ILP and Register Tiling

Jessa Rothenberg

August 16, 2007

Abstract

This summer I developed 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.

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.

The goal of our research is to determine whether or not a machine with register renaming can expose ILP that is not being exposed by the compiler. If this is true, it will enable us to use storage allocations with fewer registers.

1 Description of the Project

1.1 Experiment Purpose

Current register tiling methods do not fully utilize a computer's ILP. Full utilization of available processor ILP can result in 2-4 times speedup. The current approach, loop unrolling, uses too many registers for the needed ILP. Specifically, there is a trade-off between register pressure and exposing the maximum amount of ILP in a particular architecture.

The dependence pattern in our experiment code is often seen when solving partial differential equations on a computer, comparing DNA strings, or computing atmospheric simulations. This dependence pattern is displayed in Fig. 1. The unoptimized code represented by this dependence is shown in Fig. 2. Our code uses the techniques of loop unrolling and scalar replacement to optimize the code. We use register tiles of various sizes, with different storage mappings, to achieve the desired amount of ILP.

The goal of this research was to determine if fewer registers could be used in order to improve run time for a sample dependence graph. By using a new storage mapping, we attempted to reduce the number of registers, while still enabling ILP.

The approach used to examine this question was to use register tiling and scalar replacement with two different storage mappings, one of which used much fewer registers than the other. We will evaluate the performance of these two storage mappings by comparing execution times.

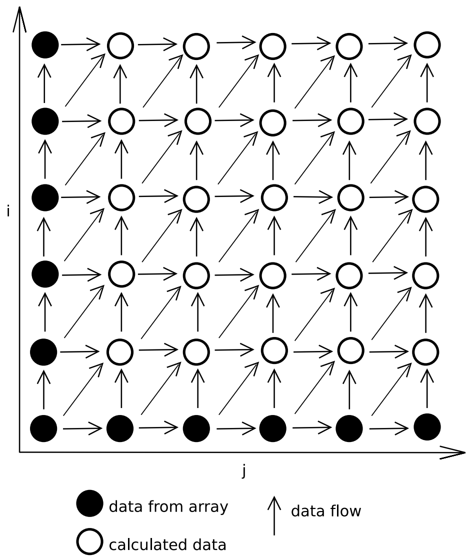

Figure 1: Iteration Space Graph for the computation

```
for (i=1; i<=TIME; i++)
{
   for (j=1; j<=LENGTH; j++)
   {
      A(i,j) = A(i,j-1)*A(i-1,j-1)*A(i-1,j);
   }
}
```

Figure 2: Code for the computation

1.2 Register Tiling & Scalar Replacement

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 3, we unroll the loop to produce the code seen in Figure 4. 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.

Scalar replacement is the assignment of array values to scalars. Each position in the array is represented by a scalar value, which holds the value of the computation. By doing this, the compiler can more easily assign particular array values to registers, which greatly increases performance and makes it easier for the processor to use ILP.

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

Figure 3: 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 4: Unrolled

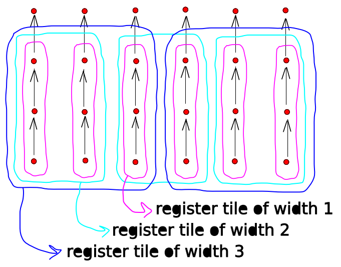

Figure 5: Different Sizes of Register Tiles on a Dependence Graph

```
R0 R1 R2 R3 R4
R5 R6 R7 R8 R9
R0 R1 R2 R3 R4
R5 R6 R7 R8 R9
R0 R1 R2 R3 R4
```

Figure 6: Ping Pong Storage Mapping

```
R2 R3 R4 R5 R0
R3 R4 R5 R0 R1
R4 R5 R0 R1 R2
R5 R0 R1 R2 R3
R0 R1 R2 R3 R4
```

Figure 7: Rotating Registers Storage Mapping

```
* b c d e
* a b c d
* * * * *
```

Figure 8: 5x3 Register Tile

```
* e f g
* d e f
* c d e
* b c d
* a b c
* * * *
```

Figure 9: 4x6 Register Tile

```
* 1
* * 2
  * * 3
    * * 4
      * * 5
        * *
```

Figure 10: Required Unique Registers for ILP 5

Figure 5 gives a visual representation of different widths of register tiles for the example loop shown above (Figure 3). 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 Models for Two Storage Mappings

We are using two different storage mappings, inspired by *Schedule-Independent Storage Mapping for Loops* [2]. The "ping pong" storage mapping is shown in Figure 6. The dots in Fig. 1 will be stored in a separate scalar beginning with R. So, array position (1,1) is stored in R0. The "rotating" storage mapping is shown in Figure 7. This mapping differs from "ping pong" because it reuses registers sooner. Instead of having two full rows of unique registers (like ping pong does), it only uses one more registers than the tile width. This is possible because in a serial schedule, by the time the calculations for array position (2,2) are needed, the data in array position (1,1) is no longer needed by any other value.

