Experimental Determination of ILP

Jessa Rothenberg

July 13, 2007

This summer I am developing techniques to expose Instruction Level Parallelism (ILP) through the use of register tiling. ILP is the amount of operations that can be performed at the same time in a single processor, due to pipelining and/or superscalar architecture features. The goal of this experiment was to determine how much various machines could take advantage of ILP.

Many simulations performed by computers involve task graphs with regular dependences, which calculate changes in data over time. By breaking up the graph into blocks (called register tiles) the programmer can use techniques that allow the compiler to better optimize code to allow for parallelism in the instruction pipeline. In other words, by replacing array references with scalar addresses, the compiler is able to assign values to registers instead of accessing them from memory. This speeds up the compiler and allows for simultaneous, or parallel, operations.

By increasing the number of possible parallel instructions for code that represents simple data dependencies, we can determine maximum ILP experimentally. Our ultimate goal is to use this information to improve the parallelism in computation of partial differential equations.

1 Description of the Project

1.1 The Code

To experimentally determine ILP, we modified the loop from Figure 1. This loop can be modeled graphically as a series of "upwards" data dependencies. Figure 3 shows a representation of this dependence graph. Each "timestep" is represented by i, and each element in the row is represented by j. Even though we are only using a one-dimensional array, the iteration space is two-dimensional. Each previous timestep of `A[`j`]` is used to calculate the next timestep of `A[`j`]`; because `A[`j`]` uses the previous value, the data flows "up" each iteration of i. The ultimate goal is to see the result of the array (of length N) at timestep M.

1.2 Register Tiling

Register tiling is the multidimensional unrolling of code loops in order to enable ILP and improve temporal data locality. Instead of having the code segment from Figure 1, we unroll the loop to produce the code seen in Figure 2. This unrolling is what enables the compiler to do multiple floating point multiplications in parallel. By unrolling the loop, we now have two multiplication instructions with no associated dependencies; the compiler can now set up the processor to do them both at the same time.

```
for (i = 0; i < M; i++)
{
    for (j = 0; j < N; j++)
    {
        A[j] = A[j] * K;
    }
}
```

Figure 1: Original

```
for (jj = 0; jj < N - width + 1; jj += width)
{
    R1 = A[jj];
    R2 = A[jj+1];

    for (i = 0; i < M; i++)
    {
        R1 = R1 * K;
        R2 = R2 * K;
    }

    A[jj] = R1;
    A[jj+1] = R2;
}
```

Figure 2: Unrolled

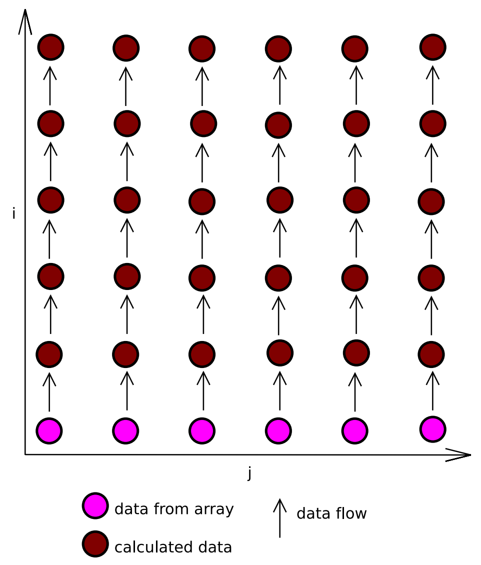

Figure 3: A diagram of the loop

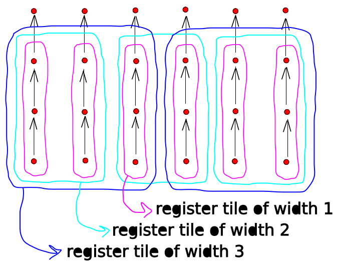

Figure 4: Different Sizes of Register Tiles on a Dependence Graph

Figure 4 gives a visual representation of different widths of register tiles for the loop we are using in our program. Because of the upward dependence, a register tile of width 1 does not allow for parallelism. However, each register tile width greater than one allows for one additional concurrent operation. As you can see in the diagram, a tile of width two has two operations that are data-independent, and as such can execute simultaneously. A tile width of three exposes three parallel operations, and so on.

