Cell broadband engine, SPE assisted user space device driver

2007.02.22

Hiroyuki Machida

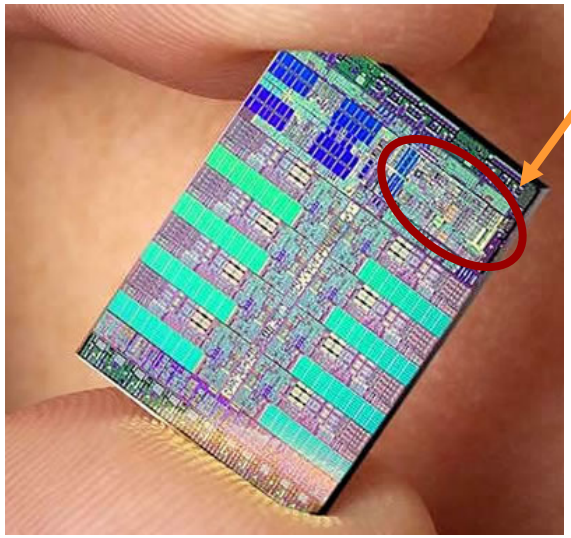
Abstract

* Cell broadband engine is a heterogeneous multi-core processor which consists of a Power PC element, (PPE) and Synergistic Processor Elements (SPEs). SPEs can be used to achieve better performance. This paper proposes tilizing SPEs from user space to accelerate kernel services. Our solution allows kernel services to access to SPEs easily. We'll show evaluation of the concept, using modified compressed loop device driver, CLOOP, to utilize SPE. We'll also discuss possible other kernel services to be accelerated by SPEs.

* Shinohara, Machida
* Tsukamoto,
* Suzaki

Motivation

* Accelerate kernel services including device drivers utilizing Cell features like SPEs.



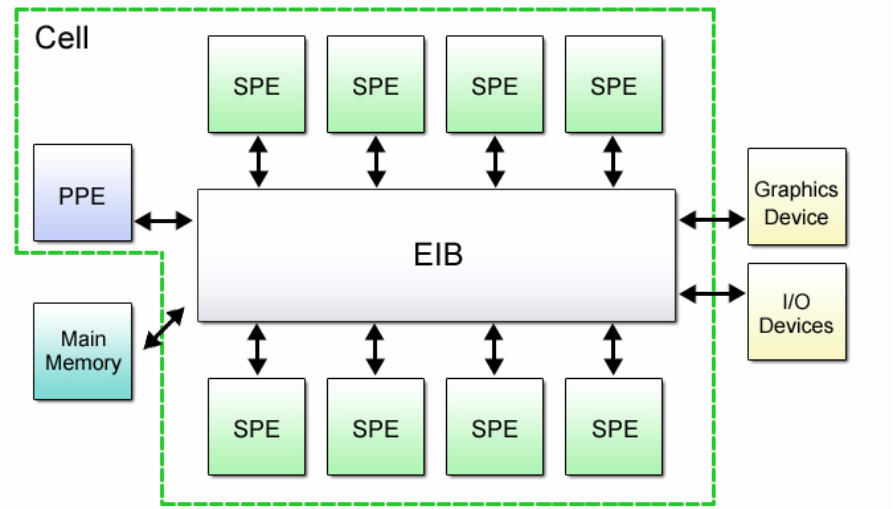
Cell employs following strategies
- “simpler structure and higher clock”
-“more room for SPEs on silicon”

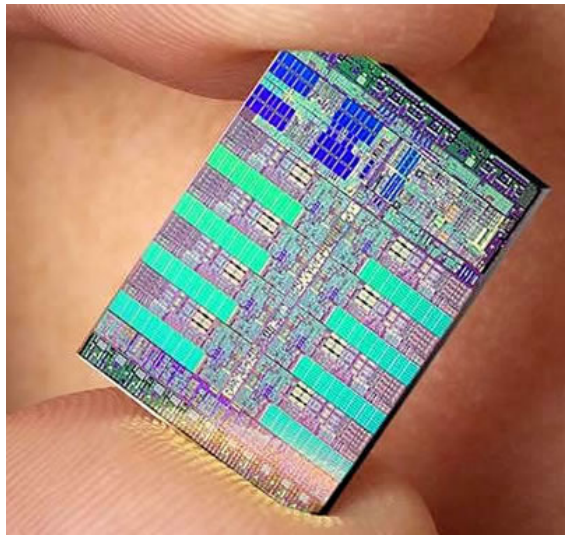
PPU could not achieve same performance with same clock G5(PPC970) due to lack of logics..

