

CSSE3001/7000 2007

Prac 5: Optimising for Performance

Last Update: 20/04/2007 01:37:00 PM

Author: [Simon Leung](#)

Preparations

You will need to have a good understanding of Prac 1, 2 3 and 4 as this prac will explore all those skills to optimise a simple program.

Don't forget you need to use the very useful reference document "[The SPARC Architecture Manual, version 9](#)" (also [available internally](#)) to look up information. Use it smartly and carefully, looking at the relevant sections and use the search function of the PDF reader to help you navigate in this book.

In addition, the VIS Instruction Set User's Manual ([available internally](#)) will be useful.

Setup

For this prac, we will use compilers from SUN (Sun Workshop 6). You will need to setup your path to the SUN compiler installed in agave, so make sure \$PATH has these entries:

```
/opt/SUNWspro/bin:/usr/ccs/bin:/opt/local/bin:/usr/bin\  
:/sbin:/usr/sbin:/opt/local/sbin:/usr/openwin/bin.
```

For example, the commands:

- `PATH=/opt/SUNWspro/bin:/usr/ccs/bin:/opt/local/bin\
:/usr/bin:/sbin:/usr/sbin:/opt/local/sbin:/usr/openwin/bin`
- `export PATH`

will achieve the desired result.

The VIS toolkit is installed on `/opt/local/pkg/cad/csse3001/SUNWv sdk`. You will need to add some additional command flags at compile time to integrate VIS instructions into your program.

Procedure

Task 1: Original Program

Let's assume that we need to sum a large (1 dimensional) array of numbers. Each element of this array is 16 bit in size and contains a number ranged from 1 to 16. We need to setup the data first:

```
#include <stdio.h>  
  
#define size 16  
#define group 1024  
#define repeat_times 1024
```

```

short int array[size*group*repeat_times];

void init()
{
    int i;
    short int j;
    for(i=0; i<group*repeat_times; i++)
    {
        for (j=0; j<size; j++)
        {
            array[i*size+j]=j+1;
        }
    }
}

```

Examine the code segment above, and answer the following questions:

1. In the code above, what are the maximum and minimum values for j?
2. Hence, or otherwise, does the code satisfy the ranged required in our task?
3. How are the numbers arranged in the array? What is the value for: element 1? element 2? element 16? element 17?
4. What is the purpose of the index `i*size+j`?

Now that the array is setup with some values, we will need to find the sum of all the values in the array. The most straight forward method would probably be:

- i. initialise `sum=0`;
- ii. add the value of the first element to `sum`;
- iii. go to the next element, and add it to the `sum`;
- iv. repeat the step above until the end of the array.

We can implement the function as:

```

int sum_array(short int* array)
{
    int i=0;
    int sum=0;
    while (i++!=size*group*repeat_times)
    {
        sum += *array++;
    }
    return sum;
}

```

Answer the following questions about the `sum_array` function:

1. How is the array passed into the `sum_array` procedure?
2. Why is the function of type `int`? Why not `short int`?
3. How many times will the while loop be run? Is it one iteration too many, too few or correct?
4. What is the precedence of operators for the line `sum += *array++`? Will this line sum the numbers correctly?

Now that we have the two essential functions, we also need to keep track of the time taken to calculate the sum. In the previous prac, we have seen some constructs which access the real time clock of the server. That's fine if the real time clock is not adjusted by external factors, eg. daylight saving changes. We can access another time-keeping mechanism - a free running counter incrementing every nanosecond.

```

#include <sys/time.h>
static hrtime_t start_time, end_time;

int main()
{
    int sum;
    init();                /* initialise the array */

    start_time=gethrtime(); /* record the value of counter at the start of
sum */
    sum=sum_array(array);
    end_time=gethrtime();  /* record the value of counter when sum is
finished */

    printf("The time taken to sum the array is %5.3f in miniseconds.\n",
        ((double)(end_time-start_time)*1.0e-6)); /* the multiplication converts
ns to ms */
    printf("The sum is %d.\n", sum);
    return 0;
}

```

Combine the code fragments above, and answer the following questions:

1. How many mini-seconds does it take to sum the array? Use the SUN compiler without any optimisation (it should be between 400 to 500ms).
2. What is the sum? Calculate the sum manually and compare it to the result given by the program.
3. Examine the "hot loop" (the code which is executed most often) and its assembly equivalent. Is the assembly code efficient? Are all the memory loads and stores useful? What would you do to optimise this assembly routine?

Task 2: Compiler Optimisation

After you have checked and corrected the code fragments above, let's see how much performance improvement would the compiler be able to perform for us.

1. Compile the code above using the SUN compiler, but add the optimisation flag, -O (note this is capital letter O), as the last flag.
2. How much time does it take to run the program now? (it should be between 150 to 200ms)
3. So it seems that compiler optimisation can double the performance of this binary. What is the difference between this version and the optimised version? Generate the assembly output of the optimised code (-S flag) and observe the code produced for the hot loop. Does it now have most of the optimisation you have suggested in the previous part?
4. Why does the compiler choose to use the global registers in the optimised code?

Of course there are other optimisation flags available, so feel free to take a look at the manual pages and try them out.

Task 3: VIS Optimisation

Compilers can only help us to a certain extent. Smart programmers would take a look at their programs and algorithm to see if there are other ways of improving the performance.

