CS 530 Fall 2014
Computer Architecture

Homework 1 Solutions

1. Problem 1.13

   a. *If you were choosing just based on overall SPEC performance, which would you choose and why?*

Benchmark	Usage %	Opteron time (sec)	SPEC Ratio	Itanium 2 time (sec)	SPEC Ratio	Opteron/Itanium times (sec)
wupwise	60	51.5	31.06	56.1	28.53	0.92
ammp	20	136.0	16.14	132.0	16.63	1.03
apsi	20	150.0	17.36	231.0	11.27	0.65
Geometric mean			**20.57**		**17.49**	

Based on overall SPEC performance of the benchmarks that have similarities with the company's applications, I would choose the Opteron since it has a higher overall SPEC ratio than the Itanium 2 (20.57 > 17.49), and a higher SPEC ratio for wupwise and apsi which will constitute 80% of its run time usage.

   b. *What is the weighted average of execution time ratios for this mix of applications for the Opteron and Itanium 2?*

The weighted average of execution time ratios (WT) can be computed as follows,

$$WT = \sum_{i=1}^{3} \frac{Opteron\ time_i}{Itanium\ 2\ time_i} \times w_i$$

$$WT = (0.92)(0.6) + (1.03)(0.2) + (0.65)(0.2)$$
$$WT = 0.888$$

   c. *What is the speedup of the Opteron over the Itanium 2?*

The speedup of the Opteron over the Itanium 2 can be computed as follows,

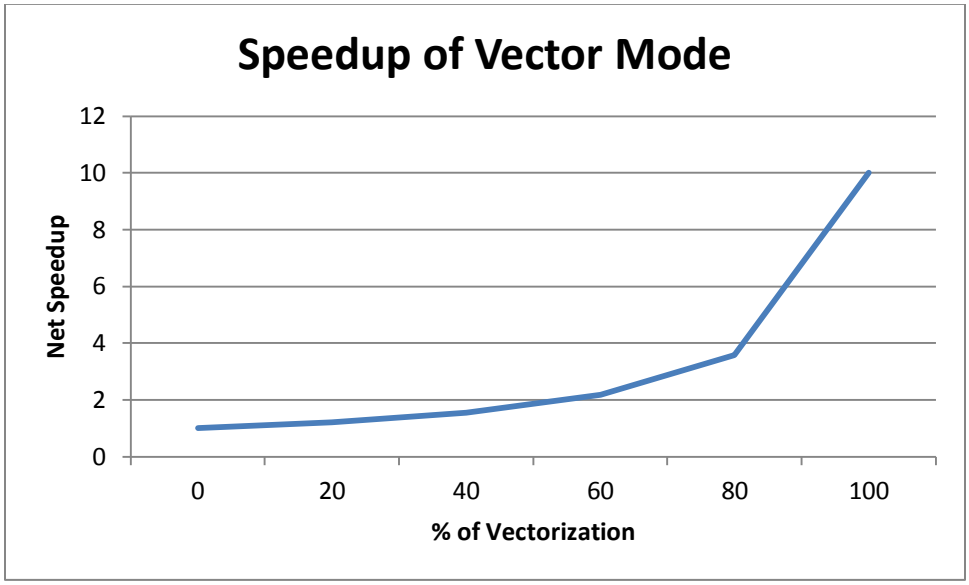
$$speedup_{Opteron} = \sum_{i=1}^{3} \frac{Itanium\ 2\ time_i}{Opteron\ time_i} \times w_i$$

$$speedup_{Opteron} \approx \frac{1}{0.888}$$
$$speedup_{Opteron} \approx 1.126$$

The Opteron is about 1.126x faster than the Itanium 2.

2. Problem 1.14

a. *Draw a graph that plots the speedup as a percentage of the computation performed in vector mode.*



b. *What percentage of vectorization is needed to achieve a speedup of 2?*

Use Amdahl's Law and solve,

$$2 = \frac{1}{(1-V) + \frac{V}{10}}$$

$$V = \frac{5}{9} \approx 0.\overline{55}$$

About 55.56% of vectorization is needed to achieve a speedup of 2.

c. *What percentage of the computation run time is spent in vector mode if a speedup of 2 is achieved?*

