A Microthreaded Chip Multiprocessor with a Vector instruction Set

Keywords: multithreaded processor, microthreaded processor, chip multiprocessor, dynamic register allocation, vector instruction set

Abstract

This paper describes a microthreaded, multiprocessor and presents simulations from a single processor implementation. The microthreaded approach obtains threads from a single context and exploits both vector and instruction level parallelism (ILP). Threaded code can be generated from sequential code, where loops may be transformed into families of, possibly dependent, concurrent threads. Instruction fetch and issue are controlled by statically labelling instructions for vertical or horizontal transfer, depending on whether a potential data or control dependency exists. Horizontal transfer is the deterministic component of conventional next instruction processing (increment PC or unconditional branch). Vertical transfer is a context switch, which executes the next instruction from a ready thread. This allows non-determinism in data access (cache miss or signalling between concurrent threads) and in control (all conditional branches are labelled vertical). The paper will outline the microarchitecture, the thread creation mechanism based on a vector instruction set and the synchronisation techniques used. It describes a novel approach to dynamic register allocation that supports regular dependencies between families of threads. We present simulation results for a simple 5-stage pipeline, using a three level memory hierarchy. We have measured the influence of two parameters, cache delay and number of registers. The results show that the microthreaded performance is significantly superior to the conventional pipeline on which it is based. We show that the microthreaded pipeline can achieves an IPC of 0.8, even in the presence of a 1000 cycle L2-cache miss penalty.

1. Introduction

in microarchitecture, certainly in commercial microprocessor developments, have sought to extract ILP in multiple-issue pipelines by the use of speculation to expose concurrency in compiled code. Having exposed the concurrency it is then exploited using out-of-order execution. This dynamic partitioning of sequential code into concurrently executing instructions, while maintaining sequential order semantics, requires substantial hardware resources to control and does not provide a universal solution. In particular, irregular accesses to memory and unpredictable branching patterns will mean dramatic reduction in performance. The motivation for this approach is the ability to execute legacy binary code. It is accepted that the limit to the effectiveness of this approach is an issue width of approximately 4.

More recent commercial microprocessor developments have adopted source-code rather than binary-code compatibility. The EPIC architecture, for example, revitalises a technique that predates superscaler, namely VLIW, which avoids the substantial hardware resource required in superscaler pipelines. It does this by having the compiler generate a fixed schedule of instructions, and that also causes its major limitation. The fixed schedule is destroyed on every conditional branch or cache miss. In EPIC, this is solved by speculation again but in a slightly disguised form. Loads are hoisted speculatively, with the computational schedule simply checking the success of those prefetches. Conditionals, where possible, are transformed into predicated execution, which in practice executes both paths committing only one. Again if everything goes well, performance is good but now the failure of this speculation results in a software exception and again a large hit in performance.

We will show in this paper that speculation can be avoided altogether, while still maintaining high performance, but only by sacrificing binary-code compatibility. The solution is to adopt a multithreaded approach, which like EPIC, requires recompilation of source code. Multithreaded architectures are by no means new[5] and this early work by Burton Smith provided solutions to very fine grain ILP and could cope effectively with both branch and data non-determinacy. Its performance however, was poor on single-threaded code and this issue seems to have dominated recent research on multithreaded architectures. It seems that a combination of superscaler and multithreading, simultaneous multithreading (SMT) is now the most likely studied direction. There have been many papers showing the benefits of the SMT approach[6,7] and also its problems[8,9].

SMT inherits the complexity of the superscaler approach and although this may not be considered an issue with today's superchips, we must reflect on whether that real-estate could not be better used in developing a chip multiprocessor (CMP). There are complex cost and trade-off issues involved here. We have no complete answer to this question yet but we do demonstrate an alternative approach that avoids the cost of speculation and out-of-order analysis in hardware. We also demonstrate high efficiencies on a range of codes with memory latencies up to 1000 processor cycles. This approach has many of the advantages SMT but without the complexity. Microthreading was first introduced 1996[1] and has been developed into an approach that supports chip multiprocessing with a vector instruction set[2,3]. Microthreads are very small fragments of code that can be interleaved in a pipeline, creating a dynamic schedule. Other work uses this level of granularity[10] but as an adjunct to the SMT approach. It is our contention that the simplification of the processor in the microthreaded approach can be used to introduce on-chip multiprocessing and moreover, that the microthreaded scheduling mechanism can be used to give a scalable solution, within hardware constraints. In this paper, we present our first significant results from the simulation of this approach. The simulations are for a single processor system and a comparison is made against the base architecture.

Code for the microthreaded architecture can be compiled from legacy source code by identifying and exploiting the ILP that conventional approaches use. In addition the vector instruction set allows loop-level parallelism to be exploited. The instruction set and synchronisation model allow the vectorisation of loops with single regular dependencies, as will be demonstrated in this paper. A comparative analysis of microthreading against other approaches, such as SMT, requires more work on compilation tools than we have currently completed and the results in this paper were all hand compiled from C code fragments.

