Working Paper Series
ISSN 1170-487X

Constraints on Parallelism Beyond 10 Instructions Per Cycle

by John G Cleary, Richard H Littin, J A David McWha & Murray W Pearson

Working Paper 97/27
November 1997

© 1997 John G Cleary, Richard H Littin,
J A David McWha & Murray W Pearson
Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Constraints on Parallelism Beyond 10 Instructions Per Cycle

John G. Cleary, Richard H. Littin, J. A. David McWha and Murray W. Pearson
Dept. of Computer Science, University of Waikato,
Private Bag 3105, Hamilton, New Zealand
{jcleary, rhl, jadm, mpearson}@cs.waikato.ac.nz

Abstract

The problem of extracting Instruction Level Parallelism at levels of 10 instructions per clock and higher is considered. Two different architectures which use speculation on memory accesses to achieve this level of performance are reviewed. It is pointed out that while this form of speculation gives high potential parallelism it is necessary to retain execution state so that incorrect speculation can be detected and subsequently squashed. Simulation results show that the space to store such state is a critical resource in obtaining good speedup. To make good use of the space it is essential that state be stored efficiently and that it be retired as soon as possible. A number of techniques for extracting the best usage from the available state storage are introduced.

Keywords: instruction level parallelism, speculation

1 Introduction

Increasingly computer architects and system designers are seeking to extract more computer performance by making use of parallelism. There are a number of ways of approaching this. For example processes which have disjoint address spaces can be run independently. For single integrated programs it is possible to explicitly decompose them into threads that communicate via messages or via explicitly synchronised shared memory. Other possibilities include the use of different programming paradigms such as functional or logic programming where a compiler can readily generate code to run on message passing or shared memory machines. This paper considers the problem of extracting Instruction Level Parallelism (ILP), that is, parallelism from conventional imperative programs which have not been explicitly modified to extract parallelism.

The drive for ILP comes from a number of directions. Firstly, the great majority of the world's code is written in imperative languages and it seems almost inconceivable that it could all be rewritten to explicitly take advantage of parallelism. Secondly, it is now difficult to take advantage of and keep busy the number of transistors available on modern CPU chips without making some use of

parallelism. Thirdly, at some point in the next decade or two Moore's law will fail and the steady exponential increase in performance of silicon based computers will cease. Parallelism will then be needed to maintain performance increases.

Current production architectures are limited in the amount of parallelism they extract because of the difficulty in identifying code which can be executed in parallel. This tends to restrict the parallelism extracted to less than a factor of ten [LW92][SBV95]. In order to exceed this figure a large number of instructions which can be executed in parallel must be identified. One method which has been proposed to identify and extract more parallelism is speculative execution.

Speculation allows instructions to be executed even when it is not certain that they should be executed or that they are executing with the correct data. This means that, sometimes, incorrect execution may take place, which must then be undone and re-executed correctly. Speculation thus increases the pool of instructions which can be executed concurrently, allowing more hardware to be effectively used.

Speculation is usually considered in the context of speculation on branch instructions. However, this paper will consider a different class of speculation. To make the terminology more precise we will refer to three types of speculation: branch speculation, value speculation, and memory speculation.

In branch speculation a prediction is made as to the direction a branch will take and the CPU follows this direction. It has been shown that these mechanisms can achieve up to 93% prediction accuracy [PS93]. However, after only a few branches the probability that the correct path is being followed is too low to be useful. A more aggressive approach is to follow both paths through a branch but again this can achieve only a small amount of parallelism because of the exponential explosion in the number of possible paths.

In value speculation guesses are made as to the value that will be returned by a read and this value is used before the true read has time to complete. It has been shown in [LS96] that value speculation would give performance gains of 4.5%–23% if it were incorporated into a PowerPC 620. This form of speculation is still theoretical and has not been included in any delivered architecture.

In memory speculation values are returned early from reads by making use of local or current information. For example, a value might be present in a local cache but it is unclear that it is coherent or safe. That is, there might be a remote instruction which has already written to the location but the value has not yet arrived or there may be an earlier instruction, which has not yet completed execution, which will write to the location. As in value speculation the system will continue executing with the possibly erroneous value until a new (possibly correct) value arrives.

For speculation to be effective it must still have a sufficient number of instructions to operate on. One way to achieve this is by using the fact that some blocks within a program are guaranteed to execute no matter what combination of branches precede them. That is, the execution stream is guaranteed to converge at a later point. None of today's production architectures make use of code convergence, but there is ongoing research into this area.

There is a strong synergy between the different speculative techniques and code convergence, both because they can all add to the potential parallelism and, as we will see, they can use the same mechanisms for detecting incorrect speculation and undoing it. This paper focuses on architectures

that use both code convergence and memory speculation and examines the limits imposed by the resources consumed in supporting speculation. Two such architectures are the Multiscalar Machine [SBV95] and the WarpEngine [CPK95]. Both use convergence information but vary in the amount of speculation they do and their mechanisms for undoing incorrect execution.