Achieving a speedup of 2 tells us that the total run time with the vector hardware enhancement takes half the original time. This enhancement provides a speedup of 10 to the fraction of vectorization. In part (b) we obtained the fraction of vectorization (V) needed to achieve a speedup of 2, therefore, we can compute the percentage of computation run time in vector mode (CRV) as follows,

$$CRV = \frac{Time_{vector}}{Time_{vector} + Time_{non-vector}} = \frac{\frac{V}{speedup_{enhanced}}}{\left(\frac{V}{Speedup_{enhanced}}\right) + (1-V)}$$

$$CRV = \frac{\frac{V}{speedup_{enhanced}}}{\frac{1}{2}}$$

$$CRV = \frac{\left(\frac{\left(\frac{5}{9}\right)}{10}\right)}{\frac{1}{2}} = 0.\overline{11}$$

The percentage of computation run time spent in vector mode is about 11.11%.

d. *What percentage of vectorization is needed to achieve one-half the maximum speedup attainable from using vector mode?*

From part (a) we know that the maximum speedup attainable is 10, therefore, we need to find the percentage of vectorization needed to achieve a speedup of 5. Use Amdahl's Law and solve,

$$5 = \frac{1}{(1 - V) + \frac{V}{10}}$$

$$V = \frac{8}{9} \approx 0.\overline{88}$$

About 88.89% of vectorization is needed to achieve a speedup of 5.

e. *What percentage of vectorization would the compiler team need to achieve in order to equal an addition 2x speedup in the vector unit (beyond the initial 10x)?*

First, let us compute the overall speedup (S) attainable if an additional 2x speedup enhancement is provided by the vector unit. Use Amdahl's Law and solve,

$$S = \frac{1}{(1 - 0.70) + \frac{0.70}{2 \times 10}}$$

$$S \approx 2.985$$

Now we need to find the percentage of vectorization (V) needed to achieve a 2.985 speedup using the vector unit with its original speedup enhancement of 10. Use Amdahl's Law and solve,

$$2.985 = \frac{1}{(1 - V) + \frac{V}{10}}$$

$$V = \frac{1.985}{2.6865} \approx 0.7389$$

The compiler team would need about 73.89% of vectorization to achieve the same speedup if the vector unit provides 2x its original speedup.

3. Problem 1.15

   a. *What is the speedup we have obtained from fast mode?*

   Consider $T_{enhanced}$ as the total run time with the enhanced mode and $T_{non-enhanced}$ as the total run time without the enhancement. Since the enhanced mode is used 50% of the time and the speedup factor is 10, we can express the non-enhanced time using the fraction of time that does not makes use of the enhancement and the fraction of time that makes use of the enhancement but takes 10 times longer,

$$T_{non-enhanced} = (0.5 \times T_{enhanced}) + (10)(0.5 \times T_{enhanced})$$

   The overall speedup (S) can be computed as follows,

$$S = \frac{T_{non-enhanced}}{T_{enhanced}}$$

$$S = \frac{(0.5 \times T_{enhanced}) + (10)(0.5 \times T_{enhanced})}{T_{enhanced}}$$

$$S = 5.5$$

   A speedup of 5.5 is obtained from fast mode.

   b. *What percentage of the original execution time has been converted to fast mode?*

   Use Amdahl's Law and solve,

$$5.5 = \frac{1}{(1 - V) + \frac{V}{10}}$$

$$V = \frac{4.5}{4.95} \approx 0.\overline{90}$$

   The percentage of the original execution time that has been converted to fast mode is about 90.9%.

4. Problem A.1: *Compute the effective CPI for integer instructions.*

Instructions	Instruction Type	gap	gcc	Integer average	Clock cycles
Load	Loads-stores	26.5%	25.1%	25.8%	1.4
Store	Loads-stores	10.3%	13.2%	11.75%	1.4
Add	ALU	21.1%	19%	20.05%	1
Sub	ALU	1.7%	2.2%	1.95%	1
Mul	ALU	1.4%	0.1%	0.75%	1
Compare	ALU	2.8%	6.1%	4.45%	1
Load imm	ALU	4.8%	2.5%	3.65%	1
Cond branch	Cond branches	9.3%	12.1%	10.7%	1.8*
Cond move	ALU	0.4%	0.6%	0.5%	1
Jump	Jump	0.8%	0.7%	0.75%	1.2
Call	Jump	1.6%	0.6%	1.1%	1.2
Return	Jump	1.6%	0.6%	1.1%	1.2
Shift	ALU	3.8%	1.1%	2.45%	1
AND	ALU	4.3%	4.6%	4.45%	1
OR	ALU	7.9%	8.5%	8.2%	1
XOR	ALU	1.8%	2.1%	1.95%	1
Other logical	ALU	0.1%	0.4%	0.25%	1

