Effects of Re-ordered Memory Operations on Parallelism

Richard H. Littin, John G. Cleary
Dept. of Computer Science, University of Waikato,
Private Bag 3105, Hamilton, New Zealand
{rhl, jcleary}@cs.waikato.ac.nz

Abstract

The performance effect of permitting different memory operations to be re-ordered is examined. The available parallelism is computed using a machine code simulator. A range of possible restrictions on the re-ordering of memory operations is considered: from the purely sequential case where no re-ordering is permitted; to the completely permissive one where memory operations may occur in any order so that the parallelism is restricted only by data dependencies. A general conclusion is drawn that to reliably obtain parallelism beyond 10 instructions per clock will require an ability to re-order all memory instructions. A brief description of a feasible architecture capable of this is given.

Keywords: memory access, parallelism, out-of-order execution.

1 Introduction

Today's computer architectures make use of out-of-order instruction execution to extract parallelism (and performance) out of the code they run. This dynamic execution is achieved through the use of reservation stations [Tom67] and reorder buffers within limited sized instruction windows.

Many studies have been performed [JW89] [SJH89] [BYP+91] [Wal91] [LW92] to evaluate the potential parallelism available in sequential code. In most of these studies the major emphasis has been on the effects that control flow has on parallelism. They have looked at the performance issues associated with different forms of branch prediction and speculation. When register renaming has been examined it has been in a limited context, either with an upper limit on the number of registers or on the size of the instruction execution window. Alias analysis has assumed omniscience, with no indication of how this might be achieved.

A study by Lam and Wilson [LW92] examined the effects on parallelism of allowing varying amounts of control dependence resolution. Memory disambiguation and register renaming were

assumed to be perfect. The results, as expected, showed that as the complexity of the control dependence resolution increases so does the amount of available parallelism. They indicate that a machine with oracular analysis capabilities is required to extract large amounts of parallelism.

An earlier study by Wall [Wal91] showed how different levels of branch prediction, jump prediction, register renaming and alias analysis affect the amount of parallelism that is available. The potential parallelism numbers given were substantially lower than those observed by Lam and Wilson [LW92]. This was due to the limit placed on the number of instructions that could execute per cycle. But the results agree with Lam and Wilson that a near perfect machine is needed to extract large amounts of parallelism.

In typical programs approximately one in seven instructions are branches and one in five are memory accesses [HP90]. This implies that a system that has perfect branch prediction but restricts memory accesses to only one per cycle can only gain a maximum of 5 times speedup through parallelism. This has been supported by the studies of [JW89] and [SJH89].

In this paper, which is in contrast to [LW92], we examine the effects of constraining data flow on system performance. The order of accesses to the memory system are restricted and the resulting amount of potential parallelism is observed.

2 Relaxing Data Flow Constraints

A sequential machine executes all memory accesses in order. A machine which attempts to execute many instructions in parallel must be careful, however, as memory operations to a particular location are constrained in the order in which they can occur. These constraints are often described in terms of the 'hazards' that exist. For example, if logically a write is followed by a read then they must be executed in that order. This is called a Read After Write (RAW) hazard. In contrast two successive reads to a location can be reordered and there is no RAR (Read After Read) hazard.

When building a parallel computer there is a trade off between increased parallelism and the extra complexity of hardware needed to ensure that hazards such as RAW are not violated. Such machines can be classified by the sets of re-ordering that are possible. For example, if reads can be re-ordered then this is said to be an RPR machine (Reads can Pass Reads). Similarly if (whenever it is safe) a later write can be interchanged with an earlier read then the machine is said to be an WPR (Writes can Pass Reads) machine. WPW (Writes Pass Writes) and RPW (Reads Pass Write) round out the possible re-orderings.

In the following work we examine how much parallelism is available for 9 different combinations of these possibilities. We consider the machine where no re-ordering of memory operations is permitted, while the rest of the machines all allow RPR and are formed from all eight combinations of the possibilities RPW, WPR and WPW.

Clearly if memory operations are to different addresses then the instructions are not constrained by any hazards. There are two possible ways of checking this for memory writes. One is to wait until both the address and the data value are known and then to do the safety check, the other is to check as soon as the address is known, before the data is available. In many cases addresses are static or computed from loop indices and may be known much earlier than the actual value.

