

ITANIUM TRICKS AND GOTCHAS

Matt Chapman and Gernot Heiser

National ICT Australia
and
University of New South Wales

May 2004



AN EXERCISE IN MICRO-OPTIMISATION

- Aim: Ultra-fast system calls
 - for *any* operating system
- Testbed: L4Ka::Pistachio microkernel (Karlsruhe & UNSW)
 - very fast and small message-passing kernel
 - performance-critical operation is IPC system call
 - switches to receiving thread in same or other address space
 - delivers a message in registers and (optionally) memory
 - designed to be highly optimised

L4 IPC FAST-PATH

LOGICAL OPERATION

1. Enter kernel mode (`epc`)
2. Inspect TCBs of source and destination threads
3. Check fast path conditions hold
4. Copy message (if longer than 8 message registers)
5. Switch register stack and a few other registers
Note: scratch registers presently not cleared
6. Switch address space
7. Update TCB variables and current thread pointers
8. Return

IPC PATH PERFORMANCE

- C++ version (gcc)
 - 508 cycles
- Initial (unoptimised) assembler version
 - 170 cycles
 - 83 instruction groups
 - $170 - 83 = 87$ bubbles!
- First attempt at manual scheduling:
optimised code and increased parallelism
 - 95 cycles
 - 39 instruction groups
 - $95 - 39 = 56$ bubbles!
- Why so many bubbles?

WHAT'S GOING ON???

THE PMU IS YOUR FRIEND!

```
56      BACK_END_BUBBLE.ALL
  30      BE_EXE_BUBBLE.ALL
    16      BE_EXE_BUBBLE.GRALL
    14      BE_EXE_BUBBLE.ARCR
  15      BE_L1D_FPU_BUBBLE.ALL
    10      BE_L1D_FPU_BUBBLE.L1D_DCURECIR
     5      BE_L1D_FPU_BUBBLE.L1D_STBUFRECIR
  11      BE_RSE_BUBBLE.ALL
     4      BE_RSE_BUBBLE.AR_DEP
     7      BE_RSE_BUBBLE.LOADRS
```

But what does it all mean?

- info in microarchitecture manual is insufficient for understanding causes!
- need to experiment

PART 1: RSE

Test RSE latencies for all combinations of two instructions:

From	To	cyc	PMU counter
mov ar.rsc=reg	RSE_AR†	12	BE_RSE_BUBBLE.AR_DEP
mov ar.rsc=imm	RSE_AR†	2	BE_RSE_BUBBLE.AR_DEP
mov ar.bspstore=	RSE_AR†	5	BE_RSE_BUBBLE.AR_DEP
mov =ar.bspstore	mov ar.rnat=	8	BE_EXE_BUBBLE.ARCR
mov =ar.bsp	mov ar.rnat=	8	BE_EXE_BUBBLE.ARCR
mov =ar.rnat/ar.unat	mov ar.rnat/ar.unat=	6	BE_EXE_BUBBLE.ARCR
mov ar.rnat/ar.unat=	mov =ar.rnat/ar.unat	6	BE_EXE_BUBBLE.ARCR
mov =ar.unat	FP_OP	6	BE_EXE_BUBBLE.ARCR
mov ar.bspstore=	flushrs	≥ 13	BE_RSE_BUBBLE.OVERFLOW
mov ar.rnat=	flushrs	≥ 2	BE_RSE_BUBBLE.OVERFLOW
ANY	flushrs	≥ 2	BE_RSE_BUBBLE.OVERFLOW
mov ar.rsc=	loadrs	≥ 13	BE_RSE_BUBBLE.LOADRS
mov ar.bspstore=	loadrs	≥ 13	BE_RSE_BUBBLE.LOADRS
mov =ar.bspstore	loadrs	≥ 3	BE_RSE_BUBBLE.LOADRS
loadrs	loadrs	≥ 9	BE_RSE_BUBBLE.LOADRS
ANY	loadrs	≥ 2	BE_RSE_BUBBLE.LOADRS

†RSE_AR = any access to ar.rsc / ar.bspstore / ar.bsp / ar.rnat

PART 1: RSE

- Found a number of undocumented cases
- Resulting table enabled better scheduling
 - eliminated most RSE-related bubbles

56	BACK_END_BUBBLE.ALL
30	BE_EXE_BUBBLE.ALL
16	BE_EXE_BUBBLE.GRALL
14	BE_EXE_BUBBLE.ARCR
15	BE_L1D_FPU_BUBBLE.ALL
10	BE_L1D_FPU_BUBBLE.L1D_DCURECIR
5	BE_L1D_FPU_BUBBLE.L1D_STBUFRECIR
11	BE_RSE_BUBBLE.ALL
4	BE_RSE_BUBBLE.AR_DEP
7	BE_RSE_BUBBLE.LOADRS

- remaining two (`loadrs` and `flushrs`) are unavoidable
- overall saved 23 cycles

PART 2: SYSTEM INSTRUCTION LATENCIES

- Measured latencies of system instructions
- Modified Linux kernel:
 - enable userspace to create gate pages using mprotect
 - allows privileged instructions to be tested from a user program
- So far incomplete coverage of cases