*The clock cycles for the conditional branches (taken and not taken) are combined into a single value,

$$CC_{cond\ branch} = (0.6)(2.0) + (0.4)(1.5)$$
$$CC_{cond\ branch} = 1.8$$

The effective CPI is computed by combining the average of instructions issued and their corresponding clock cycles,

$$CPI = \frac{\sum_{i=1}^{17} integer\ average_i \times clock\ cycles_i}{100}$$

$$CPI \approx 1.24$$

5. Problem A.3: *Compute the effective CPI for floating-point instructions.*

Instructions	Instruction Type	lucas	swim	FP average	Clock cycles
Load	Loads-stores	10.6%	9.1%	9.85%	1.4
Store	Loads-stores	3.4%	1.3%	2.35%	1.4
Add	ALU	11.1%	24.4%	17.75%	1
Sub	ALU	2.1%	3.8%	2.95%	1
Mul	ALU	1.2%	0%	0.6%	1
Load imm	ALU	1.8%	9.4%	5.6%	1
Cond branch	Cond branches	0.6%	1.3%	0.95%	1.8*
Shift	ALU	1.9%	0%	0.95%	1
AND	ALU	1.8%	0%	0.9%	1
OR	ALU	1%	7.2%	4.1%	1
Load FP	Load-store FP	16.2%	16.8%	16.5%	1.5
Store FP	Load-store FP	18.2%	5%	11.6%	1.5
Add FP	FP add	8.2%	9%	8.6%	4
Sub FP	FP add	7.6%	4.7%	6.15%	4
Mul FP	FP mul	9.8%	6.9%	8.35%	6
Div FP	FP divide	0%	0.3%	0.15%	20
Mov reg-reg FP	Other FP	1.8%	0.9%	1.35%	2
Compare FP	Other FP	0.8%	0%	0.4%	2
Cond mov FP	Other FP	0.8%	0%	0.4%	2
Other FP	Other FP	1.6%	0%	0.8%	2

*The clock cycles for the conditional branches (taken and not taken) are combined into a single value,

$$CC_{cond\ branch} = (0.6)(2.0) + (0.4)(1.5)$$
$$CC_{cond\ branch} = 1.8$$

The effective CPI is computed by combining the average of instructions issued and their corresponding clock cycles,

$$CPI = \frac{\sum_{i=1}^{20} FP\ average_i \times clock\ cycles_i}{100}$$

$$CPI \approx 2.12$$

6. Problem A.8

   a. *Is it possible to have instruction encodings for the following? 3 two-address instructions, 30 one-address instructions, 45 zero-address instructions*

      The total number of instructions required is $3 + 30 + 45 = 78$.
      The total number of possible instructions is $(2^2 - 1) + (2^5 - 1) + 2^6 = 98$.
      This is a valid encoding.

   b. *Is it possible to have instruction encodings for the following? 3 two-address instructions, 31 one-address instructions, 35 zero-address instructions*

      The total number of instructions required is $3 + 31 + 35 = 69$.
      The total number of possible instructions is $(2^2 - 1) + (2^5 - 1) + 2^5 = 66$.
      This is not a valid encoding. A single unique combination remains after satisfying the 31 one-address instructions, thus only 32 zero-address instructions can be encoded with the remaining 5 bits.

   c. *3 two-address instructions, 24 zero-address instructions. What is the maximum number of one-address instructions that can be encoded for this processor?*

      The 24 zero-address instruction encodings can be satisfied with the rightmost 5 bits. This means that we only need a single unique combination from the one-address combinations to expand the opcode, therefore, the maximum number of one-address encodings is $(2^5 - 1) = 31$.