The hope is that by doing the check early more re-ordering will be possible and more parallelism available. In the results later we consider both early and late checking of addresses.

A number of different mechanisms that allow re-ordering of operations including those to memory have been proposed, including *Reservation Stations* [Tom67]. A study in [INT95] proposes a *Memory Order Buffer* and concludes that for the Pentium Pro architecture:

- Writes can be constrained from passing other writes, for only a small impact on performance.
- Writes can be constrained from passing reads, for an inconsequential performance loss.
- Constraining reads from passing other reads or writes has a significant impact on performance.

In other words they propose an RPR–RPW machine as a good compromise between complexity and available parallelism.

Our results below support these conclusions but only at the low levels of parallelism usable in current architectures. The main conclusion we draw is that at higher levels of parallelism a more relaxed memory model is essential, to the point that we advocate all memory operations being able to pass each other.

3 Abstract Machine Models

To examine the performance properties of each of the reordering types we define a set of abstract machine models and analyze the potential parallelism that is available for each machine. The machines are formed by taking combinations of the memory ordering relaxation types. They each assume infinite resources and unlimited memory bandwidth. The abstract machines we have used in our experiments are summarized in Table 1.

Machine	*Description*
NONE	All memory accesses happen in their virtual order.
RPR	Reads can pass reads.
RPR–WPW	Reads can pass reads and writes can pass writes.
RPR–WPR	Reads and writes can pass reads.
RPR–WPR–WPW	Reads and writes can pass reads and writes can also pass writes.
RPR–RPW	Reads can pass reads and writes.
RPR–RPW–WPW	Reads can pass reads and writes, and writes can pass writes.
RPR–RPW–WPR	Reads can pass reads and writes, and writes can pass reads.
ALL	Memory accesses can happen in any order.

Table 1: Abstract machine models used.

At the extremes there are two machines, one with all ordering constraints in place and the other with all ordering constraints relaxed. The totally constrained machine is included to give a base line. The fully relaxed machine is only constrained by the true data dependencies of the program and closely approximates the data disambiguation capabilities of an *Oracle* machine [LW92]. Current

(1997) architectures are approximated by the RPR-WPR machine with a limited sized instruction window.

4 Experimental Framework

This section describes the methodology used to analyze the parallelism for each machine model. Since we are only interested in examining the different levels of data flow constraint we have to provide an oracular control flow model.

4.1 Control Structure

In a perfect control model sequential code is broken down into basic blocks which all start executing at the same time. The original *virtual* order of these basic blocks needs to be maintained so that correct data flow decisions are made. Any reads from memory are blocked until the latest prior write to that location has been completed. This ensures that the data being manipulated is in the appropriate state.

Our control flow model is not perfect, but it is realistic. The program basic blocks are mapped onto a control flow tree. The program's virtual ordering is maintained by traversing the tree top-down and left-to-right (a pre-order traversal). But the nodes in the tree are executed as they are created, in a top-down manner.

This control flow model falls in the range of the *SP-CD* and *Oracle* models of [LW92] and between the *Great* and *Perfect* models of [Wal91]. In either case it is near the top end of the models that they both examined. The control model is based on an optimistic parallel architecture that we are designing called the WarpEngine [CPK95].

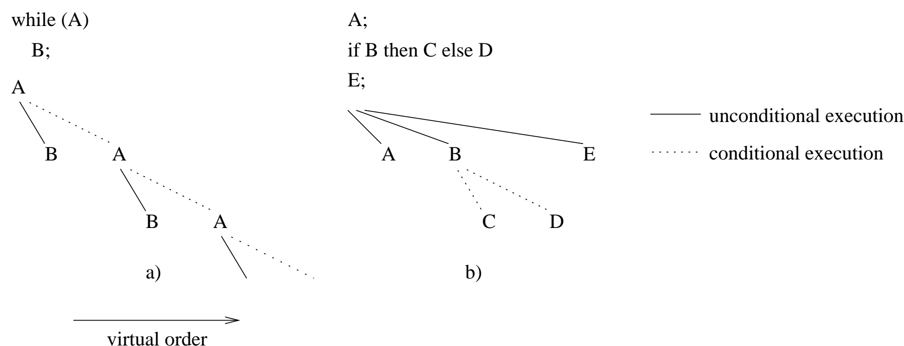

Figure 1: Control trees for simple code structures; a) a while loop, and b) a conditional statement surrounded by unconditionally executed statements.