2. Micro Architecture of the microthreaded pipeline

2.1 Microthreaded approach

The microthreaded approach is generic and can be applied to any pipelined RISC architecture. For the purpose of this evaluation we chose the MIPS instruction set architecture with a simple 5-stage pipeline. Because the technique of microthreading avoids all speculation or multi-path execution, it is both simple and efficient and hence it is eminently suitable as a basis for a chip multiprocessor design. The datapath of a microthreaded pipeline is given in figure 1. It shows a single pipeline but indicates which components are shared in a CMP design. The components added to a conventional pipeline to support microthreading are:

- *Global continuation queue (GCQ),* this holds thread descriptors that comprise a pointer to the code, the register resources required and an iterator, itself comprising of a *start*, *step* and *limit* to create families of identical threads (the *vector instructions*). The descriptor also hold a *dependency distance.* There is one GCQ per CMP.
- R*egister allocation unit (RAU),* this takes a thread descriptor and if the processor has resources (i.e. registers and LCQ slot), it issues a thread to the LCQ. There is one RAU per CMP, which which has allocation tables for global and each processor's local registers.
- *Local continuation queue (LCQ)*, this holds the state of all allocated threads, which comprise a program counter, three base addresses for the dynamically allocated registers and the thread's current status, {*waiting, ready, running* or *killed*}. The register base addresses are *local base*, *shared base* and *dependent base*. There is one LCQ per processor in a CMP, each can accept one new thread per cycle from the GCQ, via the RAU

The microthreaded execution model is of a main thread, which dynamically creates other threads as and when required by executing instructions generated at compile time. Threads execute and die, possibly creating other threads during their short existence. Synchronisation between threads is via the registers.

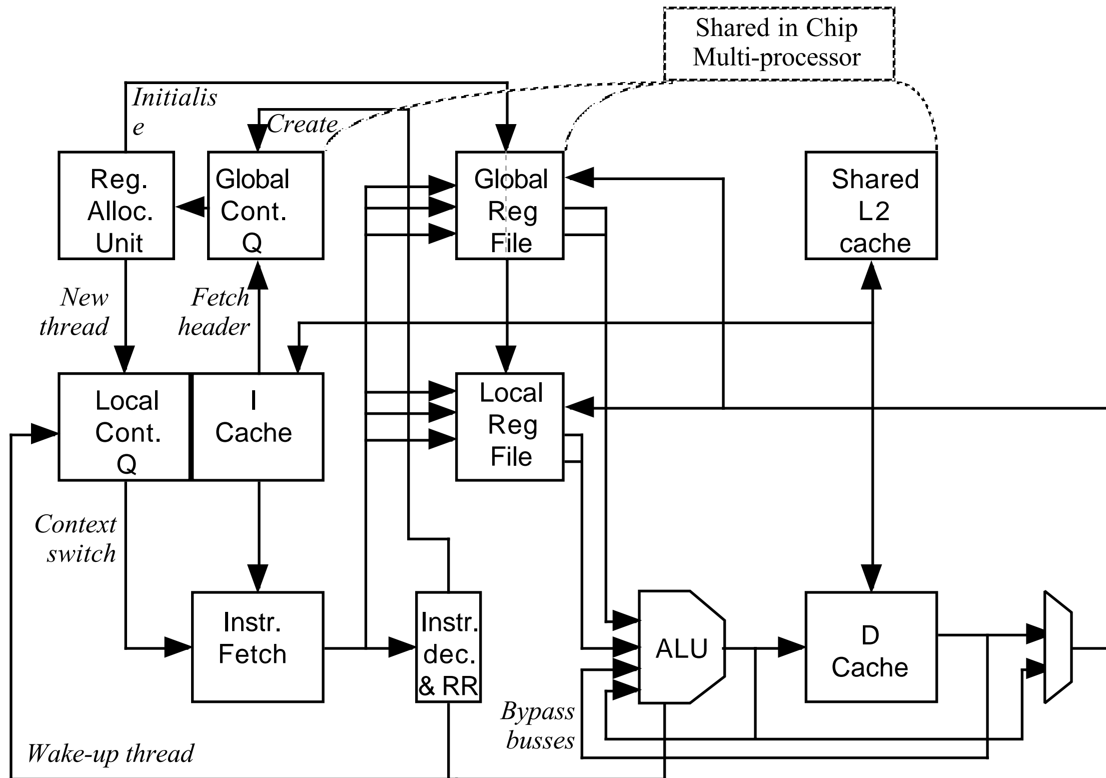

Figure 1. Datapath of a microthreaded vector instruction set pipeline

2.2 Instruction set

To a the MIPS instruction set we add just a few new instructions. We add a create thread instruction, *Cre*, with the same format as a jump, it creates a family of Microthreads and contains a pointer to the thread's code and header. For a loop, we use one thread per iteration and the iterator is stored in the thread's header. We also add conditional create instructions, *Creq* and *Crne*, with the same format as Beq. *Wait* $x is a pseudo instruction, which adds $0 to $x and waits until $x is valid. *Last* is an instruction, which waits until the LCQ is empty except for the main thread. Finally we have a *Killall* instruction that clears all entries in the LCQ, except the main thread. Last and Killall can only be issued from the main thread and must transfer control horizontally. A thread is allocated to a processor based on resource availability and runs to completion there. Code generation strategies should therefore produce threads that execute only a few instructions and then die. From [1] it is shown that thread termination and context switching are a zero cost operations and that thread creation requires one pipeline slot. To complete the model, synchronisation is also a zero-cost operation when it succeeds, and costs 1 pipeline cycle per failed synchronisation (up to 2 for 2 operands). Those cycles are for the failed instruction's re-execution after the synchronisation has been satisfied.