Using these storage mappings, we would like to create some models to determine the performance of each mapping. This experiment has four key variables: registers used, parallelism achieved, and width and height of register tiles in the code. We are trying to determine the bounds for the registers. These will be given by determining the minimum registers required for a given storage mapping (Fig. 6 and 7), the minimum registers required for a given ILP, and the maximum registers for a given ISA). Based on these bounds, we can also determine the bounds on our tile sizes (tile width and tile height).

We also want to maximize the number of "full diagonals" in our register tile. A full diagonal is the largest wavefront in a tile. Fig. 8 has an ILP of 2, and so the number of full diagonals is the number of wavefronts of length 2 present in the tile (in this case, 3).

We will be determining functions r (number of registers), p (parallelism, or ILP achieved), d (number of diagonals in the tile), w (tile width), and h (tile height). These functions will be based on given values of ILP (desired ILP), TW (tile width), and TH (tile height).

The upper bound for our registers is equal to the number of register available in the Instruction Set Architecture.

Minimum number of registers for a given mapping as a function of tile width and tile height:
Figure 6, ping pong mapping: $r(TW,TH)_{ping} = 2 * TW$
Figure 7, rotational mapping: $r(TW,TH)_{rotate} = TW + 1$

As you can see, both are independent of tile height.

Minimum number of registers for desired ILP:
$r(ILP)_{min} = 2 * ILP + 1$

In Figure 10, we see that the required number of unique registers to do 5 operations concurrently is 11. If we generalize this, we see that to achieve an ILP of n, we need to use $2n+1$ unique registers. Note that the rotating scheme (Fig. 7) does not have sufficient registers to account for the necessary ILP as a function of tile width and height ($r(ILP)_{rotate} < r(ILP)_{min}$). In fact, this scheme is not valid for a wavefront schedule (unlike ping pong, Fig. 6). However, we predicted that the register renaming on our Intel machines would be able to compensate for the missing registers, enabling the same amount of ILP as for ping pong.

ILP as a function of tile width and tile height (independent of storage mapping): $p(TW,TH) = min(TW,TH) - 1$

Note that the ILP is limited by the smallest dimension of the tile. A 5x3 tile, restricted by the height, will have a maximum ILP of 2 (see Fig. 8), and a 4x6 register tile, restricted by the width, will have a maximum ILP of 3 (see Fig. 9).

Based on the minimum number of registers in terms of tile size, and our knowledge of the relationship between ILP and tile size, we can determine the registers in terms of ILP.

Minimum number of registers for a given mapping as a function of ILP:
Figure 6, ping pong mapping: $r(ILP)_{ping} = r(TW-1) = 2 * ILP + 2$
Figure 7, rotational mapping: $r(ILP)_{rotate} = r(TW-1) = ILP + 2$

We also would like to know something about the attributes of the register tile itself; given the tile size, what is the number of full diagonals in a wavefront pattern (useful for pipelining), and given the desired ILP, what is the ideal minimum tile size?

Number of full diagonals in a wavefront pattern (independent of storage mapping):
$d(TW,TH) = |TW - TH| + 1$

This indicates that a rectangular tile is preferable to a square one.

Desired minimum tile size (independent of storage mapping):
$w(ILP)_{min} = w(p(TW,TH)) = ILP + 1$ $h(ILP)_{min} = h(p(TW,TH)) = ILP + 1$

Therefore the smallest tile size must be $ILP+1$ by $ILP+1$, consisting of $(ILP+1)^2$ calculations.

These models indicate that the ideal tile size is one that uses just enough registers to expose a computer's maximum ILP, while also being as large in the other dimension as possible to allow for more full diagonals; therefore, TW should be at least $ILP+1$, and TH should be as large as possible (and no less than $ILP+1$).