The Multiscalar Machine [SBV95] has multiple processors (typically four) and uses a conventional MIPS instruction set with a few extra instructions. These extra instructions enable code to be broken up into blocks of contiguous instructions, with several blocks executing in parallel on essentially independent sequential CPUs. These blocks, known as *tasks*, may be as large as a whole program or as as small as an individual instruction. Each task is treated as an independent program. Branches to locations outside the block determine which block should follow the currently executing one. Between the blocks there may be both control and data dependencies which are maintained through *task squashing* and an Address Resolution Buffer (ARB), respectively. If a location has a new value written to it by a sequentially earlier store operation then the task the load was executed by is squashed and re-executed.

The WarpEngine [CPK95] organises its instructions into blocks with a fixed maximum size (currently 16). Within each block instructions are executed using dataflow parallelism. An active incarnation of a block is called a *frame* and contains its own set of registers holding local temporary values. To maximise code convergence the WarpEngine generates code in the form of a tree structure. Each block can conditionally execute blocks which form its children in the tree. Thus it avoids explicit branch instructions completely. [CMP97] gives examples of how to generate various loop and control patterns given this style of control. In contrast with the Multiscalar Machine the WarpEngine is able to *rollback* , that is, undo and re-execute individual instructions rather than entire tasks. This is intended to reduce the overhead of incorrect speculation and subsequent re-execution.

To achieve speed-up in such highly parallel architectures a number of criteria must be met. Firstly, the control and code decomposition must be able to generate a large pool of instructions for parallel execution. Secondly, the overheads of the control and synchronisation must be kept low enough not to vitiate the gains from parallel execution. It is the thesis of this paper that there is a third class of constraints, those imposed by the resources needed to maintain the information needed for speculation. The next section describes in more detail the information that is needed to support speculation and how this is done in the WarpEngine and the Multiscalar Machine. Section 3 gives simulation results on a number of programs where the resources for speculation are limited and shows that they can become a critical bottleneck for performance. Section 4 describes a number of techniques that might be used to improve the usage of such resources. The final section gives some general conclusions and comments on future work.

2 Resource Usage for Speculation

Speculative systems need to be able to undo computation when it is discovered that a computation path has been taken incorrectly. In the context of memory or value speculation this means that an undo should occur when any write occurs out-of-order. Recovery from incorrect execution can be

broken into four stages:

- undoing the results of the original computation, that is, deleting the memory writes (and any consequences of this for later computation);
- restarting the computation at the point of the causality violation, that is, restarting execution at the point of the out-of-order read;
- recomputing the results from that point onward, that is, re-executing instructions dependent on the read;
- regenerating the outputs (the writes) from this execution, including any consequence of this for later computation.

In order to be able to perform these functions it is necessary to retain sufficient state from the execution. This state can be broken down roughly in parallel with the four functions above:

- There needs to be state retained to detect causality violations. For memory speculation this means retaining the addresses of all reads.
- It is necessary to record the consequences of the initial execution so that they can be undone if a speculative path is incorrect. In the case of memory speculation this means recording the values and addresses of all writes.
- To be able to restart execution it is necessary to retain either a copy of the instructions or their address.
- Varying amounts of state can be retained to facilitate re-execution of code. For example, by keeping intermediate values from the initial computation.

Any speculative system must also be able to delete this state information when it is no longer live. That is, when the speculated execution is known to be safe.

The following two subsections briefly summarise the mechanisms used by the Multiscalar Machine and the WarpEngine to deal with invalid execution and retention of state information.

2.1 Multiscalar Machine

In the Multiscalar Machine detection of an out-of-order write causes any (later) task which executed a corresponding read to be squashed as well as *all following* tasks. Note that the entirety of each task is squashed. Re-execution consists merely of restarting the (earliest) task at its initial address which will then restart any later tasks during normal forward execution.

To detect violation of causality between the reads and writes copies are kept of the addresses of all reads and the addresses and values of all writes in the ARB structure. The ARB also propagates written values to (later) tasks and compares addresses of reads to check for out-of-order writes. As

well the code address is kept along with the state of the registers at the start of each task. This is sufficient to enable the task to be restarted if it is squashed.

The oldest or trailing task is special as it can never be invalidated and so any writes it has done can be written directly to memory (as well as being propagated forward through the ARB). Also it is not necessary to retain the addresses of reads as there are no earlier writes for them to be checked against. When a task becomes the trailing task as a result of its predecessor completing then any accumulated writes can be committed to memory and the accumulated read addresses deleted.

It can be seen that the capacity of the ARB to record reads and writes is finite (as it is an active device on the main CPU chip). Thus if the trailing task and a non-trailing task both run for a long time then the non-trailing task will eventually be blocked and be unable to proceed until the trailing task terminates.