2.3 Synchronisation model

Context switches are flagged by the compiler by analysing dependencies and tagging those instructions whose register's contents can not be guaranteed, e.g. after a decoupled memory read[4] (memory-processor dependency) or when another thread must write to that register (RAW dependency). Thread switching also occurs on a conditional branch instruction. These flagged instructions transfer control vertically, which means the next instruction comes from another thread. The thread executing the vertical instruction has its state in the LCQ set to waiting and its LCQ slot number is passed down the pipeline. For a branch, the slot number is passed back to the LCQ with the branch target at the execute stage and the thread becomes ready

again. In the case of data synchronisation, the action is determined at the register read pipeline stage, which uses register tags with *full*, *empty* and *waiting* states. If both source registers contain valid data, i.e. are flagged as being full, then the instruction continues to execute and the LCQ slot number is returned to the LCQ after only two cycles and the thread becomes ready again with no penalty in pipeline slots used. If an empty state is detected on either operand, the current instruction is mutated into one that stores the LCQ slot and processor number into the empty register (or one of the registers if both operands are empty), setting the register state to waiting. When data arrives from memory or when the register is written to by another thread, the LCQ slot and processor number are extracted from that register and the slot number is returned to the appropriate processor's LCQ, where the thread becomes ready again. The register is then set to full. Thus in the waiting state the synchronising register contains the LCQ and processor number of the waiting thread. This thread wake-up requires the instruction to be reissued to read the data that has just arrived. Its PC must therefore be decremented. The failed synchronisation requires just 1 pipeline slot for the reissue. An additional stall for one cycle may also be required to generate a 'move' instruction to write the result from L2 cache into the appropriate register, depending on implementation. (Note that all memory accesses are tagged with processor and register number).

Thus each register may act as a synchroniser for one dependent thread only but that thread can act as a guard and create as many other threads as necessary after synchronisation. This synchronisation model is very similar to that used in the INMOS transputer[13], although there the units of synchronisation were processes and not threads and the synchronisation variables were channels, either mapped onto memory locations or communication channel registers. The principle of storing a reference to the suspended process in the synchronisation variable is the same. This method of synchronisation is also very efficient compared to recent proposals from one of the designers of the transputer[12].

2.4 Register allocation

Registers are allocated to threads when the thread is allocated to a processor from the GCQ. The GCQ iterates the vector description in the thread header and allocates one thread instance per clock to each processor, until resources limit this rate or until the iteration is complete. Then it iterates the next descriptor in the next cycle. Both local and shared registers are allocated and by default the first local register, $L0, is initialised to the index value for that thread and all other resisters are set to empty.

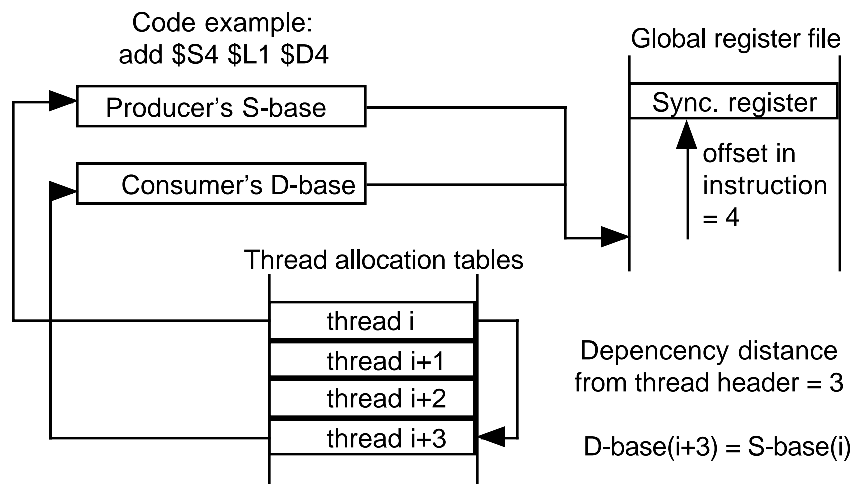

Figure 2. Locating shared registers between producer and consumer threads

Thread-to-thread synchronisation must occur in the global register file, as producer and consumer threads may be running on different processors. Memory synchronisation may be in local registers. The way in which the producer and consumer threads locate the same register is as follows:

i. the producer is allocated first and all threads must be allocated in create/index order;
ii. the producer writes writes to $Sx (at shared-base + x) n.b. shared registers are dynamically allocated from the global register file;
iii. when the consumer thread is allocated, the producer's shared base is located in the allocation tables using a dependency distance stored in the thread's header and the consumer's dependent base is set to the shared base of the producer (see figure 2 and 3.). No registers are allocated for the dependent base.
iv. The consumer thread reads from $Dx. (at dependent base + x).

Thus a further restriction is that a thread may be dependent on one and only one other thread, although there may be as many dependencies between them as there are register specifiers in the ISA. This restriction can be overcome by dividing loop bodies with multiple regular dependencies into multiple threads, with one dependency per thread.

2.5 Thread state and I-cache pre-fetching

Another advantage of using Microthreads is that the state of a thread can be used to determine a pre-fetch and replacement strategy for the I-cache. All threads are identified in the processor by their slot in the state table. This holds for each thread a full/empty flag, a program counter, the base addresses of local, shared and dependent registers, the slot of any thread dependent on it and two bits encoding the {waiting, ready, running, and killed} states. See figure 3.

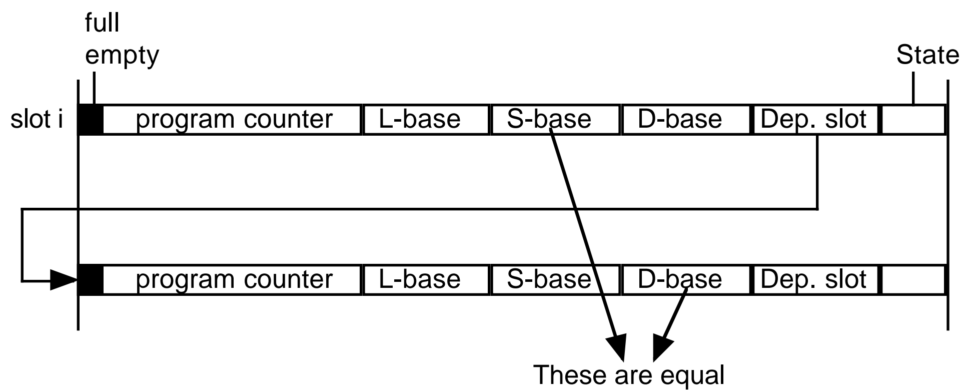

Figure 3. LCQ thread state table.