Most conventional codes are running on PPU.
It would be very nice to accelerate those conventional code without modification.

What Is Cell ?

* 1 PPE + 8 SPE connected with EIB
  - PPE - Power Processor Element
    * PPC64+VMX insns/SMT/in-order/deep pipeline
  - SPE - Synergetic Processor Element
    * general purpose SIMD with Local Storage (LS), not cache
  - EIB - Element Interconnect Bus
    * Hi-speed coherent SMP Bus





Requirements

* Preserve existing Kernel-User APIs
* Changes should be minimized
* Utilize existing functions as much as possible
* Improve performance
  - Less CPU(PPU) usage
  - Faster in execution

Constraints

* SPE itself doesn’t have privilege mode on execution.
  - CPU core of SPE doesn’t have “privilege” concept.
  - However, MFC has capability to switch kernel mode and user mode address space of PPU side main memory。

* Current Kernel doesn’t support executing heterogeneous CPU core instructions.

Possible solutions

* Kernel Mode
  - Adding new infrastructures in kernel to support SPE acceleration in kernel mode.

* User Mode
  - Adding helper feature inside kernel to allow off loading function for user space for SPE.

Pros v.s. Cons

* Kernel Mode
  - Pros
    * Less overhead
  - Cons
    * It’s difficult to debug with this model
      - kgdb do not speak SPEs
    * New code required for spe control in kernel
* User Mode
  - Pros
    * No new code required for spe control in kernel
    * This allows programmers to use existing tools for debug
  - Cons
    * Overhead switching between kernel and user space
    * Require protecting user space SPE data/prog from other regular user space application

It won't be security hole

* MFC access to the main memory (XDR) is bound inside corresponding user process, due to MFC talking with MMU on PPU.
  - SPU can’t reach XDR directly, just to Local Storage (LS) .
  - Data transfer between LS and XDR is taken place by MFC.

* PPU access to SPU registers and LS are virtualized in each user process unit.
  - PPU can map LS and SPU registers, however SPU registers have privillage and those mappings are under controlled by kernel.
  - Kernel prevents to map LS and SPU registers from other user process.
  - If it could, it’s a bug of kernel.

* As a conclusion, SPU on running is well isolated from the other user processes.

Feasibility Study on User mode

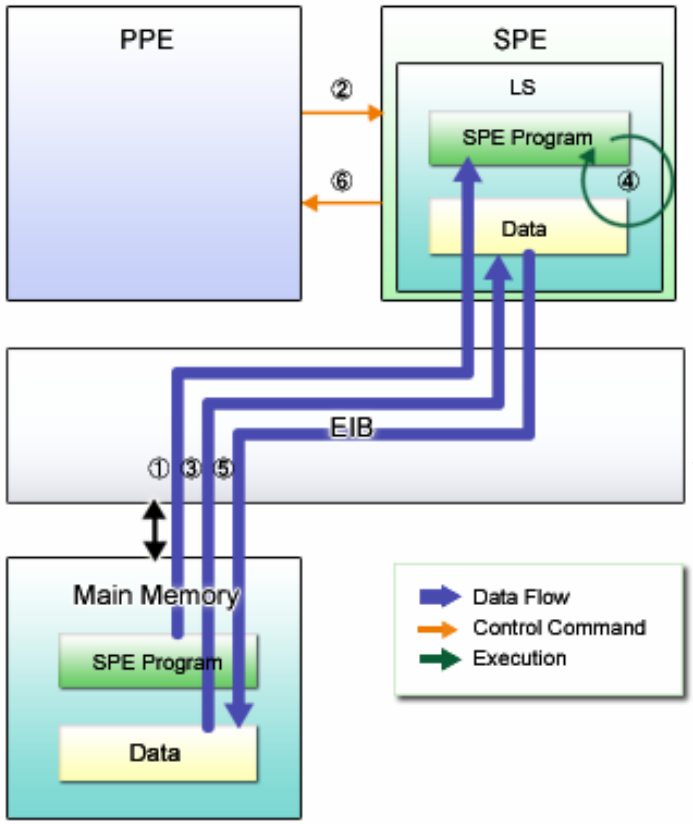
* Try out a simple example using cloop, software device driver.