2.2 WarpEngine

The WarpEngine records all reads and writes in a *time-space* cache [PLMC97]. The time-space cache fulfills a function similar to the ARB and retains the addresses of all reads and writes and the values of all writes. The WarpEngine also keeps a copy of every block of code executed in an active *frame*. The frame records all temporary values during execution of the code (essentially retaining all values that would be stored in registers in more conventional architectures). Thus if a read is invalidated by an earlier out-of-order write only those instructions directly affected need be re-executed. Any other values computed by other computation paths are recorded in the frame and are immediately available for re-computation. This mechanism minimises the cost of any invalidation, albeit at the cost of extra storage. As in the Multiscalar Machine the trailing frame can be detected and the frame and its associated reads and writes deleted.

The capacity of the time-space cache also places a constraint on the available parallelism. Consider two long sequences of frames executing in parallel where there are read/write dependencies between the frames within each sequence but not between sequences. The trailing frame in the earlier sequence will be able to be deleted as soon as it has completed execution but both the frames and the read/write information need to be retained in the second sequence. If the two sequences are sufficiently long then eventually the second will be blocked by a lack of space in the time-space cache and the entire machine will be reduced to sequential execution.

2.3 The Problem

The rest of this paper examines the severity of the problem of the finite capacity of the time-space cache and the effect this has on the potential parallelism available in a number of different programs. It is important to realise that the problem is not particular to the WarpEngine, but as we have seen with the Multiscalar Machine it will be critical to any CPU which attempts to scale speculation out to the region beyond 10 instructions per clock.

3 Results

3.1 Assumptions

To test the effects of limiting the space available for the time-space cache and frames a number of programs were simulated for the WarpEngine. In order to isolate the effect of resource limitations, infinite bandwidth to memory and an infinite number of functional units were assumed. During simulated execution a restriction was placed on the total number of frames that could be in use at any one time.

The simulator does not keep track of all possible undo and re-execute operations but instead keeps track for each instruction of: the time at which it was first created; the time at which it last did any computation; and finally the time at which it was retired. In the results below we refer to a frame as *active* when it has been created but has one or more instructions not finished executing and *waiting* if all its instructions have finished execution and it is waiting to be committed. An active frame may not actually be executing any instructions because it is waiting on earlier data values. Also any one instruction will be executed at least once and may be executed many times as earlier speculative reads are resatisfied a number of times.

Resources are allocated in strict time order. That is, if an earlier block needs a frame then it is allocated to it even if a later block might have started prior to it. In a WarpEngine implementation this might be done by pre-empting the later frame and re-allocating it. The time of creation of a frame is taken to be the earliest time that it can be allocated so that it will not be later pre-empted.

Results on resource usage are reported in terms of the number of frames that are active or waiting. The space needed for the time-space cache will be proportional to the number of frames to a good approximation because the number of reads and writes per frame is consistent across a range of applications [HP90]. To simplify the results the usage of frames and locations in the time-space cache are not reported separately.

Results for execution time are reported as potential parallelism compared with a sequential execution. This is calculated by counting the number of machine cycles to execute sequentially, with no overlap between instructions, divided by the number of machine cycles for the parallel execution to complete. Instructions types were allocated different typical numbers of machine cycles to execute. Using this measure a 5 stage pipeline machine which issued 4 instructions per clock (and had no stalls) would achieve a parallelism of 20.

3.2 Programs

As no compiler exists for the WarpEngine the programs were hand coded in WarpEngine assembler. Although they are simple they demonstrate the effects that limited resources have on parallelism for a number of different classes of programs. The programs simulated were: matrix multiplication (mat); AVL binary tree insertion (avl); Gauss-Jordan elimination (gj); naive binary tree insertion (bin); Fibonacci numbers (fib); and quicksort (qs).

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 potential parallelism of $\Theta(N^2)$.

Naive binary tree insertion (bin) inserts N randomly chosen items into a binary tree. No attempt is made to balance the tree so that worst case insertion time is $\Theta(N)$. However, with random data the expected insertion time is $\Theta(log_2 N)$ [LD91].

AVL binary tree insertion [LD91] successively inserts N randomly chosen keys into a binary tree using rotations to maintain balance. It is guaranteed to have (amortized) $\Theta(log_2 N)$ insertion time. The AVL algorithm is difficult for most parallel systems as any part of the tree might be modified on any insertion and so it is hard to extract parallelism statically. Memory speculation can achieve significant parallelism via code re-convergence, while branch speculative execution cannot. An explicit shared memory system needs to lock all the nodes from the root to the leaf during insertion which serializes the computation.

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 $\Theta(N)$. Unlike matrix multiplication, parallelism can not be statically predicted as decisions on which row to pivot are made dynamically.

Quicksort (qs) [Qui87] sorts N randomly chosen items. It has a critical path of $\Theta(N)$ and a potential parallelism of $\Theta(logN)$. The quicksort version used here determines the pivot points working from left to right.

Fibonacci numbers (fib) are computed using the naive recursive algorithm which is exponential in the index of the Fibonacci number.