While the LCQ thread-state table is not completely full the LCQ accepts a *new* thread request from the RAU, with PC and all base addresses for that thread. This associates an LCQ slot with the thread and sets its state to waiting. A thread may be in the waiting state because its code is not in I-cache or because it is waiting on a register pending synchronisation. A request is then made to the I-cache to pre-fetch instructions for that thread. The request is acknowledged, either immediately or when it has been satisfied by a higher level of the memory hierarchy. The I-cache *ack*nowledge signal changes the thread's state to ready.

At any time there is one thread, which is in the running state, initially this is the main thread. The running thread's slot, PC and base addresses are held in the pipeline's PC. When the running thread encounters an instruction tagged for a vertical transfer or kill, the instruction fetch logic requests a *context* switch from the LCQ and one of the ready threads is selected as running and its state is passed to the PC. If no threads are available the current thread continues executing but may be stalled further down the pipe. If the context signal was a kill, the sate of the issuing thread is set to killed but its resources are not released until any thread dependent on it has also been killed. Only then is the slot is set to empty and the RAU sent a *release* signal for that slot. If the context signal was vertical, the issuing thread is set to waiting and the I-cache is sent a release signal for that slot. The I-cache may now use this information for cache line replacement if it has requests pending.

The slot reference for all instructions is available at all stages in the pipeline. When vertically transferred

instructions are resolved, i.e. a branch target is calculated or a data dependency is resolved, the slot reference is returned to the LCQ with a *wake* up signal with an optional *decr*ement, if the instruction suspended requires reissuing. This again initiates an I-cache request/acknowledge cycle before the thread's state is changed from waiting to ready. These interaction with the LCQ are illustrated in figure 4 in more detail than figure 1.

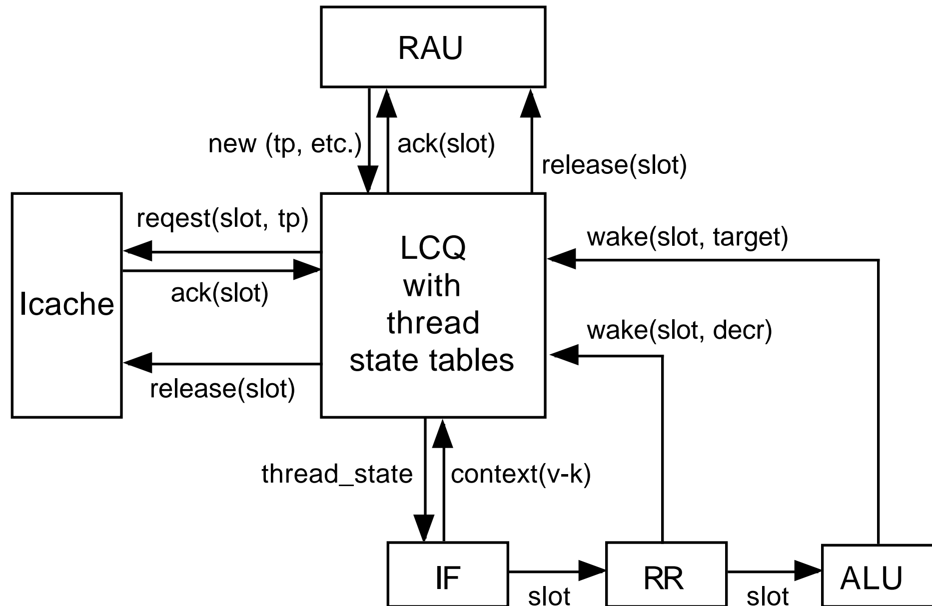

Figure 4. LCQ and its interactions with register allocation unit, I-cache and pipeline stages

2.6 Example of code generation for a simple loop

To illustrate code generation we hand compile a simple relaxation loop, which is also simulated.

```
For (i=1; i<=n; i++)
    A[i] = a[i-1] – 2*a[i] + a[i+1] ;
```

This is a tight loop that contains a single data dependency between adjacent iterations (dependency distance = 1) and because of this the code could not normally be vectorised. It also contains very little ILP. The assembler code is given below.

```
main:     (h)cre vect              # creates all threads to execute the loop
          (h)cre sync              # creates a synchronisation thread for loop termination
          (h)mv $S0 0              # $S0 <- 0… a[i-1] in first iteration
          (v)mv $G1 $G1            # waits for sync thread to set $G1
          end                      # end simulation

vect:     {1,n,1; 1; 2,2}          # thread header
          (h)lw $L1 a+1($L0)       # $L1 <- a[I+1] n.b. $L0 = i
          (v)mv $S1 $L1            # save value to S1
          (v)mul $L1 $D1 2         # wait for a[i] and multiply by #const 2
          (h)sub $L1 $S1 $L1       # wait for a[i+1], $L1 <- a[i+1] - #2*a[i]
          (v)add $S0 $L1 $D0       # wait a[i-1] (a[i] from prev. iter.) #& add
          (k)sw $S0 a($L0)         # finally store result and terminate

sync:     {1,1,1; 1; 0,0}          # thread header
          (k)mv $G1 $D0            # waits on last thread to generate a[i]

                                   # and signals the main thread via G0
```