2 Determining Available ILP & Registers

As you can see, the models presented in Section 1.3 are based on a computer's ILP and registers. Though there is a minimum desirable amount of registers, the maximum is determined by the machine characteristics. This is also true for the ILP; a machine will only be able to take advantage of so much ILP, and knowing that number is useful when calculating ideal register tile sizes.

2.1 The Code

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. This particular dependence graph (Figure 5) 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.

Figure 3, which we used in this determination, can be modeled graphically as a series of "upwards" data dependencies. 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.

The code in Figure 4 is an example of "unrolling" the loop in Figure 3. Instead of having a single instruction in the loop body, there are now two; this assists the compiler in assessing and enabling ILP.

This code uses register tiles such as the ones in Figure 5. 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 made a graph that displays performance increase until we have achieved the maximum amount of ILP for a particular computer. This enabled us to visually determine the effective maximum amount of ILP.

2.2 Experimental Determination of Computer Specifications

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	ILP (fp mult)	ILP (fp add)	Registers
Carstensz	HP Core 2 Duo	2x1.86G	2Gb	32KB	Linux(64)	5	3	16
Grace	HP Opteron 250	2x2.4Gh	2Gb	128KB	Linux(64)	4	4	16
Faure	Sun Ultrasparc	2x450Mh	1.5Gb	???	Solaris	3	3	29
Nelson	Dell Xeon	1.5Ghz	512Mb	8KB	Linux(32)	4	3	8

Table 1: Computers Used in Experiment

`A[j] = A[j]*K;`). Through comparing various different variables, we were able to determine each computer's ILP.

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.

We used this method to calculate the ILP for both floating point multiply and floating point add instructions.

In order to determine the number of registers available to each computer, we examined the assembly code for the same code used for ILP calculations. When putting array values into scalars, the amount of registers would be revealed by the last program that had no memory accesses within the innermost loop. If all array values were successfully stored into registers and accessed for the floating point multiply, then that many registers were available to the processor.

2.3 Results

Table 1 shows the computer characteristics and experimentally determined ILP and number of registers.

Carstensz, the HP Core 2 Duo, and Grace, the HP Opteron 250, are both 64-bit Linux machines. However, the HP Core 2 Duo is dual core, and the HP Opteron 250 is dual processor. The HP Core 2 Duo has two 1.86 GHz intel cores. The HP Opteron 250 has two 2.4 GHz processors. Faure, the Sun Ultrasparc, is a Solaris machine with a single 450 MHz processor. Nelson, the Dell Xeon, is a 32-bit Linux machine with a single 1.5GHz processor. The HP Core 2 Duo and the HP Opteron 250 had very similar performance times and operations per cycle, the Sun Ultrasparc being slower than those, and the Dell Xeon being much slower and also less efficient. All computers behave similarly with a rapid increase in efficiency before hitting a point of maximum efficiency.

As seen in Figure 11, it appears that the HP Core 2 Duo reached its maximum efficiency with a register tile width of 5; the HP Opteron 250 and the Dell Xeon appear to reach their maximum efficiency with a register tile width of 4, and the Sun Ultrasparc reaches its maximum efficiency with only a register tile width of 3. These numbers appear to indicate that the HP Core 2 Duo can at most do 5 floating point operations concurrently, the HP Opteron 250 and the Dell Xeon can do 4, and the Sun Ultrasparc can do 3.

When tested with "+" substituted in instead of "*", we were able to determine the computer's ILP for floating point addition. We expected for each computer to be able to have more ILP for addition than for multiplication. However, this was not the case, and our experimentally determined values were an ILP of 4 for the HP Opteron 250, and 3 for the other three computers. We did not have time to examine why this is the case.

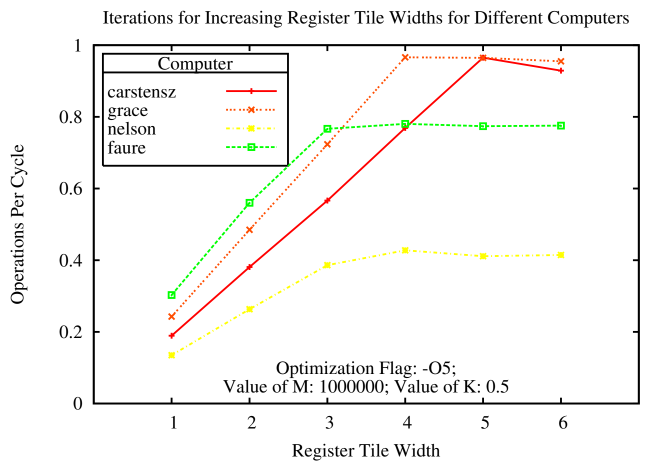

Figure 11: ILP for each computer

3 Comparing Storage Mappings

Now that we have determined the ILP and registers for our computers, we can compare the two different storage mappings.