Figure 1 a) shows how the basic blocks within a loop (an iteration and conditional code) are transformed into the tree control structure. The dotted lines indicate conditional node creation, where the dependent node is not evaluated until the calculation of the conditional test has been completed. The virtual ordering of basic blocks is maintained by moving over the tree from left

to right. Figure 1 b) shows how control flow convergence can be exploited. Basic blocks A, B and E are guaranteed to execute, whereas C and D are dependent on the result of B. If A, B and E are largely independent, starting them early increases the parallelism.

It is possible to use speculation on branch points within this framework. For example, C or D (or both) can be executed before the result of B is known. Such branch speculation is not used in this study. However, *convergence* enables blocks such as E to execute in parallel.

4.2 Program Transformations

Each basic block of the program is mapped onto a node in the control tree. Data flows in two ways: either through the memory system (when the sources and destinations cannot be determined statically); or by direct transfer between parent and child blocks or near relatives (this corresponds to transfers that would be done via registers in more conventional architectures).

The tree structured control mechanism allows loops to be parallelized efficiently. A naive approach, illustrated in Figure 1 a), is to have a node which conditionally branches a loop iteration node and the next condition node. This produces a linked list of loop iterations. Figure 2 shows more parallel methods of mapping loops onto the control tree. Figure 2 a) shows a *for* loop, where the bounds are known statically. A balanced binary tree can be generated. Unbounded loops are mapped to a structure as in Figure 2 b). Dotted arcs point to trees that are scheduled conditionally. The nodes marked B form a linear backbone which starts exponentially increasing numbers of loop iterations. This balances the overhead of executing loops which contain only a small number of iterations (which is the common case [HP96]) against the cost of executing loops with a large number of iterations.

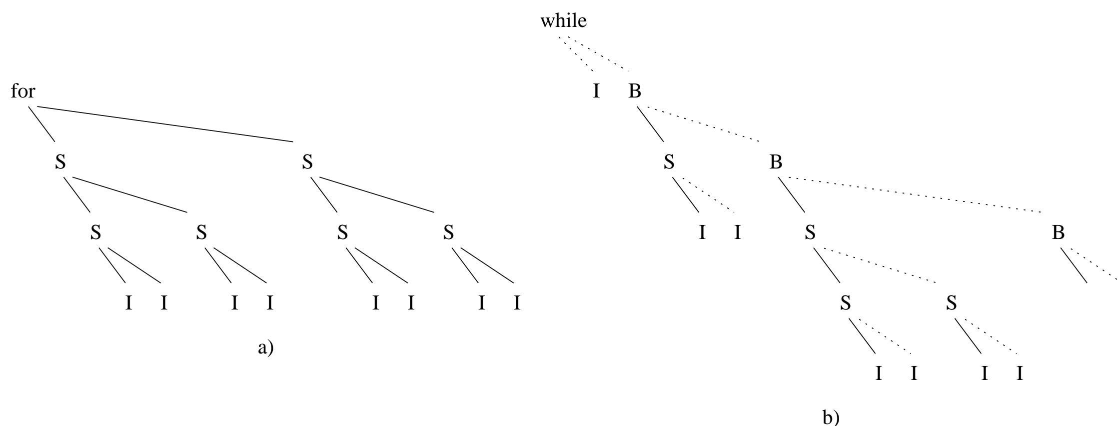

Figure 2: Loop control structures, a) bounded for loop, and b) unbounded while loop.

4.3 Programs

The programs that we have simulated span the types of operations that are performed in many programs, including; sorting, dynamic structure manipulation, matrix/array operations and recursion.

The algorithms are simple in concept but vary in the relative amounts of data and control dependence.

Two tree insertion routines are examined. In both cases a sequence of the specified number of insertions is done. The values to be inserted were generated randomly. The parallelism that is extracted comes mainly from the fact that the insertions can (to a certain extent) be run in parallel.