The compiled code comprises a family of n threads labelled **vect**. and a synchronisation thread labelled **sync**, whose purpose is only to detect the termination of the loop. The thread headers shown and are n-tuples containing {start, limit, step; dep_dist; #locals, #globals}. Note that the data dependency is encoded in one line of code, namely: **add $S0 $L1 $D0**, which waits on a[i-1] from the previous thread (in $D0)

and adds it to the partial result (in $L1). The result, a[i], is stored in $S0 which in turn signals the next thread in sequence via its register $D0. The latency for resolving a loop-to-loop dependency is therefore only one cycle per iteration as the result of the add is available from the bypass bus in the next cycle.

3. Simulation Results

3.1 Simulation Environment

We have constructed a simulator to evaluate the performance of the architecture described above. The results give the performance of a single microthreaded processor and compare this to that of the conventional pipeline used as the basis for the microthreaded design. The results therefore show the improvements that are possible by adding microthreading to an existing design. The simulator also includes a complete simulation os a three-level memory hierarchy, see table 1. This work is based on a discrete-event simulation framework from the University of Paris Sud[14]. The pipeline simulated is a modified, five-stage, MIPS-like design, which is both simple and efficient. It is single issue and does not support speculation. In the conventional pipeline a branch delay slot is simulated and code is generated accordingly. This basic design is then modified with the components required to support microthreading, such as the GCQ, LCQ RAU and the control logic for the management and execution of Microthreads. Both microthreaded and conventional simulators accurately model the cycle by cycle behaviour of the processor in all pipeline stages and levels of memory hierarchy. We can produce very detailed log files of the execution of the code, showing exactly what is happening in the pipeline in order to investigate its behaviour.

3.2 The Memory Simulation

One of the goals of this work is to bridge the memory-performance gap for all codes regardless of their access patterns. To demonstrate this we have had to design a realistic cache simulator. The cache simulator simulates the exact structure and behaviour of the real cache. We have provision for a large range of cache parameters for the evaluation (see Table 1) so that we can obtain a detailed comparison of different environments. We assume that the memory capacity is unlimited and is always hit.

	L1 cache	**L2 cache**	**memory**
Size (bytes)	1-128	64-8M	Unlimited
Associativity	1-8	1-8	N/A
Line size (bytes)	4-128	4-128	N/A
Hit time (cycles)	1	5	10-1000
Latency on miss	5-10	10-1000	N/A
Write policy	Write through	Copy back	N/A
Replacement alg.	LRU	LRU	N/A

Table 1: Parameter range for the microthreading microprocessor cache hierarchy

In all simulations presented in this paper, it is assumed that the I-cache always hits. Because we are only simulating small code fragments at this stage a full I-cache simulation is not required. However, the microthread I-cache does support a pre-fetch mechanism that avoids any cache misses on ready threads. Also unless otherwise specified, simulations use 4-way set associative L1 D-cache and L2 cache, with LRU replacement. Both caches use block sizes of 32 bytes.

3.3 Simple relaxation loop

The first simulations performed were on the code above compiled from the relaxation loop. This was simulated with a level 1 cache miss latency of 5 processor cycles and a level 2 miss latency of between 10 and 1000 processor cycles, representing a range of memory architectures from tightly coupled to distributed. The code was simulated for a range of values of loop limit, n, from 1 through to 1000 and this is shown in

figure 5. Curves are labelled x/y, where x is the L1 cache miss penalty and y is the L2 miss penalty in clock cycles. In all graphs, the curves labelled MT are the microthreaded pipeline and are solid lines and the dashed lines are the conventional pipeline. In this example, the code is very regular and consequently both microthreaded and conventional pipelines give good performance with a small value of memory latency 5/10 cycles. The microthreaded pipe always delivers the superior performance, with an asymptotic IPC of 0.8 and a half performance vector length of just a few iterations. As the L2 cache miss penalty is increased to 100 and then 1000 cycles, the performance of the conventional pipeline suffers significantly, as would be expected. For the microthreaded pipeline however, the performance always reaches the same asymptotic level (IPC=0.8), although it requires more iteration to achieve this. For a 100 cycle L2 miss penalty the half performance vector length is about 10 iterations and for a 1000 cycle L2 miss penalty about 100 cycles. In the latter case the microthreaded pipeline has an asymptotic performance 6 times that of the conventional pipeline. This simulation was repeated with a direct mapped L1 D-cache. We found a negligible decrease in performance in the convention pipeline but none in the microthreaded one.

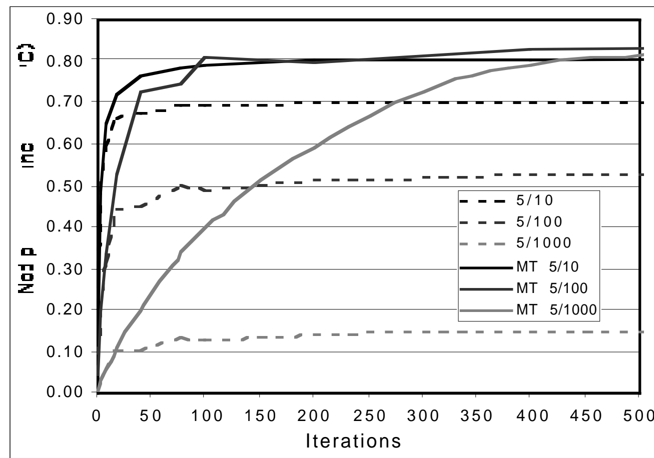

Figure 5 Relative performance of microthreaded pipe against conventional pipeline on relaxation loop.