Table 1 shows the amount of parallelism obtained from each algorithm (for a given problem size) when the number of execution resources (frames) are limited.

		maximum frames													
alg	*size*	*2*	*4*	*8*	*16*	*32*	*64*	*128*	*256*	*512*	*1024*	*2048*	*4096*	*8192*	*16384*	*unlimited*
avl	100	2.5	2.6	2.9	3.1	3.9	5.3	7.2	10.0	14.4	18.5	19.4	19.4	19.4	19.4	19.4
avl	200	2.5	2.6	2.8	3.0	3.7	5.1	6.9	10.0	14.9	21.5	26.6	29.7	29.7	29.7	29.7
avl	500	2.4	2.5	2.7	2.9	3.4	4.9	6.5	9.7	15.3	24.1	36.5	47.3	49.7	49.7	49.7
bin	100	1.9	2.0	2.5	3.5	4.6	7.2	11.2	17.9	24.5	27.2	27.2	27.2	27.2	27.2	27.2
bin	200	1.9	2.0	2.4	3.4	4.5	6.8	10.7	17.6	26.7	37.6	39.3	39.3	39.3	39.3	39.3
bin	500	1.9	2.0	2.3	3.3	4.1	6.2	9.9	16.4	27.4	44.1	58.7	59.9	59.9	59.9	59.9
fib	5	2.5	3.0	3.6	3.9	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3
fib	10	2.6	3.3	4.0	5.3	7.5	10.5	15.0	20.2	31.6	31.6	31.6	31.6	31.6	31.6	31.6
fib	15	2.6	3.3	4.0	5.5	7.8	11.5	18.5	28.3	44.7	72.8	111.2	145.4	235.0	235.0	235.0
gj	5x5	2.8	3.7	4.9	6.7	10.0	14.8	20.4	33.2	39.9	39.9	39.9	39.9	39.9	39.9	39.9
gj	10x10	3.0	4.2	5.7	7.7	9.7	15.4	23.2	39.1	56.1	91.4	101.6	101.6	101.6	101.6	101.6
gj	15x15	3.1	4.3	5.7	7.8	9.9	17.4	24.8	39.4	62.6	95.3	158.1	179.6	179.6	179.6	179.6
mat	10x10	3.2	4.1	5.6	7.2	10.6	15.6	22.8	38.9	62.9	93.2	163.1	237.8	267.4	267.4	267.4
mat	20x20	3.2	4.3	6.0	7.9	9.6	14.6	20.8	31.1	55.9	94.4	161.7	261.6	457.2	818.0	1417.0
mat	30x30	3.2	4.2	6.2	8.0	9.9	12.3	17.2	26.6	46.9	76.7	146.4	259.9	447.0	727.3	3784.2
qs	100	2.9	4.0	5.0	6.5	8.0	9.7	11.5	13.9	16.4	18.5	22.2	22.2	22.2	22.2	22.2
qs	200	2.9	4.0	5.1	6.6	8.0	10.0	12.5	14.9	19.2	22.6	26.0	35.6	35.6	35.6	35.6
qs	500	2.9	4.0	5.2	6.7	8.3	10.5	13.2	16.3	20.0	24.0	31.1	37.6	48.0	53.3	53.3

Table 1: Parallelism obtained when frame resources are limited using typical instruction timings.

3.3 Resource Utilisation

As expected the amount of parallelism increases as more resources are made available, until each program's potential parallelism limit is met. The first point to note is that there is significant par-

allelism available even in the relatively difficult cases such as AVL. The next point to note is that the number of frames is much higher than the available parallelism and the efficiency with which frames are used declines as the number of frames is increased. The inefficiency of frame usage is heightened by noting that most of the fixed size blocks have about 50% of their instructions actually occupied and so each frame represents approximately 8 instructions. So space is being consumed for many more instructions than are actually being executed at any one time. Calculation of the ratio of available space for instructions to the number of instructions executing in parallel gives values of up to 700 for the programs of Table 1 in the cases where the available frames restrict parallelism. Given the assumptions we have made the only constraints on the available parallelism are the actual data dependencies within the program and weak control dependencies imposed by the available code convergence. Similar inefficiencies can thus be expected for any architecture delivering these levels of parallelism independent of its implementation details.