The first of the two is a naive binary tree insertion (bin). That is, no explicit attempt is made to balance the tree so that the worst case insertion time is $\Theta(N)$. However, because the values inserted were random the expected tree depth (and insertion time) is $\Theta(log_2 N)$ [LD91]. This algorithm does not parallelise well on architectures that use branch speculation, as all possible paths through the tree need to be speculated for each insertion. The parallelism obtained by optimistic execution relies upon code convergence between the different insertions. (Explicit shared memory code that locks only the leaves of the tree may achieve parallelism similar to optimistic systems.)

The second tree insertion program uses the AVL algorithm (avl) [LD91]. This is guaranteed to have (amortized) $\Theta(log_2 N)$ insertion time. Again an optimistic system can achieve parallelism via code re-convergence, while speculative execution cannot. Because of the possibility of rotations of internal nodes an explicit shared memory system needs to lock all the nodes from the root to the leaf during insertion. This serializes the computation removing most opportunities for parallelism.

The following programs sort an array of the specified length. HeapSort [Pre92] and QuickSort [Qui87] are both efficient sorting algorithms (on a sequential processor), but differ in the amount of dynamic parallelism that is obtainable. HeapSort (heap) repeatedly modifies a root element which serializes computation through that point. QuickSort, on the other hand, subdivides the problem into smaller problems which are independent of each other.

We have implemented two forms of QuickSort (qu1 & qu2) that differ in the manner in which they select their pivot elements. The first selects a pivot value and moves from left to right through the array swapping elements until all elements less than the pivot are to the left of it. The second version selects the pivot value and then moves in towards the centre swapping from either end of the array until the pivot is in the correct position. Both versions have a critical execution path of approximately $\Theta(N)$ giving potential speedup of $\Theta(log_2 N)$. It is not entirely clear why qu1 does significantly better than qu2 in the later results. Although, it is note-worthy that the sub-sorts after the partition step can start earlier for qu1 because the left sub-sort can start after 1 partition step and the right sub-sort after $N/2$ partition steps. In qu2 the sub-sorts cannot effectively start until the entire partition is complete in N steps.

Matrix Multiplication (mat) is coded to multiply two $N \times N$ matrices. The inner loop, is linear, and because of the data dependency through the accumulator variable has a critical path of $\Theta(N)$. The result is total parallelism of $\Theta(N^2)$.

Gauss-Jordan elimination (gj) [Pre92] is a version of matrix inversion of an $N \times N$ matrix that makes a careful choice of which row to use as a pivot at each step: which requires a loop of $\Theta(N)$. As a consequence the overall critical path is $\Theta(N^2)$ and the parallelism $\Theta(N)$.

Transitive closure (trans) [CLR90] is another array based routine that has a combination of obvious parallel and sequential operations. On a directed graph with N vertices the sequential running time is $\Theta(N^3)$. The inner two loops are independent and the outer loop has a critical path $\Theta(N)$, giving parallelism $\Theta(N^2)$.

4.4 The Simulator

Each algorithm is hand coded from C to an assembly level language. Evaluation of the potential parallelism within an algorithm is achieved by taking the machine executable and running it on a functional level simulator of the WarpEngine [CPK95]. This simulator performs a pre-order traversal of the control tree executing the code at each node. The program is run in its virtual order and information about the real time that events occur is retained and used to determine the parallel execution time. The reported measure in each case is the potential parallelism–total time for the execution of all instructions divided by the time the last instruction completes.

The simulator assumes that an infinite number of CPUs are available and that there are no constraints from finite bandwidth to memory or cache systems. Because of this the results have no absolute significance but do provide relative information about the different machines. Also the numbers show what is necessary to achieve a given level of speedup on a real machine. That is, to achieve a real speedup of 100 the potential parallelism must be greater than 100 and any techniques necessary to reach that level of potential parallelism must be provided in the real machine.