The concurrency exposed in this loop is not great, there are two sequences of two non-dependent instructions and three data dependencies, one load and two thread-to-thread dependencies. Each iteration requires two local and two shared registers. The number of threads that can be allocated concurrently is therefore limited by the either the number of iterations or the number of registers. The above results assume unlimited registers. Figure 6 shows the performance of the microthreaded pipeline for the two higher miss latencies as a function of the number of registers. As would be expected, except when the code is overhead limited, performance is linear in the number of registers.

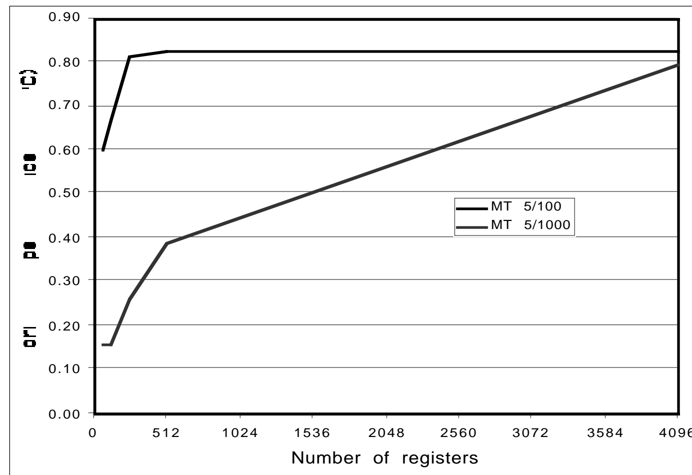

Figure 6. Performance of the microthreaded pipeline against number of registers for the relaxation loop

3.4 Results for some Livermore loops

The Livermore loops are a set of 24 loop kernels extracted from operational codes used at the Lawrence Livermore National Laboratory [11] and which are used to benchmark vectorising compilers and vector-architectures. They include both vectorisable and non-vectorisable loops and test rather fully the computational capabilities of the hardware, and the compilers in generating efficient code through vectorisation. We have simulated loops K2, K3 and K17. These have been hand analysed and compiled. All contain dependencies and all have been chosen as difficult to extract concurrency from.

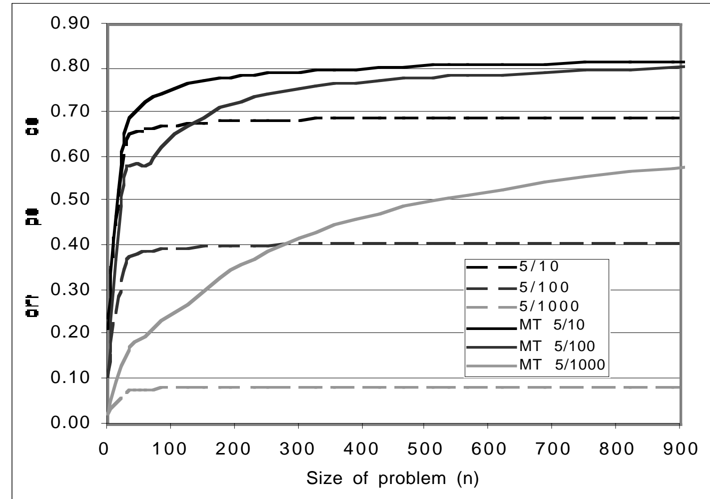

Figure 7. Relative performance of the microthreaded pipe on Livermore K2 loop

K2 is an ICCG kernel and contains both write-after-read and read-after-write dependencies. It is typical of divide and conquer algorithms and has a loop that varies from 1 to n/2 by powers of 2. Figure 7 gives the results for the K2 loop, it shows very similar pattern to our simple dependent loop only a larger problem size is required in order to gain the same asymptotic performance. This is to be expected as the loops are on average only a quarter of the problem size. Notice that for the higher cache memory latencies, while the microthreaded code still gives the same asymptotic performance, the conventional pipeline gives 20-30% worse performance, when compared to the simple relaxation above. This is due the code being less regular in the way it accesses memory, producing fewer hits.

The K3 Livermore loop is the inner product. The code sequentially accumulates the inner product in a scalar variable. In our code we have translated the loop as written, i.e.:

```
for ( k=0 ; k<n ; k++ ) {q += z[k]*x[k]; }
```

without resorting to a non-programmed divide and conquer algorithm that would expose more concurrency. The compiled code for the loop is as follows:

```
vector_inner:   {1,n,1; 1; 3,1}             # thread header
                (h) lw $L1, x_start($L0)    # x[k] // $L0 = k = threadReference-1
                (h) lw $L2, z_start($L0)    # z[k]
                (v) mul $L2, $L2, $L1       # z[k]*x[k]
                (k) add $S0, $D0, $L2       # q = q + z[k]*x[k] $S0 signals dependent thread via $D0
```

The code is similar to the relaxation loop. There are just 4 instructions in the loop, two loads, which may miss cache, a multiply requiring both but independent of each other in different iterations and an add dependent on the previous iteration. The conventional code has two more instructions to control the loop, one to increment the index and a conditional branch to terminate the loop. These functions are performed in hardware in our architecture, the loop iteration in the GCQ and the index initialisation for each thread in the RAU.