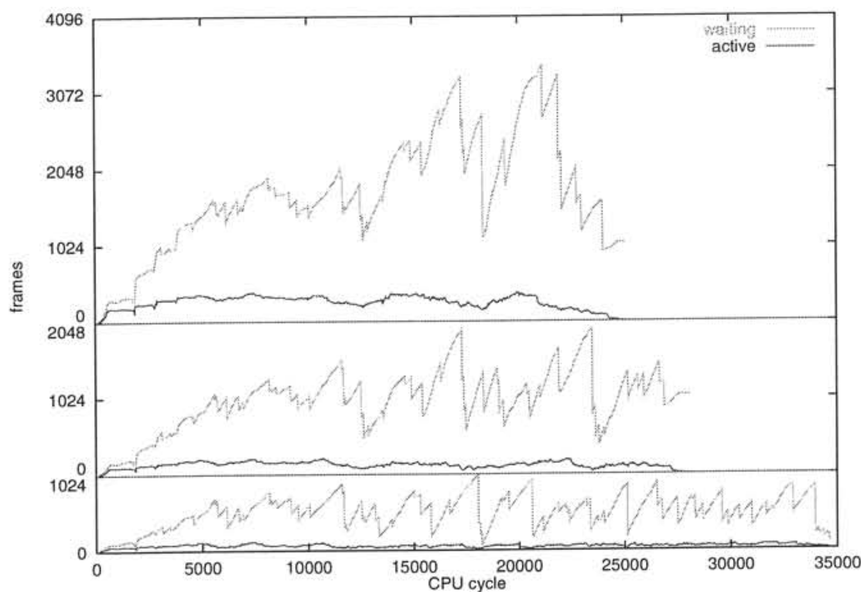

Figure 1: AVL binary tree insertion (200 items).

Figures 1, 2 and 3 show the number of frames used during execution for three of the programs against time. Each graph has three paired plots for simulation runs where the maximum number of frames have been limited to 4096, 2048 and 1024. Each pair of plots shows the number of frames that are actively doing work (the lower plot in each case), and the number of frames that are being utilised (either active or waiting).

These graphs give some insight into why so much space is required to reach these levels of parallelism. The frames can be divided into three classes, those that are unused, those that are waiting and those that are active. In Figures 1 and 2 the upper graph with 4096 available frames is limited by the available parallelism and so most frames are unused. In all the other cases the available frames limit the performance. However, even here the majority of frames are unused. The typical

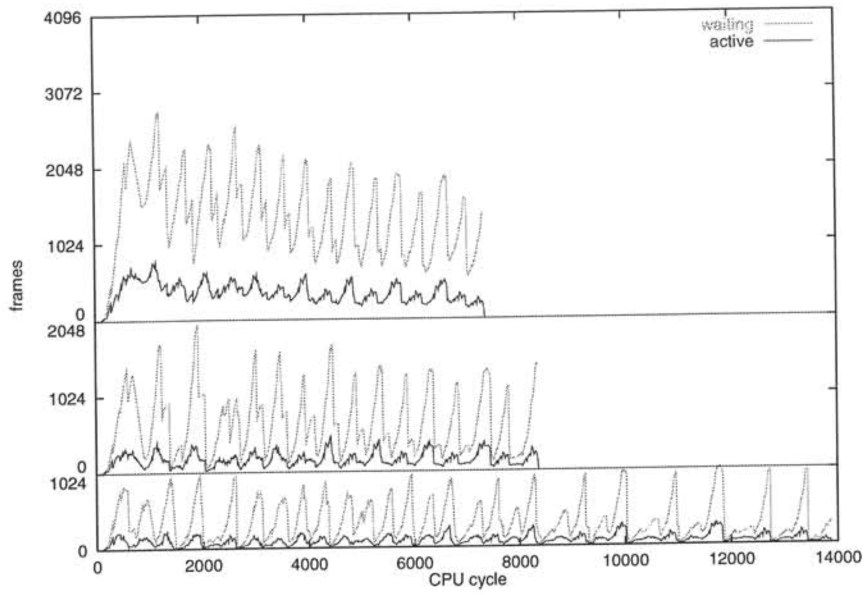

Figure 2: Gauss-Jordan elimination (15x15).

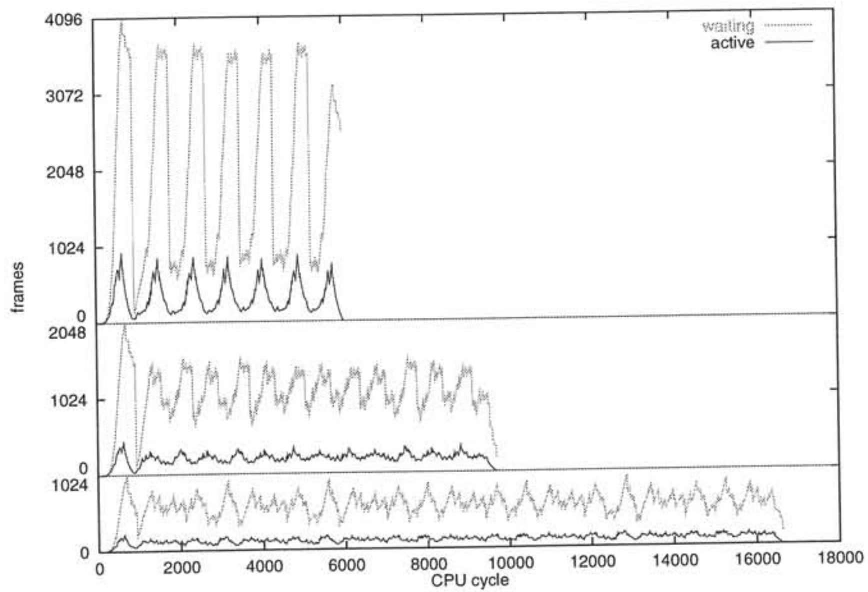

Figure 3: matrix multiplication (20x20).

