SPEC2K Investigations and Next Step Discussion



Kugan Vivekanandarajah, 30/10/03



Outline

- Objective and Set-up
- What is not considered
- What we found Areas for further study
 - Register allocation issues
 - Missed Redundancy elimination in address calculations
 - Rematerialisation opportunity
 - Optimizations impacting register allocation
 - Opportunity to further conditionalize instructions
 - Others
- What next



Objective of this study

- Study gcc 32-bit ARM performance for optimisation opportunities
- Study how GCC optimization for 32-bit ARM fares compared to other architectures
 - SPEC2K INT benchmarks are run in Chromebook (a15) and analyzed
 - numbers shown are the % difference in geomean compared to base case being with -O3
 - GCC trunk version 20130922 is used
 - 252.eon (did not compile) and 254.gap (did not run correctly) are excluded in the results



What is not considered

- Tuning for specific Micro-architecture of 32-bit ARM is not considered
- No consideration for specific microarchitecture
 - optimality of scheduling for the pipeline
 - data layout for cache architecture
 - influence of branch predictor performance
 - etc.
- Not interested in absolute numbers and all the results are relative performance (%) compared to the program compiled with -O3



What we found - Areas for further study

- Register allocation Issues
 - tuning register allocation heuristics for arm
- Making optimizations register pressure aware
- Transformations that also reduce register pressure
- Rematerialization
- Further conditionalize instructions



Register allocation

- GCC register allocator performs badly compared to x86 for 32-bit ARM
 - They are generally tuned for x86 like instructions where number of live-ranges are much lesser for same code
- Tuning IRA and Reload (LRA) heuristics
 - In some cases, register allocator performs better with combinations of -fira-loop-pressure -fsched-pressure -fira-region=one -fira-algorithm=priority compared default at -O3
 - These heuristics may need tuning for 32-bit ARM
- New pass to reduce register pressure to help further
 - Scheduling for register pressure
 - Default fsched-pressure helps neon

	-fsched- pressure	-fira-algorithm=priority	-fira-region=one
SPECINT	0.77%	3.22%	-0.66 bnaro.org

Redundancy in address calculations

- Transformation that also reduce register pressure
- Redundancy Elimination in address calculation
 - Redundancy in address calculation such as

```
(struct_array[i + 1]->cost < struct_array[i]->cost)
```

switch (struct_array_2[index].mem) {

```
struct array 2[index].mem = VAL;
                                                add r3, r0, #1
if (strArray[i+1].cost < strArray[i].cost)</pre>
                                                movw r1, #:lower16:strArray
                                                movt r1, #:upper16:strArray
                                                add ip, r1, r0, asl #3
                                                add r2, r1, r3, asl #3
                                                flds s15, [ip]
                                                flds s14, [r2]
                                                fcmpes s14, s15
```



Redundancy in address calculations

Sharing labels for globals

- Labels identify the start address for globals and loading them live range
- Label can be shared for multiple global static variables
- rearranging them will help even when offset does not fit within 4095 bytes (for arm)

```
static struct str globalStrArray2[100];
static struct str globalStrArray[1000];
static int globalInt;

int bar()
{
    globalInt = 22;
    set ();
    globalInt += globalStrArray[2].i;
    globalStrArray[77].i = globalInt - 1;
    globalStrArray2[77].i = globalInt;
    return 0;
}
```

```
movw r4, #:lower16:.LANCHOR0 movt r4, #:upper16:.LANCHOR0 ...
movw r3, #:lower16:.LANCHOR1 ldr r2, [r4] movt r3, #:upper16:.LANCHOR1 ...
.bss .align 2
.LANCHOR0 = . + 0
.LANCHOR1 = . + 8184
.type globalInt, %object .size globalInt, 4
```



Making optimizations register pressure

aware

- GCSE makes live ranges larger that results in increased register pressure
 - For cases like ply +1 and (ply + 1) * 4 gcse keeps the value ply+1 and increases register pressure
 - In this case, -fno-gcse improves performance.
- Loop invariant code motion
 - Especially constant loads with movw/movt are moved out and spilled resulting in memory loads

	-fno-gcse	-fno-move-loop-invariant
SPEC INT	2.33%	1.15%



Rematerialize instead of spilling it to stack

- Immediate loads of integer/float constants
- Loads from literal pools
- Address calculations
 - Computing a constant offset from the stack
 - Pointer Computing a constant offset from the static data area pointer

```
position[ply+1]=position[ply];
```

add ip, r0, #1
str ip, [sp, #228]
movw fp, #:lower16:position
movt fp, #:upper16:position
mov ip, r0, asl #2
str ip, [sp, #284]
ldr r3, [fp, ip] @ unaligned
ldr ip, [sp, #228]
str r3, [fp, ip, asl #2] @ unaligned



Conditionalize Instructions

- Using Conditionalized instructions
 - How much to conditionalize is very much microarchitecture dependent.
 - Performance of branch predictor and the ability of predicting types of branch in the instruction sequence matters
- Simple if-then-else blocks are not conditionalized
 - Some intentional and others not
- Conditional compares are also missed in many places
 - Currently being looked at

С



What next?





More about Linaro: http://www.linaro.org/about/

More about Linaro engineering: http://www.linaro.org/engineering/

How to join: http://www.linaro.org/about/how-to-join

Linaro members: www.linaro.org/members