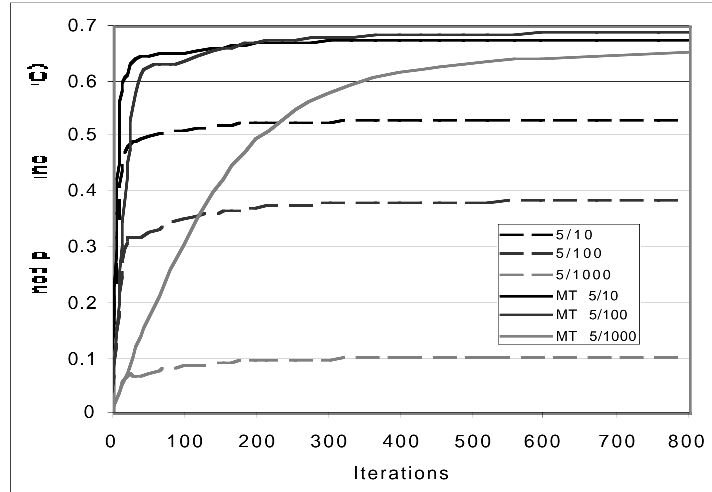

Figure 8. Relative performance of microthreaded pipe on Livermore K3 loop

Figure 8 shows the performance of the microthreaded pipeline for the Livermore K3 loop kernal. The results are similar to the relaxation loop but the performance is lower in all cases. The microthreaded pipeline again achieves the same asymptotic performance regardless of the the cache delay simulated but requires more iterations to achieve it. For the worst case, a 1000 cycle L2 cache miss penalty, the half performance vector length is 120, some 20% higher than the relaxation loop. What is significant is that for 240 plus iterations, the microthreaded pipeline has a better performance with a 1000 cycle penalty, than the conventional pipeline has with a miss penalty that is 2 orders of magnitude smaller!

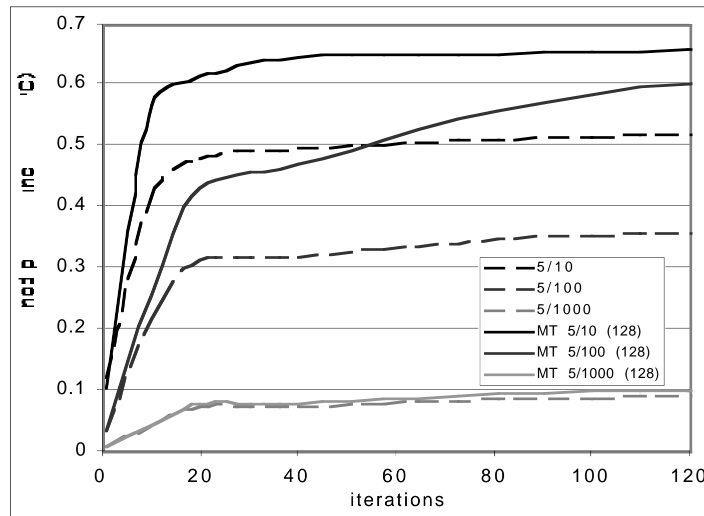

Figure 9 Relative performance of microthreaded pipe on Livermore K3 loop with 128 registers

The simulations for K3 were repeated with a fixed number of registers (128) and the results are shown in figure 9. For the 1000 cycle L2 miss penalty, performance is register limited and it is only marginally better than the conventional pipeline. Because 4 registers (3 local and 1 global/shared) are required per iteration less than 32 iterations can run in parallel in this configuration, which is insufficient to tolerate cache misses of 1000 cycles. In a multiprocessor system however, this code would scale rather well, as local registers would be added with each processor.

The K17 Livermore loop is plain old spaghetti code and is difficult to vectorise even allowing dependencies. The loop is implicit and formed with a pair of conditional branches. The technique we have used to expose concurrency is a limited form of speculation on the outcome of these branches. The next iteration is created

as a scalar thread early in the loop, prior to any branch being taken. This does expose concurrency, although at the expense of some overhead because termination control must be added to ensure sequential semantics.

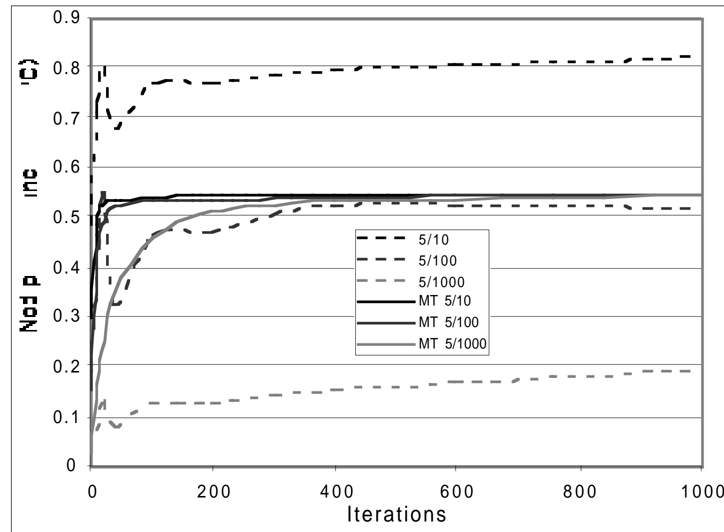

Figure 10. Relative performance of microthreaded pipe lien on Livermore K17 loop

Figure 10 shows the comparison of the microthreaded and conventional pipeline for the Livermore K17 loop kernel. This is the only code that we have analysed where, for low-latency memory, the conventional pipeline has given a better asymptotic performance than the microthreaded one. However notice that concurrency is being exposed as the microthreaded result shows identical asymptotic performance, regardless of the memory latency, whereas the conventional pipeline takes a large hit in performance as cache penalty is increased.