1.3 Experiment Purpose

In order to determine the ILP for several computers, we can either estimate based on the number of floating point units and pipeline depth, which tends to provide an unrealistic upper bound, or we can determine the value experimentally by performing benchmark tests. We wrote a program to determine the ILP of a machine, by using the above loops with register tiles. The dependence graph is structured such that all the calculations in a particular column are dependent, but all the calculations in a particular row can be performed concurrently. We have register tiles of certain widths, meaning that our loop has been unrolled and each calculation is put into a scalar, which can be used by the registers at compile time. Every time we add one to the register tile size, it means that another computation can be performed concurrently. At some point the computer will no longer be able to take advantage of this extra allowed parallelism, at which point the efficiency will level out, and we will have found the computer's level of ILP.

This dependence graph is the same one seen in our explanation of the code. As such, the code uses register tiles such as the ones in Figure 4. Every time we increase the tile width, our program should become more efficient as the computer takes advantage of ILP. At some point, the computer can no longer perform another operation simultaneously, and we see the performance level out. By starting with a tile width of one and increasing it gradually, we can make a graph that displays performance increase until we have achieved the maximum amount of ILP for a particular computer. This enables us to visually determine the effective maximum amount of ILP.

2 Results

2.1 Expected Results

We wanted to see the effect of register tile width on processor efficiency (operations per cycle). To measure this, we ran a program with 6 different tile widths. We ran the program on four different computers, using 5 different optimization levels, different values of M (the number of times the loop iterates), and different constants K (where

Machine	Description	MHZ	RAM	L1	OS
Carstensz	HP Core 2 Duo	2x1.86G	2Gb	32KB	Linux(64)
Grace	HP Opteron 250	2x2.4Gh	2Gb	128KB	Linux(64)
Faure	Sun Ultrasparc	2x450Mh	1.5Gb	???	Solaris
Nelson	Dell Xeon	1.5Ghz	512Mb	8KB	Linux(32)

Table 1: Computers Used in Experiment

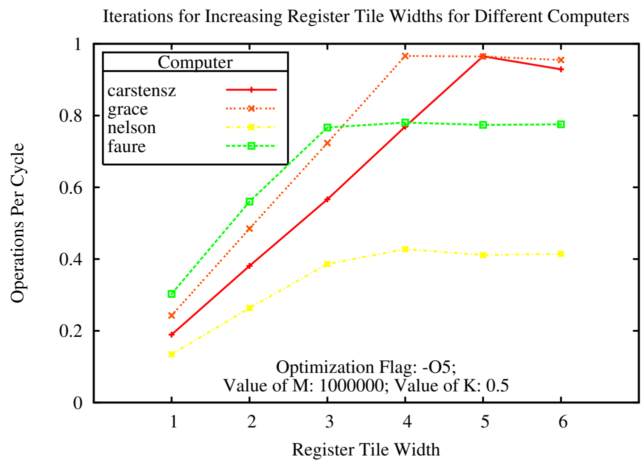

Figure 5: ILP for each computer

`A[j] = A[j]*K;`). Through comparing various different variables, we were able to determine each computer's ILP, as seen in the below graphs, Figures 5 and 6.

Every time we added one to the register tile width, we would expect the code efficiency (operations per cycle) to increase by the initial number of operations per cycle, up to some maximum efficiency, at which point the program cannot make further use of ILP.

2.2 Experimental Results

Figure 5 shows the code efficiency for each of the four computers tested. The four computers, as seen in Table 1, were Carstensz, Grace, Faure, and Nelson. Carstensz and Grace are both 64-bit Linux machines. However, Carstensz is dual core, and Grace is dual processor. Carstensz has two 1.86 GHz intel cores. Grace has two 2.4 GHz processors. Faure is a Solaris machine with a single 450 MHz processor. Nelson is a 32-bit Linux machine with a single 1.5GHz processor. Carstensz and Grace had very similar performance times and operations per cycle, Faure being slower than those, and Nelson being much slower and also less efficient. All computers behave similarly with a rapid increase in efficiency before hitting a point of maximum efficiency.