* Why cloop ?
  - It’s small
  - It has locality on memory reference (block decompression)

Terminologies

* CLOOP Compressed Loop block device
* CLD CLOOP Driver
* ULD User Level Device Driver

Typical SPE program flow



* PPE Program side
  - spe_create_context()
    * Create SPE context
  - spe_image_open()
    * Open elf file of SPE program
  - spe_program_load()
    * [1] Load SPE program to LS
  - spe_context_run()
    * [2] Let SPE start the program

* SEP Program side.
  - [3] Transfer data to be process into LS from PPE side main memory, though by MFC.
  - [4] Processi data in LS
  - [5] Transfer processed data from LS to PPE side main memory though MFC.
  - [6] Signal PPE program that SPE program has sopped.

Basic Design

* User space SPU offloading daemon waits the request from kernel.

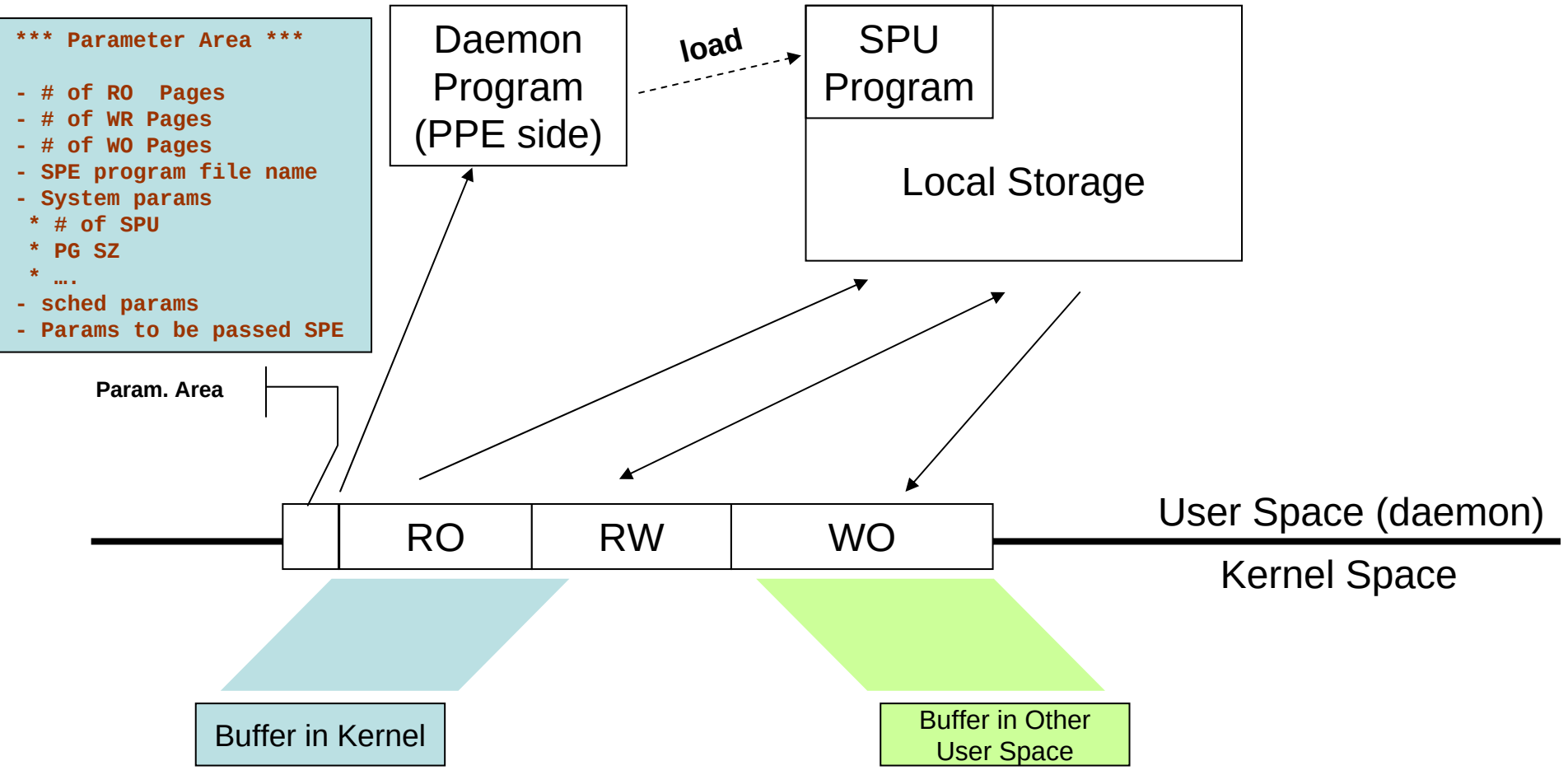
* Driver locks pages, wake up the daemon and pass the pages to the daemon.