pattern is that the number of used (active or waiting) frames peaks early in the computation reaching the total number of frames and then rapidly drops. Subsequently the number of used frames only approaches or reaches the upper limit in short peaks. Also the number of active frames is much smaller than the number of waiting frames. That is, most of the life of a frame is spent waiting for some trailing frame to complete so that it can be retired. Similar usage patterns occur for all of the programs simulated.

The strong oscillations are caused by two different mechanisms. The sharp declines in frame usage are caused by one frame completing and releasing a large group of frames that are waiting on it to retire. When this happens the large release of resources causes a subsequent climb in frames as new frames are started. This then soon approaches the upper limit and computation is again blocked until another large group of frames are retired.

This behaviour is exemplified by the simple example discussed earlier where two long independent threads execute in parallel. Eventually the second is blocked by lack of resources until the first completes releasing all the resources in the second thread in one step. Thus even such a simple example would show oscillations and a large ratio of frames to potential parallelism.

To place these results in context it was estimated in [Cal97] that a single frame requires about 11,000 gates and its associated time-space cache entries about 1,700 gates. That is, the chip space for 1,000 frames is roughly equivalent to 230K bytes of cache which is not infeasible for a modern CPU. It thus might be argued that all that needs to be done is to install a sufficiently large number of frames and that the parallelism will follow.

3.4 Variation with Problem Size

Unfortunately a paradoxical situation can arise with some programs where as the problem size increases the amount of parallelism extracted goes down. A number of the entries in Table 1 show this effect. The program where it is most marked is matrix multiply, which can be seen in Figure 4. The problem with matrix multiply is that the sequential inner loop where the sums are being accumulated gets longer as the size of the problem is increased. The result is that the number of frames required for the inner loop grows and reduces the parallelism available between the loops.

4 Improving Resource Usage

We are currently investigating a number of different approaches to improving resource usage. This section briefly outlines these and the extent to which they are expected help.

4.1 Storing Less State

One way to reduce the resource usage is to reduce the amount of state stored. In the results above it was assumed that the operation codes as well as intermediate results were stored together with active logic to detect dynamic changes in the state of the frame. This assumption led to the estimate of 11,000 gates per frame. Instead, just the intermediate results might be stored, this reduces the total

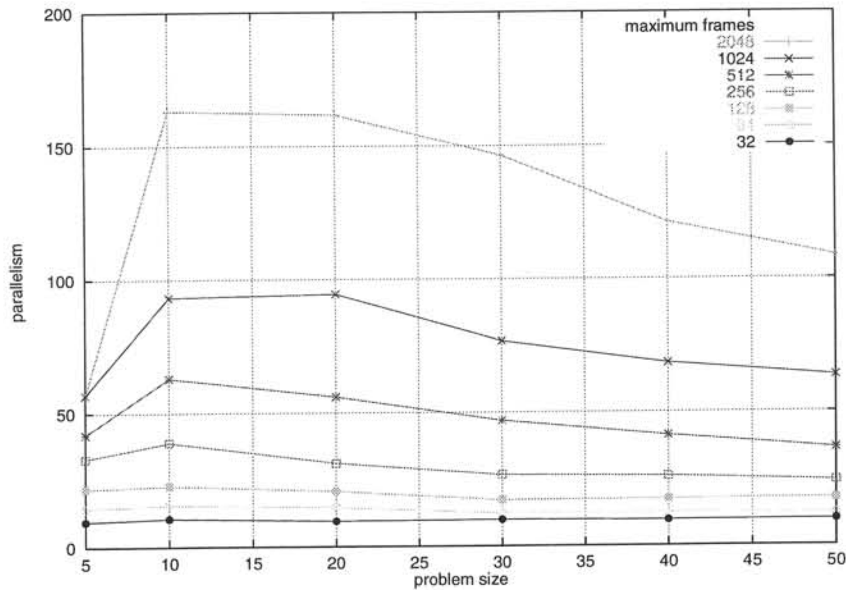

Figure 4: Parallelism vs. problem size with fixed resource limits for matrix multiplication.

gates by about half to 6,100. The cost of this solution is that if an instruction needs to be re-executed then the operation code would need to be refetched and the frame would need to be moved to a full frame location with local logic for detecting the state of the frame. A strategy with some promise is to initially store a frame in its fully expanded form and if it has been inactive for some time then discard part of the information and move it to less active storage. It might also be possible at this point to take advantage of the fact that about 50% of the instructions in a block are unused and record only the instructions that are used (this is difficult when initially executing a frame as to maintain speed it is good to keep frame processing uniform and simple). This would reduce frame size to about 3,000 gates. The effectiveness of this strategy would depend on correctly identifying frames that can be compacted and minimising the number that would need to be recalled and expanded.