In previous studies [LW92] [Wal91] all instruction cycles were set to 1 cycle. This gives a measure of the potential instruction level parallelism, but it is not realistic. In those studies execution traces were examined to determine the causal links through out the programs execution and setting all instruction times the same simplifies that task. Also, many assumptions were made about memory disambiguation and register aliasing along with the removal of loop overheads [LW92]. In the results below we use both 1 cycle instructions and a more realistic mix of instruction times. Because we are doing a functional level simulation we accurately compute the effects that control mechanisms have on the amount of parallelism that can be obtained.

5 Evaluation and Analysis

Table 2 shows the result of one set of experiments. In this case address disambiguation was done only when both the address and the value to be written were available and the instruction times were set to a mix of "realistic" values.

The results support the current wisdom that for parallelism up to 10 or so the RPR-RPW machine is a good choice. It provides parallelism of greater than 10 for all but two of the programs (heap and qu2). However, to consistently move into the region of parallelism of more than 15 clearly requires more. The RPR-RPW machine gives parallelism of 90 or more for 3 of the 8 algorithms, but the other 5 languish between 2 and 15. Adding WPR capability to give the RPR-RPW-WPR machine gives no noticeable improvement. Adding WPW to give RPR-RPW-WPW does improve the three least parallel programs by factors of between 2 and 100. The final step to the ALL machine improves two programs: heap from 9.8 to 19.9 and qu1 from 18 to 94. The result is that the ALL machine has only one program below 19 (qu2 at 5.45) and 6 of the 8 programs have parallelism of 90 or more.

The general conclusion is that to achieve reliable speedups in the region from 20 to 100 it will be necessary to use an ALL architecture. While some of the simpler programs achieved good speedups on simpler architectures the more complex codes such as heap and qu1 improve significantly with

Algo-rithm	Prob. Size	Machine								
		NONE	RPR	RPR WPR	RPR WPW	RPR WPR WPW	RPR RPW	RPR RPW WPR	RPR RPW WPW	ALL
avl	2000	2.39	2.40	2.40	2.44	2.44	2.61	2.61	118.59	118.59
bin	2000	1.91	1.92	1.92	1.92	1.92	90.40	90.40	90.40	90.40
mat	50x50	4.89	4.89	4.89	4.94	8192.4	12.55	12.55	12712	12712
gj	25x25	4.56	5.53	6.04	5.53	9.69	140.02	140.02	386.13	393.31
heap	2000	2.85	2.96	3.22	2.99	3.74	2.99	3.25	9.79	19.89
qu1	2000	6.17	6.32	6.32	6.32	27.00	14.46	14.46	17.93	93.96
qu2	2000	1.94	2.09	2.09	2.09	2.12	2.12	2.12	5.45	5.45
trans	30	5.49	7.41	7.41	7.66	409.39	262.80	262.80	5976.9	5976.9

Table 2: Potential parallelism without early address knowledge using typical instruction timings.

each step in capability. We expect this phenomenon to be even more marked for more realistic programs with less regular data and control dependencies.

5.1 Early Address Knowledge

Table 3 shows the potential parallelism when early address knowledge is used (again a "realistic" mix of instruction timings is used). Recall that in this case address disambiguation for writes is done as soon as the address is available rather than waiting for the data value as well. This will not affect any results for the ALL machine because it does not wait for disambiguation before allowing passing. In a similar vein it would be expected that the greatest effect might be on the more restrictive machines. Also programs where the addressing patterns are regular and addresses are determined on the basis of index values in loops should be improved the most. The results bear this out, two programs, mat and trans, show improvements of up to a factor of 700 on the more restrictive machines. Apart from this gj shows an improvement of about 30% on all machines except the most liberal. All the other programs show no sensible improvement.

Clearly early address disambiguation can be very important for very regular computations, although it is worth noting that they already had good parallelism and it is likely that on a real machine other factors would limit the actual speedup. Consider also the most interesting architectures: on RPR-RPW only mat was improved by a factor of more than 700 (from 12 to 8700), gj was improved by 30% and trans by less than 100%; on RPR-RPW-WPW no program improved by more than 5%; and of course on ALL there was no change. The overall conclusion is that unless you are building a machine to do matrix operations then early address disambiguation is not important.