7. Problem A.11: *What is the size of the foo struct? What is the minimum size required for this struct, assuming you may arrange the order of the struct members as you wish? What about for a 64-bit machine?*

Struct members	Size in bytes	
	32-bit machine	64-bit machine
Char	1	1
Bool	1	1
Int	4	4
Double	8	8
Short	2	2
Float	4	4
Double	8	8
Char *	4	8
Float *	4	8
Int	4	4

Since the largest member in the struct consists of 8 bytes, then all members of the struct will be aligned and padded according to 8 bytes for both, 32-bit and 64-bit machines. This alignment policy may vary depending on the compiler, for example, gcc 32-bit manages double data types using 2 memory addresses so it aligns to 4 bytes by default (use –malign-double flag) while Visual C/C++ 32-bit aligns double data types to 8 bytes.

Assume each list of struct members enclosed by a parenthesis represents a collection of 8 bytes (Visual C/C++) and 4 bytes (gcc), and integers in square brackets represent padding bytes.

32-bit Visual C: (char, bool, int, [2]) (double) (short, float, [2]) (double) (char *, float *) (int, [4])

$$(6)(8\ bytes) = 48\ bytes$$

32-bit gcc: (char, bool, [2]) (int) ((double)) (short, [2]) (float) ((double)) (char *) (float *) (int)

$$(11)(4\ bytes) = 44\ bytes$$

64-bit: (char, bool, int, [2]) (double) (short, float, [2]) (double) (char *) (float *) (int, [4])

$$(7)(8\ bytes) = 56\ bytes$$

To find the minimum size required for this struct we can rearrange the members from largest to smallest.

32-bit: (double) (double) (char *, float *) (int, int) (float, short, bool, char)

$$(5)(8\ bytes) = 40\ bytes$$

64-bit: (double) (double) (char *) (float *) (int, int) (float, short, bool, char)

$$(6)(8\ bytes) = 48\ bytes$$

8. Problem A.20

a. What is the average length of an executed instruction?

Figure A.31 presents cumulative percentage of coverage on memory references for multiple offset size magnitudes. These offset sizes do not include the sign bit, but our offset bits require a sign bit. The following table presents the independent percentage of coverage on memory references for the offset magnitudes permitted in the design.

Offset bits	Number of offset magnitude bits	Data references	Branches
0	0	30.4%	0.1%
8	7	36.5%	85.1%
16	15	33.1%	13.3%
24	23	0%	1.5%

To compute the overall average length in bits of an executed instruction ($IL_{overall}$) we need to find the average length for each type of instruction presented in Figure A.27.

$$IL_{overall} = IL_{ALU} + IL_{load-store} + IL_{branch}$$

ALU instructions are executed without restriction and do not have offset bits.

ALU Instructions	Integer average
Add	19%
Sub	3%
Mul	0%
Compare	5%
Load imm	2%
Cond move	1%
Shift	2%
AND	4%
OR	9%
XOR	3%
Other logical	0%

We can compute the overall ALU average instruction length (IL_{ALU}) in bits using,

$$IL_{ALU} = \frac{\sum_{i=1}^{11} integer\ average_i}{100} \times 16\ bits$$

$$IL_{ALU} = (0.48)(16) = \mathbf{7.68\ bits}$$

On the other hand, loads-stores make use of the percentages of coverage on memory references to compute the average instruction length($IL_{load-store}$).

Load-store Instructions	Integer average
Load	26%
Store	10%

$$IL_{load-store} = \sum_{i=1}^{2} \frac{integer\ average_i}{100} \times \sum_{j=1}^{4} \left[\frac{\%\ of\ coverage_j}{100} \times (16 + offset_j) \right]$$

$$IL_{load-store} = \sum_{i=1}^{2} \frac{integer\ average_i}{100} \times [(0.304)(16) + (0.365)(24) + (0.331)(32) + (0)(40)]$$

$$IL_{load-store} = \sum_{i=1}^{2} \frac{integer\ average_i}{100} \times 24.216$$

$$IL_{load-store} = (0.26 + 0.10)(24.216)$$

$$IL_{load-store} \approx \mathbf{8.72\ bits}$$

Branches also make use of the percentages of coverage on memory references to compute the average instruction length(IL_{branch}).