A further step in reducing storage usage is to discard all or some of the intermediate results in the frame. In the WarpEngine it is necessary at a minimum to record values transmitted directly from other frames (the WarpEngine has instructions that allow closely related frames to directly transmit data to each other without using the memory subsystem). A typical frame has between one and three such inputs so the total space for a frame might be reduced to about 600 gates (2 words of input data plus an address for the frame plus a few bits of control information). The disadvantage of this approach is that if any instruction in a frame needs to be re-executed the entire frame will need to be re-loaded and re-executed, consuming time in functional units.

As well as the information in the frame itself there is about 1,700 gates per frame in the time-space cache. This is the same irreducible minimum stored in the Multiscalar Machine ARB which is necessary to detect out-of-order memory operations. Even this information might be reduced by recording a single entry for all memory loads within a range of addresses. Any memory store

within that range causes all loads in the range to be re-satisfied. This reduces the number and size of the time-space cache entries, but creates false dependencies which will cause some unnecessary re-execution of frames.

Another means of decreasing the cost, if not the size of the resources is to use cheaper slower memory (such as an off-chip cache or on-chip DRAM) to store the frame information. This might be coupled with the idea of compacting frames as they become older.

4.2 Out-of-order Retirement

A different type of solution is to allow frames to be retired out-of-order. This requires a more sophisticated mechanism for detecting when a frame can no longer be squashed. In some cases this may be possible dynamically. For a frame to be retired three conditions need to be satisfied. First the frame should not share any addresses with earlier frames. Second all the addresses involved must be safe, that is not subject to possible squashing and re-execution. Third any other data values inherited by the frame must be safe. In some code, such as matrix multiply, where all addresses are computed from loop indices and no code is conditionally executed this may work well. However, small things such as a write via a pointer or the conditional execution of a block may prevent all subsequent frames from being retired. Thus, this mechanism may be of use in only a small number of very regular programs.

Detecting safety for out-of-order retirement is essentially the same problem as detecting safety for execution in non-speculative parallel machines. For example, if all the iterations in a loop are known to be independent of one another then they can be retired independently. The advantage of this approach is that much work has been done on the static detection of code independence and much is known about how compilers can detect it and how programmers can assert it with pragmas. The disadvantage is that the transparency of ILP is lost, and the programmer has to become involved in the extraction of parallelism.

4.3 Compiler Scheduling

A compiler can reorder some code, particularly nested loops, to use frames more efficiently. The independence of the inner loop in matrix multiply from the other loops provides a straightforward example. Figure 5 shows how resource usage varies over time for three different orderings of the loops in matrix multiply. Re-arranging the code so that the independent parallel events of the inner loop are interleaved lets the system extract more parallelism. Also, the amount of squashing is reduced, due to the increased latency in the sequential order of events. Re-ordering only helps when resources are limited as otherwise the parallelism is limited only by the control and data dependencies.

The effects of loop reordering are illustrated in Figure 6 for matrix multiplication. In 6 b) each labelled box represents an iteration of the computation in the inner loop. The data dependence constraints imposed by the algorithm are indicated by the heavy arrows. 6 c) shows how a 3×3 matrix multiplication is mapped onto a processor with space for two frames. With the loops ordered i–j–k a maximum parallelism of 1.5 is achieved. If the loops are re-ordered i–k–j then the iterations can

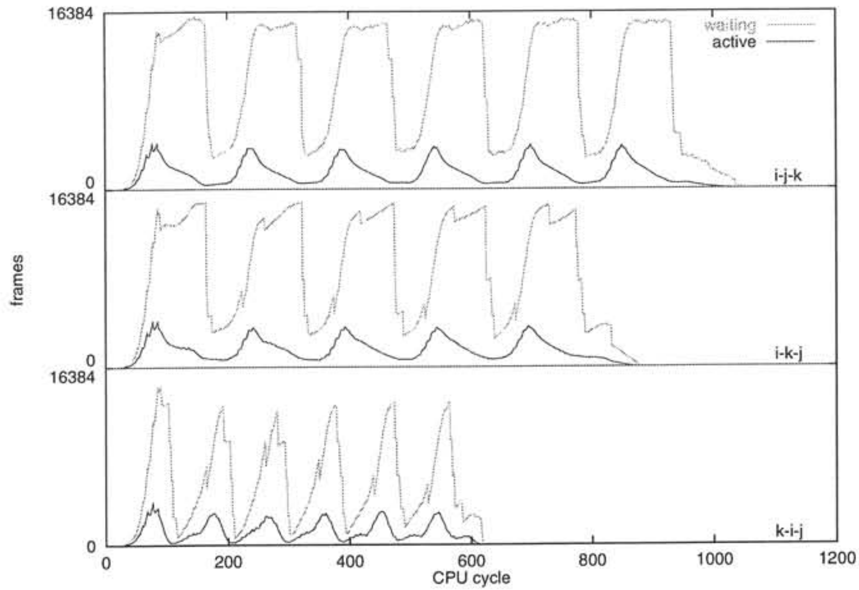

Figure 5: The effect of nested loop reordering on usage of limited resources for matrix multiplication (30x30).