* The passed page includes file name of SPE codes, parameters and input data/out put data.

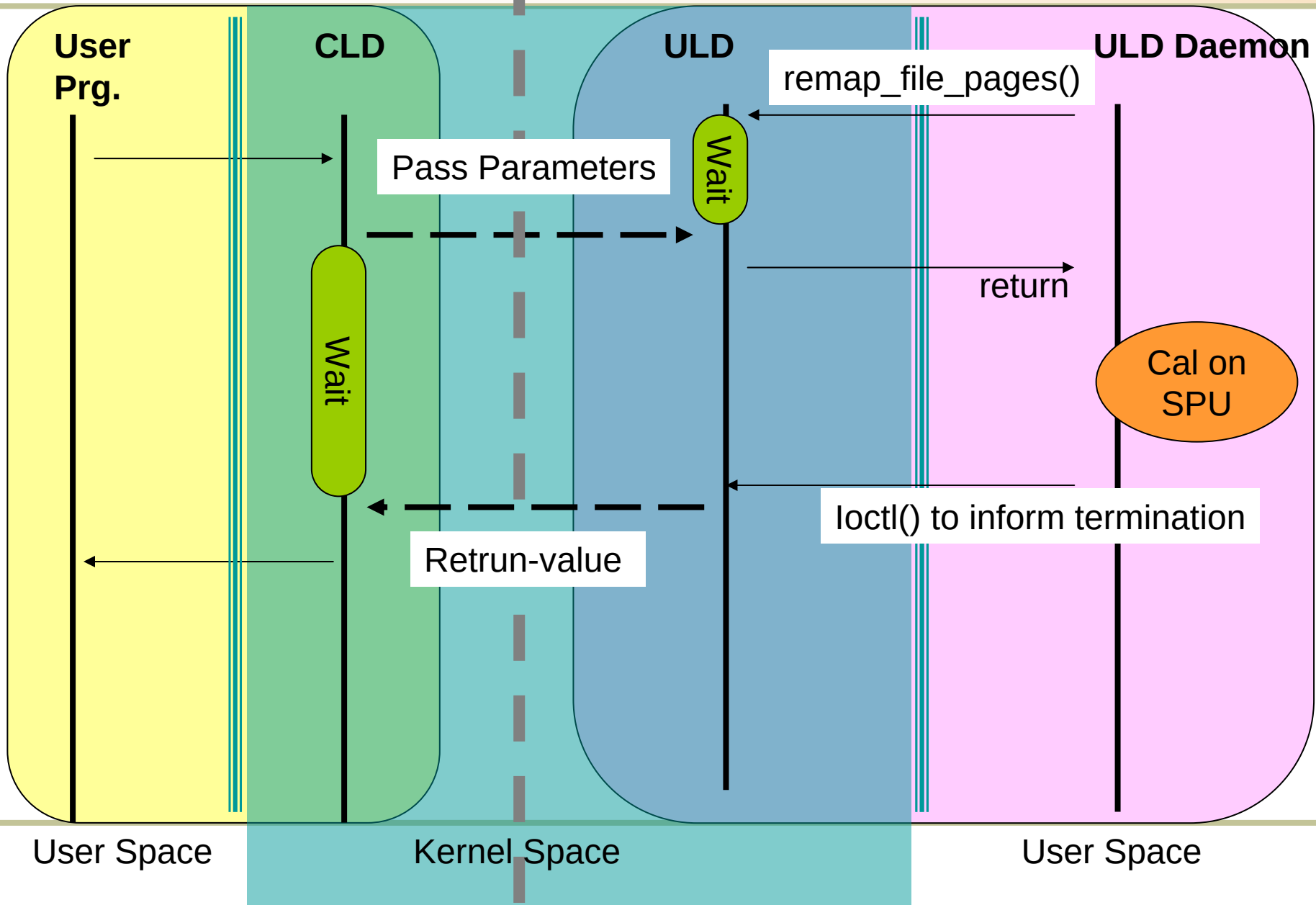
* The daemon start SPE and has been blocked until SPE execution, according the parameters passed from kernel.
  - SPE would transfer data to/from the locked pages.