Branch Instructions	Integer average
Cond branch	12%
Jump	1%
Call	1%
Return	1%

$$IL_{branch} = \sum_{i=1}^{4} \frac{integer\ average_i}{100} \times \sum_{j=1}^{4} \left[\frac{\%\ of\ coverage_j}{100} \times (16 + offset_j) \right]$$

$$IL_{branch} = \sum_{i=1}^{4} \frac{integer\ average_i}{100} \times [(0.001)(16) + (0.851)(24) + (0.133)(32) + (0.015)(40)]$$

$$IL_{branch} = \sum_{i=1}^{4} \frac{integer\ average_i}{100} \times 25.296$$

$$IL_{branch} = (0.12 + 0.01 + 0.01 + 0.01)(25.296)$$

$$IL_{branch} \approx \mathbf{3.79\ bits}$$

The overall average length of an executed instruction is

$$IL_{overall} = 7.68 + 8.72 + 3.79 = \mathbf{20.19\ bits} \approx \mathbf{2.52\ bytes}$$

b. *Determine the number of instruction bytes fetched in this machine with fixed instruction size versus those fetched with a byte-variable-sized instruction as defined in part (a).*

Similar to part (a), we need to compute the average length of executed instructions but with a fixed instruction size of 24 bits,

$$IL_{overall-fixed} = IL_{ALU} + IL_{load-store} + IL_{branch}$$

ALU average instruction length is computed as follows,

$$IL_{ALU} = \frac{\sum_{i=1}^{11} integer\ average_i}{100} \times 24\ bits$$

$$IL_{ALU} = (0.48)(24) = \mathbf{11.52\ bits}$$

Loads-stores and branches require additional instructions for offsets longer than 8 bits. I will assume that the additional instruction corresponds to the same type of instruction executed and long offsets are built by combining multiple 8 bit offsets. Since the 8 bit offset includes the sign bit, the offsets are combined using 7 magnitude bits.

Number of instructions executed	Number of offset magnitude bits	Data references	Branches
1	7	66.9%	85.2%
2	14	25%	12.9%
3	21	8.1%	1.9%

$$IL_{load-store} = \sum_{i=1}^{2} \frac{integer\ average_i}{100} \times \sum_{j=1}^{3} \left[\frac{\%\ of\ coverage_j}{100} \times 24 \times j \right]$$

$$IL_{load-store} = \sum_{i=1}^{2} \frac{integer\ average_i}{100} \times (24)[(0.669)(1) + (0.25)(2) + (0.081)(3)]$$

$$IL_{load-store} = \sum_{i=1}^{2} \frac{integer\ average_i}{100} \times 33.888$$

$$IL_{load-store} = (0.26 + 0.10)(33.888)$$

$$IL_{load-store} = \mathbf{12.20\ bits}$$

$$IL_{branch} = \sum_{i=1}^{4} \frac{integer\ average_i}{100} \times \sum_{j=1}^{3} \left[\frac{\%\ of\ coverage_j}{100} \times 24 \times j \right]$$

$$IL_{branch} = \sum_{i=1}^{4} \frac{integer\ average_i}{100} \times (24)[(0.852)(1) + (0.129)(2) + (0.019)(3)]$$

$$IL_{branch} = \sum_{i=1}^{4} \frac{integer\ average_i}{100} \times 28.008$$

$$IL_{branch} = (0.12 + 0.01 + 0.01 + 0.01)(28.008)$$

$$IL_{branch} = \mathbf{4.20\ bits}$$

The overall average length of an executed instruction is

$$IL_{overall-fixed} = 11.52 + 12.20 + 4.20 = \mathbf{27.92\ bits} \approx \mathbf{3.49\ bytes}$$

The fixed instruction size fetches 3.49 bytes per instruction while the variable instruction size fetches 2.52 bytes per instruction. The fixed instruction size requires approximately 1.38x more bytes than the variable instruction size.

c. *Now suppose we use a fixed offset length of 24 bits so that no additional instruction is ever required. How many instruction bytes would be required? Compare this result to your answer to part (b).*

An offset length of 24 bits corresponds to an instruction size of 40 bits (5 bytes). Since 24 bit offset allows 100% coverage on memory references, each instruction will be executed once and the average bytes fetched per instruction would be 5 bytes. Note that the total instruction distribution results in 99% instead of 100% from the data in Figure A.27.

$$IL_{overall-fixed} = \frac{99\%}{100} \times 40\ bits$$

$$IL_{overall-fixed} = \mathbf{39.6\ bits} \approx \mathbf{5\ bytes}$$

Using a fixed offset size of 24 bits results in fetching 1.43x more bytes per instruction than using the fixed offset size of 8 in part (b), and 1.98x more bytes per instruction than the byte-variable offset in part (a).