In our previous studies, we have encountered the set of VIS instructions provided by SUN. They are specifically designed to take advantage of the SIMD hardware resources provided by the microprocessor hardware. In the VIS instruction set, there is an arithmetic instruction which will let us perform four 16-bit signed addition at the same time. This is potentially very useful for us, as we can exploit the parallelism of this instruction to speed up our "hot loop".

- Read Ch.4.6.1 of the VIS Instruction Set User Guide.

As you are probably aware, VIS instructions only operates on the floating point registers. So we need to load our data from memory to the floating point registers, and it makes sense to load 4 short integers at the same time.

- What is the VIS instruction which loads 4 short integers into a single 64-bit floating point register? (hint: see previous prac or VIS manual)

Since we can perform four addition simultaneously, we need four spaces to hold the temporary results. One other float point register is perfect for such purpose. Now that the integers are loaded, we need to add the content to the temporary space.

- What is the VIS instruction which performs addition on 2 sets of 4 short ints?

The next logical step is to load the next set of 4 numbers, and add them to the temporary sums. It is tempting just to do this until all data is exhausted, but there is a trap - overflow. Since the addition results are stored into a short integer, which has a maximum value of 32767, and our numbers has a maximum of 16, there can only be 2047 additions (in the worst case scenario) before there is an overflow.

- Can the VIS instruction that we are using handle overflow?

On the safe side, let us perform only 1024 additions and then put the temporary sums into a long integer for safe keeping.

- Accounting for the fact that we can now perform 4 additions simultaneously, how many loops do we need to sum the entire array?

Examine the following code to see if it satisfy the discussions above, and identify the functionality of each line:

```
int sum_array(short int* array)
{
    int i=0, j;
    int sum=0;
    short int temp_sums[4];
    double d1,d2=0.0;
    while (i++!=size*repeat_times/4)
    {
        for (j=0; j<group; j++)
        {
            d1 = vis_ld_d64(array);
            d2 = vis_fpadd16(d1,d2);
            array += 4;
        }
        vis_st_d64(d2,&temp_sums);
        sum = sum+temp_sums[0]+temp_sums[1]+temp_sums[2]+temp_sums[3];
        d2 = 0.0;
    }
    return sum;
}
```

Modify the code fragments above to use this function to sum the array. Remember, you will need additional `#include` statements and compiler flags for programs with VIS specific instructions.

1. Compile and run the code, without the `-O` flag. How much time does it take to complete the sum? Is it better or worse than our original program?
2. Compile and run the code, with the `-O` flag. How much time does it take to complete the sum? Is

- it worthwhile to use the VIS instructions?
3. Examine the assembly code generated by the compiler for the two version of the programs in the previous two questions. Why are the performance so different?
 4. Is there another way to further increase the performance of the sum function using additional/other VIS instructions? Try them out.

Task 4: Memory Access Optimisation

In this program, the other factor which will governs the performance is memory access - how fast can the microprocessor fetch the data from the array in memory into its registers for processing?

Because there is no data dependence, the array elements should be put into the fastest memory available (cache), but it is not quite possible to put the entire array into the cache due to the size. So, if we can tell the program to simultaneously fetch the NEXT set of data while processing the current set, it may be very beneficial to the performance indications.

Add `#include <sun_prefetch.h>` to the start of the program, and add the bold statement to the code:

```
int sum_array(short int* array)
{
    int i=0,j;
    int sum=0;
    short int temp_sums[4];
    double d1,d2=0.0;
    while (i++!=size*repeat_times/4)
    {
        for (j=0; j<group; j++)
        {
            d1 = vis_ld_d64(array);
            d2 = vis_fpadd16(d1,d2);
            array += 4;
            sparc_prefetch_read_many(array+16);
        }
        vis_st_d64(d2,&temp_sums);
        sum = sum+temp_sums[0]+temp_sums[1]+temp_sums[2]+temp_sums[3];
        d2 = 0.0;
    }
    return sum;
}
```

The prefetch statement tells the microprocessor to prefetch 16 ints (or 32 short ints) from the current location pointed to by "array" pointer.

1. Compile this code (with prefetch statement, and don't forget the -O flag) and check its performance. Is there any significant difference?
2. If there is no significant difference, it may be due to the fact that we are still consuming the data faster than the prefetch process. Increase the number of ints prefetched from 16 to 64. Is there any significant difference now?
3. Investigate if more/less prefetching will help the performance.
4. Would using prefetching alone, without VIS, yield similar performance increase?

Reflection

- How much has the performance improved for this program during this prac? In your opinion, is the effort worth it?
- Would it be better if we code in assembly? Or at least, change the "hot loop" into handcrafted assembly code?
- Is the hassle of using VIS worth the performance improvement that we achieved?

- What is the best way to treat compilers? Can they optimise everything automatically? How would you intervene in the compiling process to achieve performance?

Summary

In this prac, we have covered the process of optimising a program using a range of techniques that we acquired from previous pracs.

Suggested Exercises

You should reinforce your understanding of these concepts by modifying the code and try to make some mistakes and redundant steps. As a suggestion, experiment with the other VIS instructions listed in the manual. Try finding another program to optimise - that is your project afterall.

Throughout this prac we have been using the C compiler from SUN, designed by SUN and optimised for their SPARC architecture. Most of you would be familiar with gcc, a free GNU compiler. It is also available on agave as long as you don't modify the PATH variable at the start of the prac. What can you do with gcc for optimisation?