* After SPE execution finished, daemon will inform it to kernel.

How to pass data from/to kernel



Sequence Diagram



ULD Daemon

* spe_create_context()
* // /dev/uldN … stands for ULD helper device file
* uldfd = open (“/dev/uldN”,O_RDWR)
* addr = mmap(udlfd) // this area are used to pass data between SPE and Kernel.

* repeat:
  - // wait request in remap_file_pages() of ULD helper device.
  - // page map of mapped “/dev/uldN” would be change by ULD helper device, on return.
  - remap_file_pages()
  - Parse parameters on mapped area.
    * If SPE program changed,
      - Close previous SPE program image by spe_image_close()
      - Then, setup new SPE program with spe_image_open & spe_program_load
  - spe_context_run() // run SPE and wait until stop
  - // notify termination of SPE program to ULD helper device
  - Goto repeat;

CLD側から見たシナリオ

* Cloopを改造 – decompress() を ULD 側へ
  - ULD側が用意したパラメータエリアのプールからN個確保
  - パラメータを埋める x N
    * 自分の使いたいSPU program名前
    * データページのリスト(RO,RW,WO)
    * など、
  - N個パラメータセットをULDに投げ、結果をまつ
  - パラメータエリア開放

Performance estimations

* Purpose
  - Does it have worth to develop ?
    * PPU execution time >> SPU execution time + Overhead

* Estimations on executions speed

* Estimations on over head

Execution speed

* Evaluation with miniLZO
  - Candidate of alternative Decompress() in CLOOP
  - http://www.oberhumer.com/opensource/lzo/
* Why?
  - Samll foot print
  - Simple algorithm
* Changes on miniLZO.202
  - Make buffer size to 64KB, so that LS can hold entirely.
  - Suzaki-san's comments:
    * 64KB block looks too small, how about 128KB block
    * gz looks better than LZO, in compression rate
  - データが'0'のみ　→ 0-255の決まったパタン
  - 2000回 compression/decompression を繰り返す
  - 1回だけ答えあわせ

測定

* 条件
  - normal GCC, SPU GCC, (参考：SPU XLC)で測定
  - SPUのバイナリもelfspuで直に実行
  - Time コマンドで real time を比較

* 環境
  - PPC64 FC5
    * gcc-4.1.1-1.fc5
  - PS3 Linux Distributor Starter’s Kit v1.1
    * libspe2-2.0.1-be0647.3.20061130.1.ps3pf
    * spu-newlib-1.14.0.200612070000-1.ps3pf
  - gcc/binutils from IBM Cell SDK 2.0
    * spu-gcc-3.3-72
    * spu-binutils-3.3-72