3.1 Description of Computation

For X-86, 64-bit machines, the ISA has a limit of 16 registers. This means that a ping pong register tile can have a maximum of width 8 before incurring register spilling. A rotating register tile, on the other hand, can have a maximum of width 15 before spilling. Unfortunately, the placement of these registers prevents the machine from fully utilizing ILP.

For computers with register renaming, there are more physical register locations than there are ISA registers. When the computer detects that two different pieces of data are assigned to the same variable (i.e. two array positions A[i] and A[j] being assigned to variable R0), it can "rename" the register, assigning the same register R0 to two separate storage locations [1]. We predicted that this ability to rename registers would allow an out-of-order processor to utilize parallelism for rotating register tiles, resulting in similar execution times for the serial schedules of both ping pong and rotating storage mappings.

The ISG, or iteration space graph, for our computation is the same as that seen in Figure 1.

Register tiles did not fit precisely into the entire grid (400x300), so the left and top edges of the graph were NOT computed. Figure 12 shows a representation of our calculation space. Only the grey/green squares were calculated; empty squares were not, leaving the top few rows and last few columns uncalculated when the register tiles did not precisely fit. This makes the graph slightly "bumpier," as performance will increase when the tile size gets just too big to fit into the entire computation space (at which point it will do fewer computations).

The black circles represent data that was given and pulled from the array to calculate the rest of the values. The white

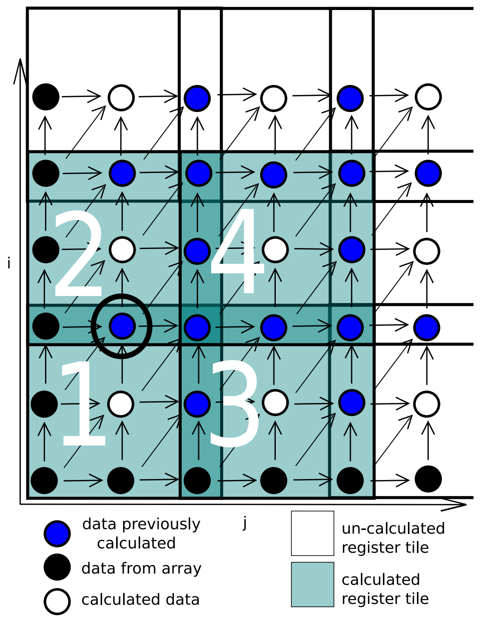

Figure 12: Register Tile "Overflow"

circles represent data that is calculated. The grey/blue circles represent data that was calculated in one register tile, then stored to the array and pulled back from the array to be used in a different register tile. For instance, the circled point, (2,3), is calculated in the bottom left register tile (1), stored to the array, then pulled from the array to be used in the top left register tile (2).

If we were to do the tests over, we would insert clean up code to handle the edges of the graph, as well.

3.2 Results

Figure 13 is a graph showing the times for four different code compilations (baseline, serial schedule ping pong mapping, serial schedule rotating mapping, wavefront schedule ping pong mapping). The baseline is a comparable program with no register tiling. The graph indicates that the tiled programs have much better execution times than this baseline.

The tile width for Figure 13 is 8, such that neither rotating or ping pong mappings will cause register spilling on the Intel Duo Core. The similar performance of both serial schedules supports our hypothesis that register renaming would compensate for lack of a legal wavefront schedule for floating point instructions. The static wavefront schedule, however, does perform better than either serial schedule, suggesting that there is some amount of overhead for the out-of-order computations.

An interesting result, that was not predicted by our models, is that the serial rotating tile does better than the other methods for small tile heights. At tile height 2, for instance, the performance is comparable to the wavefront ping pong performance at larger tile heights. We would like to investigate this further in future experiments.