PART 2: SYSTEM INSTRUCTION LATENCIES



From	To	cyc	PMU counter
epc	ANY	1	-
bsw	ANY	6	BE_RSE_BUBBLE.BANK_SWITCH
rfi	ANY	13	BE_FLUSH_BUBBLE.BRU (1), BE_FLUSH_BUBBLE.XPN (8), BACK_END_BUBBLE.FE (3)
srlz.d	ANY	1	-
srlz.i	ANY	12	BE_FLUSH_BUBBLE.XPN (8), BACK_END_BUBBLE.FE (3)
sum/rum/mov psr.um=	ANY	5	BE_EXE_BUBBLE.ARCR
sum/rum/mov psr.um=	srlz	10	BE_EXE_BUBBLE.ARCR
ssm/rsm/mov psr.l=	srlz	5	BE_EXE_BUBBLE.ARCR
mov =psr.um/psr	srlz	2	BE_EXE_BUBBLE.ARCR
mov pkr/rr=	srlz/sync/fwb/ mf/invalida_M0	14	BE_EXE_BUBBLE.ARCR
probe/tpa/tak/thash/ttag	USE	5	BE_EXE_BUBBLE.GRALL

PART 2: SYSTEM INSTRUCTION LATENCIES

- Eliminated remaining execution unit bubbles:

56	BACK_END_BUBBLE.ALL
30	BE_EXE_BUBBLE.ALL
16	BE_EXE_BUBBLE.GRALL
14	BE_EXE_BUBBLE.ARCR
15	BE_L1D_FPU_BUBBLE.ALL
10	BE_L1D_FPU_BUBBLE.L1D_DCURECIR
5	BE_L1D_FPU_BUBBLE.L1D_STBUFRECIR
11	BE_RSE_BUBBLE.ALL
4	BE_RSE_BUBBLE.AR_DEP
7	BE_RSE_BUBBLE.LOADRS

- Leaves data load pipeline stalls...

PART 3: L1D PIPELINE STALLS

- Intel[®] Itanium[®] 2 Processor Reference Manual:
 - “L1D_DCURECIR: Back-end was stalled by L1D due to DCU recirculating”
 - “L1D_DCS: Back-end was stalled by L1D due to dcs requiring a stall”
- What the \$#%& is “DCS”?
- Our best guess:
 - DCS = Data Communication Subsystem?
 - most AR/CR accesses are issued to a limited size DCS buffer (7 entry)
 - DCS stalls occur when buffer is full
- Other L1D pipeline issues:
 - avoid scheduling DCS data return to coincide with two L1D data returns
 - limited number of writeback ports?
 - L1D bubble 5 cycles after move to RR/PKR
 - other cases?

OTHER UNDOCUMENTED FEATURES

- Special split issue cases

- after `srlz, sync, mov =ar.unat`
- before `mf`
- between `mov =ar.bsp` and B-unit
- between M-unit and `fwb`
- other cases?

- M-unit dispersal rules

- documentation seems to be incorrect:

- “ $M_A M_L I - M_S M_A I$ gets mapped to ports M2 M0 I0 - M3 M1 I1. If M_S is a `getf` instruction, a split issue will occur.” (p3-6)
- experimental evidence suggests M1 M0 I0 - M2 M3 I1 mapping
- split issue does **not** occur

- generally load subtype seems to be allocated first, followed by others

RESULT

Version	cycles	inst. grps	bubbles
C++	508	231	277
Initial asm	170	83	87
Optimised	95	39	56
Final	36	33	3
Optimal	34	32	2
Ideal [†]	9	9	0

[†] Ideal: unlimited resources, all latencies 1 cycle

- Compare: Linux fast path `gettimeofday`: 97 cycles
- Remaining bubbles:
 - compulsory `flushrs` bubble
 - compulsory `loadrs` bubble
 - region register write bubble
 - could avoid stall but would need to reschedule

OPTIMAL INSTRUCTION SCHEDULING

- Critical path (34 cycles):
 - 2 cycles `flushrs` and `epc`
 - 12 cycles kernel register read (needed for source TCB pointer)
 - 5 cycles load and check source TCB variables
 - 12 cycles `mov ar.bspstore=` to `loadrs` latency
 - 2 cycles `loadrs`
 - 1 cycle `return` (bubbles after this not counted)
- Bottlenecks preventing optimisation past 34 cycles:
 - 12 cycle kernel register read
 - 12 cycle `mov ar.bspstore=` to `loadrs` latency
 - M2 (system) unit availability
- Other issues:
 - unable to prevent branch mispredict on return
return address always comes from Return Stack Buffer?

FOR COMPARISON

Architecture	port/ optimisation	C++		optimised	
		intra AS	inter AS	intra AS	inter AS
Pentium-3	UKa	180	367	113	305
Small Spaces	UKa				213
Pentium-4	UKa	385	983	196	416
Itanium 2	UKa/NICTA	508	508	36	36
cross CPU	UKa	7419	7410	N/A	N/A
MIPS64	NICTA/UNSW	276	276	109	109
cross CPU	NICTA/UNSW	3238	3238	690	690
	NICTA/UNSW	330	518	200 [‡]	200 [‡]
Alpha 21264	NICTA/UNSW	440	642	≈ 70 [†]	≈ 70 [†]
ARM/XScale	NICTA/UNSW	660	660	120–140 [‡]	120–140 [‡]
UltraSPARC	NICTA/UNSW			100 [‡]	100 [‡]

[†] “Version 2” assembler kernel

[‡] Guestimate!

LESSONS LEARNED

- GCC sux
 - ok, we knew that
- Itanium performance critically dependent on instruction scheduling
 - yes, we knew that too
- Decent documentation would go a long way
 - Can't Intel do better to help their developers????

WHY SHOULD YOU CARE?

- Linux kernel compiled with gcc
- Plenty of scope for performance improvement
- Will be visible for applications with significant system time component
 - little relevance for number crunchers
 - potentially high relevance for future Itanium markets:
 - web servers
 - database servers
 - ISPs
- Hand-optimised assembler code is not a viable approach
 - Need decent compilers!
 - ... But that is not an excuse for insufficient documentation!