計測結果

	PPU	SPU	SPU-XLC	
Before Optimization	1.015	0.996	1.603	sec: 2000回
	507.5	498	801.5	usec(compress/decompress)
	253.75	**249**	**400.75**	**usec**

```
% spu-gcc  -Wall -O3 -mbranch-hints -mdd3.0 -funroll-all-loops ¥
  -fomit-frame-pointer -g -ftree-vectorize   -finline-functions ¥
  -ftree-vect-loop-version   -ftree-loop-optimize -o testmini ¥
  testmini.c minilzo.c
% time ./testmini
```

* spu elf なら load & run するプログラムが登録済みなので、うまく動く

高速化の余地の検討

* SPU-GCC/SPU-XLCは、単独だとPPUより遅い or 大差がない。
  - SIMD化されていないためと考えられる。
  - どこを最適化すればよいか?

* gcov での実行回数を観測
  - PPU上で行った。
  - 実行回数が多いのは、以下の場所
    * Byte copy しているところ　　4箇所
    * 0 をbyte searchしているところ　1箇所

* 簡単な最適化実験
  - 32byte 以上なら、Byte copy → memcpy

最適化後結果

	PPU	SPU	SPU-XLC	
Before Optimization	1.015	0.996	1.603	sec: 2000回
	507.5	498	801.5	usec(compress/decompress)
	253.75	**249**	**400.75**	**usec**

After source level Optimization	0.581	1.156	0.874	sec: 2000回
	290.5	578	437	usec(compress/decompress)
	145.25	**289**	**218.5**	**usec**

簡単な最適化実験

* 結果
  - SPU (GCC)では、遅くなった。
  - Memcpy()が遅い?

* 今後
  - SIMD化すれば何とかなりそうな気がしている
    * 「0 をbyte searchしているところ」
    * Investigate into SPE memcpy()
  - Enlarge block size
    * Keeping whole in LS
      - LZO can share in/out buffer 64KB -> 128KB
      - Multiple 64KB blocks (4x64K)
    * Change program living beyond LS
  - Try with various input data
  - Consider other de/compression algorithms

Overhead

* 大まかな設計から
  - 1回のコンテキストスイッチ　5usec
  - 2-3回のシステムコールが必要　1usec x 2-3
  - Remap　128KB (64Kx2 or 4Kx32)　10usec??
* lmbench の結果から推測？
* 20usec程度のoverhead

```
Processor, Processes - times in microseconds - smaller is better
------------------------------------------------------------------------------
Host                 OS  Mhz null null      open slct sig  sig  fork exec sh
                             call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
localhost  Linux 2.6.16 3185 0.26 0.94 8.35 11.8 39.4 1.02 4.58 605. 2333 6804
localhost  Linux 2.6.16 3183 0.26 0.93 8.41 11.9 39.6 1.01 4.59 603. 2340 6824


Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
Host                 OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                         ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ------ ------ ------ ------ ------ ------- -------
localhost  Linux 2.6.16 2.9200   13.5   12.1 9.1400   57.1    12.0    72.6
localhost  Linux 2.6.16 3.3800 4.5100   13.2 8.3000   52.5    16.2    72.1
```

Next Steps - 1

* Using UIO ?
  - What's UIO (Userspace IO devices) ?
    * Documents/DocBook/kernel-api.tmpl
      - drivers/uio/uio.c
      - include/linux/uio_driver.h
    * Documents/DocBook/uio-howto.tmpl
  - Features
    * Kernel side helper driver of userspace IO driver could be easily implement
      - Just few methods and attributes

* Implementation

* Evaluations

Next Steps - 2

* Propose Helper functions
  - In Kernel space
  - In User space

* Other Candidate of User Space SPE Device Drivers
  - Discuss applications on other device drivers and additional requirements on helper functions.

  - Crypto API
    * -> OCF looks very good candidate ?
  - VFB
  - USB web cam ?? It could be user space driver even now.
    * Decompression
    * Color space converter