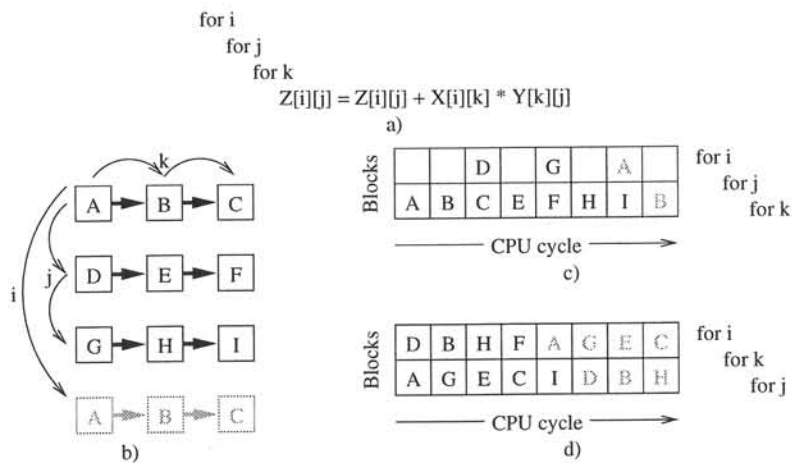

Figure 6: Matrix multiplication loop iterations mapped onto two frames with different loop orderings in place.

be mapped onto the processor as shown in 6 b). Here the maximum parallelism of 2 is attained as the processor resources can be fully utilised. A similar re-ordering can be done on the inner two loops of the Gauss-Jordan algorithm.

5 Conclusions

By using memory speculative execution, large amounts of inherent parallelism can be extracted from code. However, as we have shown, when this is done more resources are needed than simply those for the active parts of the program. Code which has been successfully speculatively executed may still be consuming valuable resources for sometime after it has completed. This will result in oscillations in resource utilisation, as seen in the graphs in Section 3, as large segments of successfully speculated execution are retired in-order. As well a small fraction of the resources are being used by code that is actively being executed. This inefficient usage of resources is an important potential performance bottleneck.

Fortunately there are some measures we can take to ameliorate this problem. If it is unlikely that a block of code will be squashed then it is possible to store less complete information about he code at the cost of more re-execution if a squash does occur. Another possibility is storing the state information on slower but less expensive memory at the cost of a lag to retrieve this information if a squash occurs.

One very important part of such a system is detecting when code can be retired and the state recorded with it deleted. A generic solution is to find the earliest code segment that can possibly be squashed and to retire everything earlier than this. However, as noted above, this can lead to a lot of state sitting inactive waiting for retirement. This can be helped by ensuring a fast accurate estimation of the earliest squash point and by the compiler rescheduling code to minimize the gap between becoming inactive and retirement. For example, reordering the loops in matrix multiply significantly reduces this lag. To do this requires substantial code analysis to determine the most inherently parallelisable parts of the code and the inter-relationships between the loops.

Another possibility is to detect code which can safely be retired out-of-order. The analysis for this could be done at execution time or at compile time. Such analysis is very similar to the safety analysis necessary in non-speculative systems to detect when code fragments are independent of one another and so can safely be executed in parallel.

The impact and complexity of the solutions proposed varies widely and much work is still required to assess their relative merits. In particular out-of-order retirement requires careful analysis of code dependencies and this may prove difficult. We believe the issues raised in this paper will be important in any architecture which attempts to significantly extend ILP beyond what is currently possible.

6 Acknowledgements

This work was funded by the New Zealand Marsden fund.

References

[Cal97] Jason Calvert. Design of the execution control structure for the "WarpEngine' CPU. Master's thesis, University of Waikato, 1997.

[CMP97] John G. Cleary, J. A. David McWha, and Murray Pearson. Timestamp representations for virtual sequences. In *Proceedings of 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, Lockenhaus, Austria, June 1997.

[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.

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

[LS96] Mikko H. Lipasti and John Paul Shen. Extending the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE Symposium on Microarchitecture*, December 1996.

[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.

[PLMC97] Murray W. Pearson, Richard H. Littin, J. A. David McWha, and John G. Cleary. Applying Time Warp to CPU design. In *High Performance Computing Conference '97*, Bangalore, India, 1997. IEEE.

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

[PS93] C. H. Perleburg and Alan J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

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

[SBV95] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd International Symposium on Computer Architecture (ISCA-22)*, 1995.