All computers were tested; the Dell Xeon and the Sun Ultrasparc did not have register renaming, but the HP Core 2 Duo and the HP Opteron 250 did. The Dell Xeon had poorer performance (likely due to lower numbers of registers

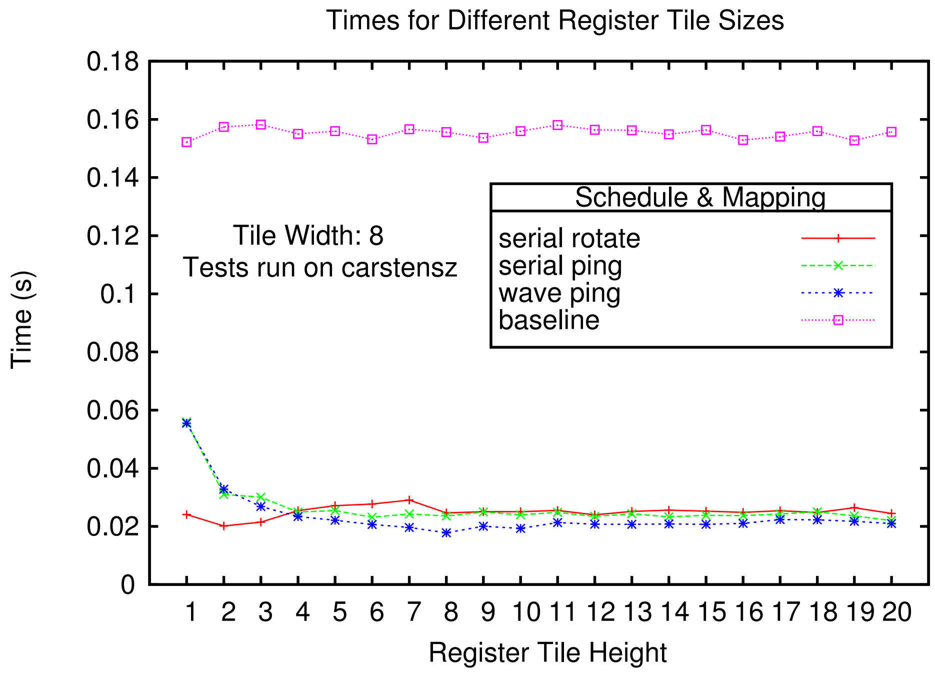

Figure 13: Results of computations

and generally slower hardware capabilities), the Sun Ultrasparc had more even results (probably because it had the greatest number of registers), and the HP Opteron 250 and the HP Core 2 Duo performed similarly.

4 Unexpected Results

When determining floating point multiplication ILP experimentally, the computers exhibited strange behavior.

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, the Dell Xeon, 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.

Also, it was predicted that floating point addition would have more ILP than multiplication; however, our experimentally determined values contradicted this.

Finally, our graphs show that a serial schedule with a rotating register mapping is MORE effective in small tile heights than the other two techniques (serial and wavefront schedule, with ping pong register mapping).

5 Conclusions

We encountered some strange performance issues when determining the ILP for floating point multiplication. 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.

We also encountered problems when attempting to determine ILP for addition (instead of multiplication). We expected for each computer to be able to have more ILP for addition than for multiplication. However, this was not the case, but due to time constraints, we did not investigate further.

Another consideration of this experiment is that when our program was compiled with optimization flag -O5 in gcc, the larger register tiles had unfeasibly large compile times. For example, on the HP Core 2 Duo, the largest register tile (21x20) took over 27 hours to compile for a serial schedule; 20x20 took 9 hours to compile. Both the ping pong and rotating storage mappings took similar compile times, but the wavefront schedule took slightly more time than the serial schedule. All compile times for tile widths less than 16 with tile heights less than 25 were under 10 minutes, so we collected data from these tile sizes.

The graph indicates that the baseline time, for a program without register tiling (loop unrolling and scalar replacement), is much worse than the times for programs with register tiles, supporting our use of register tiles to optimize code. The graph also indicates that the rotating storage mapping does better than the ping pong storage mapping for small tile heights for either schedule. Due to time constraints, we did not investigate further, but would like to explore this in future experiments.

The goal of this work was to determine whether a computer with register renaming. Our analysis of the data gathered from these experiments indicates that for the HP Core 2 Duo, register renaming likely does compensate for the lack of registers in the initial storage mapping, as the serial rotating storage mapping performed similarly to the serial ping pong storage mapping. To further confirm these results with future experiments, we would want to compare these results with a computer system with similar attributes to the HP Core 2 Duo, but without register renaming.

References

[1] Register renaming. http://en.wikipedia.org/wiki/Register_renaming, July 17, 2007.

[2] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 24–33, San Jose, California, October 3–7, 1998.