4. Conclusions

In this paper we have introduced a technique for microthreading simple pipelines that draws all its threads from the same context and hence has very efficient thread-scheduling and synchronisation. The architecture also supports a vector instruction set, where the vector instruction is a loop body, which can contain a regular dependency that is iterated over a regular index space. Registers are allocated to loop bodies, which are dynamically instanced as Microthreads. This means a single instance of the loop code can be used for as many iterations as there are registers for their results. In a multiprocessor system, this would scale with the number of processors.

We show that using this technique we can obtain a high utilisation of a simple pipeline, both in the presence of branches of control and when we have cache misses. This is true for even very large L2 miss penalties, of the kind that would be encountered in a distributed memory system. We show results for a relaxation loop with a dependency between iterations and for three difficult-to-vectorise loops from the Livermore loops benchmark. In all cases the asymptotic performance of the microthreaded pipeline is independent of the cache delay. These results were obtained by assuming unlimited registers. We have shown that concurrency exposed is limited by either the number of iterations (even in the presence of dependencies) or the number of registers available. On any given problem the crossover will depend on the ILP in the loop body and the number of registers allocated to it.

To date we have been unable to simulate complex benchmarks due the the limitation of not having a code generator for the microthreaded processor. Work is currently underway to build a C compiler but first we have had to investigate and simulate code generation strategies.

Although we have only simulated a single processor system, we are confident that the results we have shown will scale almost linearly with the number of pipelines used. The scheduling strategy that is used is to allocate a thread to an arbitrary processor and allow it to run to completion there. Typical thread lengths

are about 10 instructions and hence load imbalance is mitigated. In a multiprocessor system, interprocessor synchronisation is not a problem, as synchronisation between threads is achieved using the global register file that is shared between processors. Our code is both configuration independent (will run on 1 to n processors) and schedule invariant. The only issue that must be addressed is resource deadlock where dependencies can not be satisfied due to register limitation.

Physical scalability in a microthreaded multiprocessor is limited by the shared register file. The number of ports it requires will scale with the number of processors, although sub-linearly and many requests are to local registers. Secondly the size of the register file must scale with the number of processors, to maintain sufficient synchronisation variables. However, by copying shared read-only variables into local registers, we can relieve pressure on the global register file. The same issues also limit superscaler pipelines issue width. These key issues will be investigated when we complete our multiprocessor simulator.

5. References

[1] A.Bolychevsky, C.R.Jesshope, V.B.Muchnick, (1996) Dynamic scheduling in RISC architectures, *IEE Proc.-Comput. Digit. Tech.* Vol.143, No.5, September

[2] Jesshope C. R. and Luo, B. (2000) microthreading: A New Approach to Future RISC, *Proc ACAC 2000*, pp34-41, ISBN 0-7695-0512-0 (IEEE Computer Society press), Canberra Jan 2000.

[3] Jesshope, C. R. (2001) Implementing an efficient vector instruction set in a chip multi-processor using microthreaded pipelines, *Proc. ACSAC 2001, Australia Computer Science Communications*, Vol 23, No 4., pp80-88, IEEE Computer Society (Los Alimitos, CA, USA), ISBN 0-7695-0954-1.

[4] Smith J. E. (1998) Retrospective: decoupled accss/execute architectyures, *in 25 years of the Internation Symposia on Computer Architectures*, p42.

[5] Smith, B. J. (1978) A pipelined shared resource MIMD computer, *Proc. Intl. Conf on Parallel processing.*

[6] Gulati, M. and Bagherzadeh, N. (1996) Performance study of a multithreaded superscalar microprocessor, *Proc. of the Second International Symposium on High-Performance Computer Architecture*, pp291-301, San Jose, California, February 3-7, 1996. IEEE Computer Society TCCA.

[7] Eggers, S. J., J. Emer, J., Levy, H. M., Lo, J. L., Stamm, R., and D. M. Tullsen. *Simultaneous multithreading: A platform for next-generation processors*, TechnicalReport TR-97-04-02, University of Washington, Department of Computer Science and Engineering, April 1997.

[8] Farcy, A.and Temam, O. Improving single-process performance with multithreaded processors. In Proc. of the 1996 International Conference on Computing, pp 350-357, New York, May25-28 1996. ACM.

[9]. Hily, S. and Seznec, A. *Contention on 2nd level cache may limit the effctiveness of simultaneous multithreading*. Technical Report PI-1086, IRISA, University of Rennes 1, 35042 Rennes, France, February 1997.

[10] Chappell R. S., Stark, J., Kim, S. K., Reinhardt, S. K. and Yale, (1999) Simultaneous subordinate microthreading (SSMT) *Proc. of the 26th annual Intl. Symposium on Computer aArchitecture.* IEEE Computer Society Press, Los Alamitos, CA, USA, ISSN:0163-5964, pp186-195.

[11] F. H. McMahon. The Livermore Fortran Kernels test of the Numerical Performance Range. In J. L. Martin, editor, Performance Evaluation of Supercomputers, pages 143--186. Elsevier Science B.V., North-Holland, Amsterdam, 1988.

[12] Shondip, S., Muller, H., and May, D. (2000) Synchronisation in a multi-threaded processor, *Communicationg processor architectures* (Welch, P. H. and bakkers, A.W.P.) IOS Press, Amsterdam, Netherlands, pp137-144.

[13] Whitby-Strevens, C. (1985) The transputer, *IEEE conf. Parallel Processing And Computer Architecture*, p292-300.

[14] Architecture Simulation Framework, http://www.lri.fr/~osmose/ASF/, latest updated in 28/06/2001.