5.2 Single Cycle Instructions

In the studies by [LW92] and [Wal91] all instructions are set to take 1 cycle. To test the sensitivity of our results to instruction timings we repeated the experiments of Table 3 with all instructions set

Algo-rithm	Prob. Size	Machine								
		NONE	RPR	RPR WPR	RPR WPW	RPR WPR WPW	RPR RPW	RPR RPW WPR	RPR RPW WPW	ALL
avl	2000	2.44	2.46	2.46	2.48	2.48	2.63	2.63	118.59	118.59
bin	2000	1.92	1.93	1.93	1.93	1.93	90.40	90.40	90.40	90.40
mat	50x50	10.54	10.54	520.03	10.75	12712	8764.1	8770.9	12712	12712
gj	25x25	5.74	7.38	8.31	7.39	13.39	173.08	173.08	386.53	393.31
heap	2000	2.98	3.07	3.33	3.07	3.81	3.07	3.33	10.18	19.89
qu1	2000	6.40	6.55	6.55	6.74	28.14	14.55	14.55	18.16	93.96
qu2	2000	2.02	2.11	2.11	2.11	2.13	2.13	2.13	5.45	5.45
trans	30	5.70	11.40	11.40	11.79	4309.7	446.25	446.25	5976.9	5976.9

Table 3: Potential parallelism with early address knowledge and typical instruction timings.

to 1 cycle. The results in Table 4 are all within 30% of the values in Table 3 and none of the earlier qualitative conclusions are altered.

Algo-rithm	Prob. Size	Machine								
		NONE	RPR	RPR WPR	RPR WPW	RPR WPR WPW	RPR RPW	RPR RPW WPR	RPR RPW WPW	ALL
avl	2000	2.55	2.58	2.58	2.63	2.63	2.76	2.76	106.33	106.33
bin	2000	2.10	2.11	2.11	2.11	2.11	56.33	56.33	56.33	56.33
mat	50x50	7.02	7.02	349.98	7.16	13817	7830.6	7882.3	13817	13817
gj	25x25	4.73	6.40	6.84	6.41	10.84	143.32	143.32	368.58	372.34
heap	2000	3.14	3.25	3.54	3.25	4.00	3.25	3.54	10.61	20.45
qu1	2000	6.09	6.26	6.26	6.45	26.99	11.96	11.96	14.59	66.22
qu2	2000	1.96	2.10	2.10	2.10	2.12	2.13	2.13	5.38	5.38
trans	30	3.85	7.71	7.71	7.97	3755.3	314.63	314.63	6808.7	6905.5

Table 4: Potential parallelism with early address knowledge and all instruction times set to 1 cycle.

5.3 Increasing Problem Size

The potential parallelism for these different programs can be expected to vary (probably increase) with increasing problem size. Figures 3 through 6 show the potential parallelism for a number of the programs for each of the machines. Typical instruction times and early address disambiguation are used in each case. The results are striking. In each case the relative performance of the machines is maintained across a wide range of problem sizes. It seems that the access patterns of the different programs remain unchanged as the sizes are varied. Similar results were obtained for all the other programs.

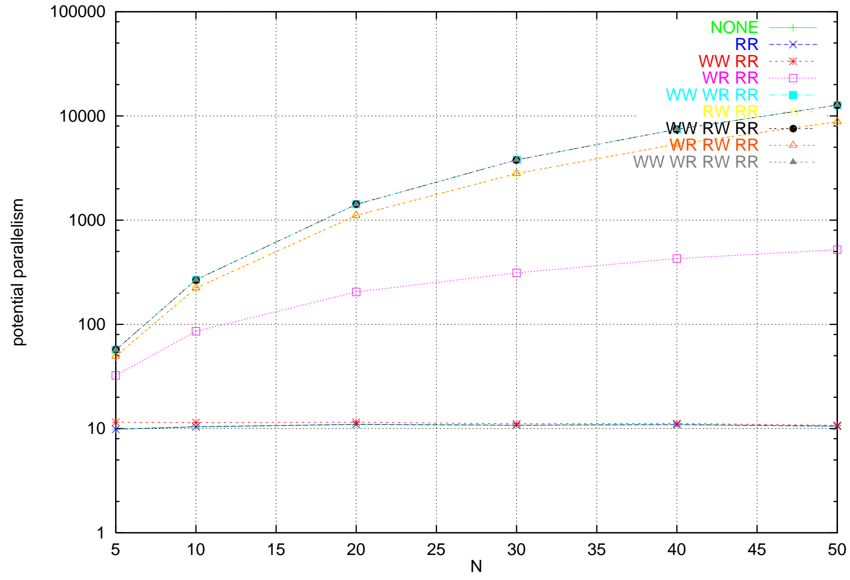

Figure 3: $N \times N$ matrix multiplication (mat) potential parallelism vs problem size.

