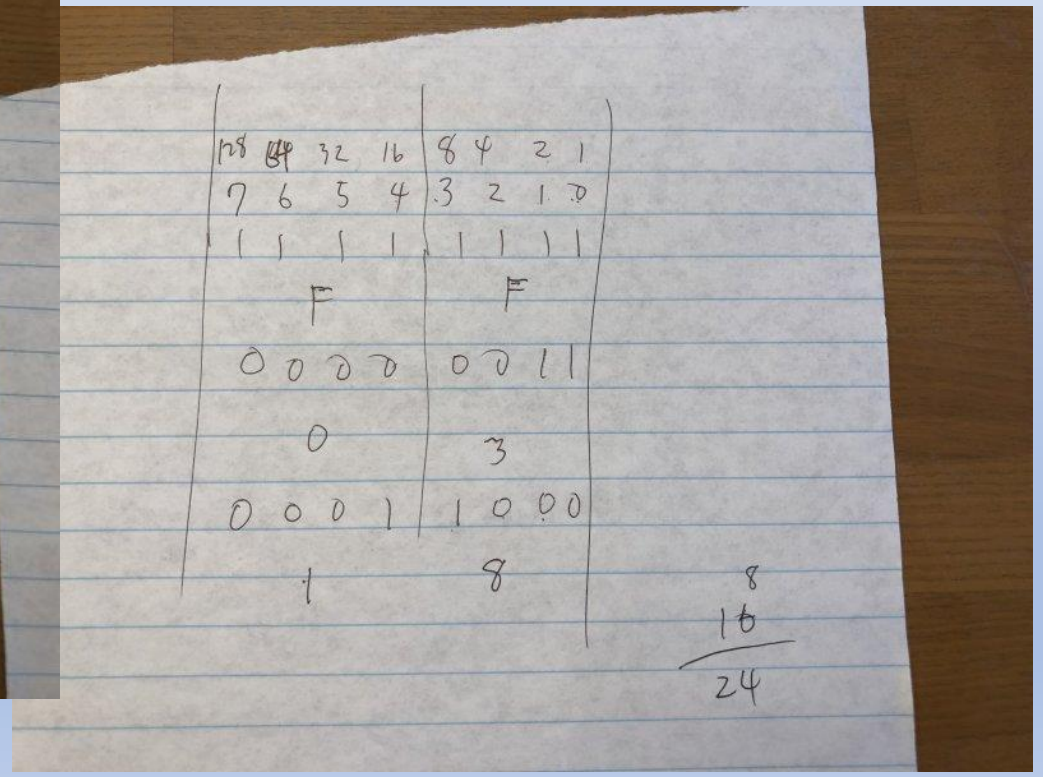
The src is not aligned, read src every 8 byte in aligned address but shifting data in registry to compensate of reading unaligned data on closest aligned address

```
.Lword_copy:
        /*
         * Both src and dst are aligned, unrolled word copy
         *
         * a0 - start of aligned dst
         * a1 - start of aligned src
         * t0 - end of aligned dst
         */
        addi    t0, t0, -(8*SZREG) /*
2:
        fixup REG_L   a4,        0(a
        fixup REG_L   a5,    SZREG(a
        fixup REG_L   a6,  2*SZREG(a
        fixup REG_L   a7,  3*SZREG(a
        fixup REG_L   t1,  4*SZREG(a
        fixup REG_L   t2,  5*SZREG(a
        fixup REG_L   t3,  6*SZREG(a
        fixup REG_L   t4,  7*SZREG(a
        fixup REG_S   a4,        0(a
        fixup REG_S   a5,    SZREG(a
        fixup REG_S   a6,  2*SZREG(a0), 10f
        fixup REG_S   a7,  3*SZREG(a0), 10f
        fixup REG_S   t1,  4*SZREG(a0), 10f
        fixup REG_S   t2,  5*SZREG(a0), 10f
        fixup REG_S   t3,  6*SZREG(a0), 10f
        fixup REG_S   t4,  7*SZREG(a0), 10f
        addi    a0, a0, 8*SZREG
        addi    a1, a1, 8*SZREG
        bltu    a0, t0, 2b

        addi    t0, t0, 8*SZREG /* revert to original value */
        j       .Lbyte_copy_tail
```