For Carstensz, there is a very clear linear increase in efficiency for the first 5 tile widths. The efficiency of a register tile of width 1 is approximately 0.2 operations per cycle. Every added register tile up to 5 increases the efficiency by almost 0.2 operations per cycle. This is very close to the ideal results, where operations per cycle would increase equally with every added register tile width, before reaching some peak efficiency (here, one operation per cycle).

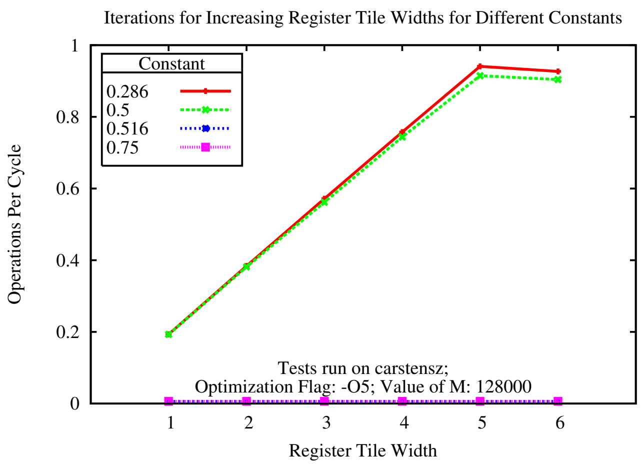

Figure 6: Constant Comparison for Register Tile Width Efficiency

As seen in Figure 5, it appears that Carstensz reached its maximum efficiency with a register tile width of 5; Grace and Nelson appear to reach their maximum efficiency with a register tile width of 4, and Faure reaches its maximum efficiency with only a register tile width of 3. These numbers appear to indicate that Carstensz can at most do 5 floating point operations concurrently, Grace and Nelson can do 4, and Faure can do 3.

The efficiency of different optimization settings for any given computer was pronouncedly lower when unoptimized (flag `-O0`). Notably, all other optimizations, from `-O1` to `-O5`, have the same operations per cycle for the first 6 tile widths.

Figure 6 shows that the efficiency (in operations per cycle) to run the program for the constant K = `3.0/4.0` is vastly different from that of K = `1.0/2.0`. With no compiler optimization, the program is less efficient than that of K = `1.0/2.0` by two orders of magnitude. Additionally, changing the size of the register tile has no appreciable effect on the calculations. However, except for the constant values, the assembly code for these two versions of the program is identical. The problem is not merely one of optimization; even with optimization flag `-O5`, there is minimal improvement in performance. Additionally, the optimization caused rounding errors for one of the computers (Nelson). On further inspection, it was found that many constants $K > 0.5$ behaved similarly, and many constants from 0.0 through 0.5 also behaved similarly.

2.3 Unexpected Results

The main strange result was the vast difference in behavior between constants greater than 0.5, and constants less than or equal to 0.5. All constants tested from 0.0 through 0.5 yielded similar results that satisfied our expectations of the data. All constants greater than 0.5 yielded similar results that were extremely inefficient.

One computer, Nelson, experienced some bizarre errors not seen in the other machines tested. For example, constants greater than `1.0/2.0` caused rounding errors for optimizations greater than `-O0`. There was also a large jump in execution time between two values of M, the number of iterations. Between M = 16,000 and M = 32,000 there was a three-orders-of-magnitude slowdown; the expected slowdown was 2.

3 Conclusions

We encountered some strange performance issues when multiplying by different constants. We checked the assembly code for the different constants and found no noticeable instruction differences that would explain the performance differences. We also examined the hardware performance counters on one machine, but these results did not match well with the execution times. Due to time constraints, we did not further explore these issues. In the future, we could use PAPI to check the L1, L2, and TLB misses, which may explain the strange behavior. We could also try to compare the assembly code from our program with a simpler test example that gave reasonable results for the number of floating point multiplications.

The goal of this work was to experimentally determine how much these computers could take advantage of Instruction Level Parallelism. Our analysis of the data gathered from these experiments reveals that the largest number of floating point multiplications that each machine was able to do concurrently is relatively small. Carstensz could do five, the most parallel floating point multiplications of all the computers tested. Grace and Nelson each could do four multiplication instructions at once, and Faure could only do three.