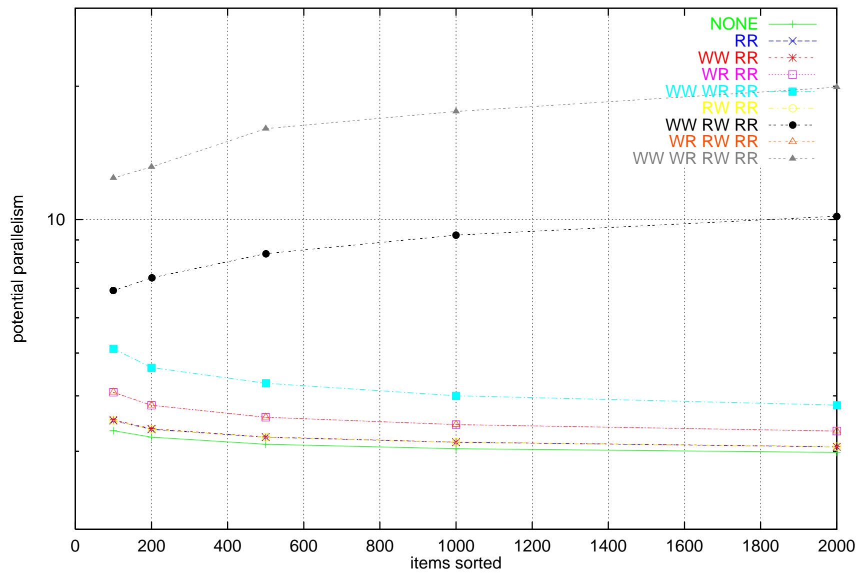

Figure 4: HeapSort (heap) potential parallelism vs problem size.

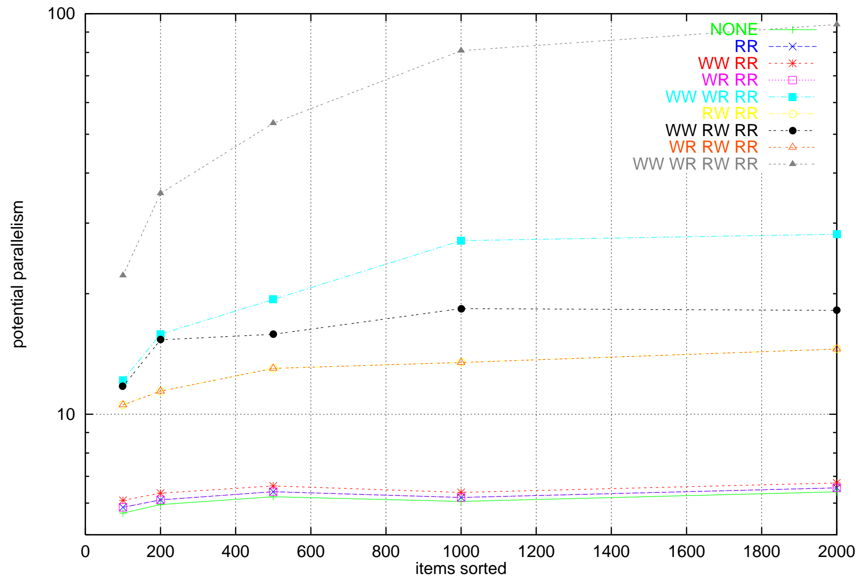

Figure 5: QuickSort (qu1) potential parallelism vs problem size.