Both src and dst are aligned, perform unrolled copy with every 8 byte in aligned address

My notes when implementing shift copy





Bench results after the patches

- CPU Usage change
  only with my copy_to_user patches

```
--- TCP recv ---
  * Before
   40.40%  [kernel]  [k] memcpy
   33.09%  [kernel]  [k] __asm_copy_to_user
  * After
   50.35%  [kernel]  [k] memcpy
   13.76%  [kernel]  [k] __asm_copy_to_user

--- TCP send ---
  * Before
   19.96%  [kernel]  [k] memcpy
    9.84%  [kernel]  [k] __asm_copy_to_user
  * After
   14.27%  [kernel]  [k] memcpy
    7.37%  [kernel]  [k] __asm_copy_to_user

--- UDP send ---
  * Before
   25.18%  [kernel]  [k] memcpy
   22.50%  [kernel]  [k] __asm_copy_to_user
  * After
   28.90%  [kernel]  [k] memcpy
    9.49%  [kernel]  [k] __asm_copy_to_user

--- UDP recv ---
  * Before
   44.45%  [kernel]  [k] memcpy
   31.04%  [kernel]  [k] __asm_copy_to_user
  * After
   55.62%  [kernel]  [k] memcpy
   11.22%  [kernel]  [k] __asm_copy_to_user
```

- Network performance, near 1Gbps
  with both Matteo's and my patches

Before	After
--- TCP recv ---	
686 Mbits/sec	904 Mbits/sec
683 Mbits/sec	898 Mbits/sec
695 Mbits/sec	905 Mbits/sec
--- TCP send ---	
383 Mbits/sec	393 Mbits/sec
384 Mbits/sec	392 Mbits/sec
--- UDP recv ---	
630 Mbits/sec	875 Mbits/sec
730 Mbits/sec	873 Mbits/sec
--- UDP send ---	
307 Mbits/sec	402 Mbits/sec
307 Mbits/sec	402 Mbits/sec

What's next

- Convert assembler to C
  - copy_to_user(), copy_from_user() is fully written in assembler, able to convert in C partially
- Porting them to runtime selection of choosing the best optimized functions for different CPU core designs
  - Typically, embedded engineers tend to rebuild binary optimized for target CPU, and do not care of other CPU
  - Linux kernel prefer one single kernel binary on all CPU core designs on one architecture. Then distro's do not have to make different kernels for each CPU design. Moving to applying on runtime of the best optimized functions for different CPU core designs might be the way in the future
- Require more spare time for this task ☺

Summary

- Achieving high performance requires deep knowledge and experience in CPU architecture level
  - Register access stall, latency in memory hierarchy, memory alignment restriction, bus architecture, etc.
- Providing generalize software stack will expand adaptation in industry
- How to optimize copy performance on RISC-V
- Patches are merged in upstream kernel version 5.14

Appendix: iperf3, procedure in the slide

--- TCP recv ---

** on PC side, using default mtu 1500

$ iperf3 -c 192.168.1.112

** on riscv side, using default mtu 1500

[root@fedora-starfive ~]# iperf3 -s

--- UDP recv ---

** on PC side first, changing mtu size to 9000

$ sudo ifconfig eth0 down

$ sudo ifconfig eth0 mtu 9000 up

$ iperf3 -u -b 1000M --length 6500 -c 192.168.1.112

** on riscv beaglev side, changing mtu size to 9000 too

[root@fedora-starfive ~]# sudo ifconfig eth0 down

[root@fedora-starfive ~]# sudo ifconfig eth0 mtu 9000 up

[root@fedora-starfive ~]# iperf3 –s

--- TCP send ---

** on PC side, using default mtu 1500

$ iperf3 -s

** on riscv side, using default mtu 1500

[root@fedora-starfive ~]# iperf3 -c 192.168.1.153

--- UDP send ---

** on PC side first, changing mtu size from 1500 to 9000

$ sudo ifconfig eth0 down

$ sudo ifconfig eth0 mtu 9000 up

$ iperf3 -s

** on riscv, No changing the mtu size on riscv beaglev

[root@fedora-starfive ~]# iperf3 -u -b 1000M --length 50000 -c 192.168.1.153