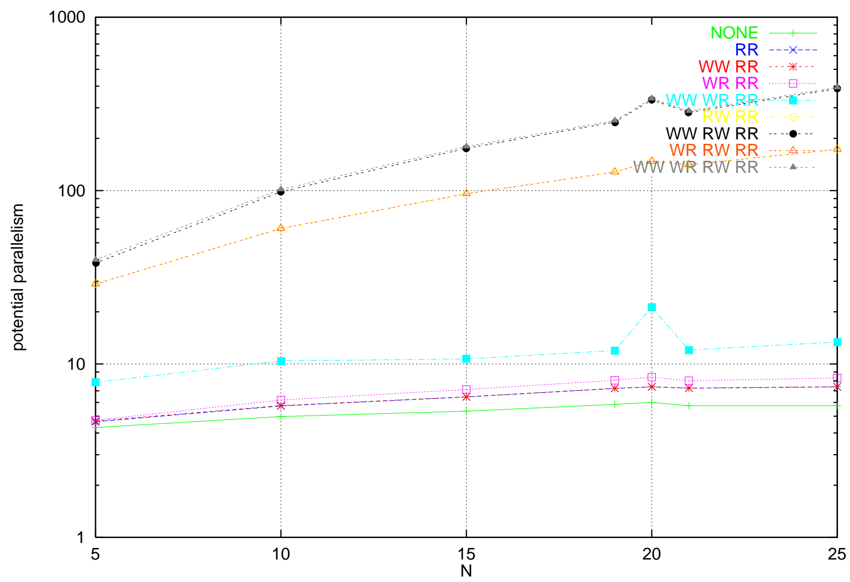

Figure 6: Gauss-Jordan elimination (gj) potential parallelism vs problem size.

6 Conclusions

The main conclusion is that if ILP parallelism is to be pushed significantly beyond currently attainable values of 2 to 4, especially beyond 10, then it is necessary that the memory system support all possible out-of-order executions of memory operations. Some regular programs require less aggressive architectures, but more complex (and possibly realistic) programs clearly need the greater freedom of full reordering. These conclusions are based on a small set of simple programs and clearly it is necessary to verify the results in more complex programs. We are currently working on a compiler for the WarpEngine and we will extend these results to more standard benchmarks when it is available.

To extract parallelism, memory disambiguation mechanisms need to be in place that allow accesses to pass each other. For example, if two writes to a given address occur out of order then it is necessary to remember both values. The false causal links of dependent read requests can then be removed, allowing them to be satisfied as early as possible. The WarpEngine [CPK95] achieves this through the use of a timestamped memory system.

Our simulation results assumed infinite resources constrained only by control and data dependencies and so the reported parallelism values are an upper bound on what can be achieved. Thus, any architecture which is going to achieve these levels of parallelism must be sufficiently powerful to allow all memory operations to be performed out of order. It will be interesting to see if it is possible to build such a system.

References

[ACM89] ACM. *3rd International Conference on ASPLOS*, New York, N.Y., 1989.

[BYP+91] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *18th Annual International Symposium On Computer Architecture*, pages 276–286, New York, N.Y., 1991. ACM.

[CLR90] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, New York, 1990.

[CPK95] John G. Cleary, Murray W. Pearson, and Husam Kinawi. The architecture of an optimistic CPU: The WarpEngine. In *Proceedings of HICSS*, volume 1, pages 163–172, Hawaii, 1995.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, 1990.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, second edition, 1996.

[INT95] INTEL. PENTIUM PRO processor at 150 mhz. INTEL Corporation Datasheets, October 1995. Order Number: 242769-001.

[JW89] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *3rd International Conference on ASPLOS* [ACM89], pages 272–282.

[LD91] H. R. Lewis and L. Denenberg. *Data Structures & Their Algorithms*. Harper Collins, 1991.

[LW92] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium On Computer Architecture (ISCA-19)*, pages 46–57, New York, N.Y., 1992. ACM.

[Pre92] William H. Press. *Numerical Recipes in C*. Cambridge University Press, 1992.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw–Hill, 1987.

[SJH89] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. In *3rd International Conference on ASPLOS* [ACM89], pages 290–302.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, January 1967.

[Wal91] Daivd W. Wall. Limits of instruction-level parallelism. In *4th International Conference on ASPLOS*, pages 176–188, New York, N.Y., 1991. ACM.