Multi-threaded Microprocessors – Evolution or Revolution

Chris Jesshope

Department of Computer Science, University of Hull, HU6 7RX, UK
{c.r.Jesshope@dec.hull.ac.uk}

Abstract. Threading in microprocessors is not new, the earliest threaded processor design was implemented in the late 1970s and yet only now is it being used in mainstream microprocessor architecture. This paper reviews threaded microprocessors and explains why the more popular option of out-of-order execution has a poor future and is not likely to provide a pathway for future microprocessor scalability. The first mainstream threaded architectures are beginning to emerge but unfortunately based on out-of-order execution. This paper will review the relevant trends in multi-threaded microprocessor design and look at one approach in detail, showing how wide instruction issue can be achieved and how it can provide excellent performance, latency tolerance and above all scalability with issue width. This model exploits ILP and loop level parallelism using a vector-like instruction set in a chip multiprocessor.

1 The Forces at Play in ISA Design

There are two forces that determine the form and function of microprocessor architecture today. The first is the technology and the second is the market. These forces are quite at odds with each other. On the one hand, technology is all about change. In 1965, Intel's founder Gordon Moore predicted that the number of transistors on a chip would double every 2 years. His prediction of exponential growth has not only been achieved but in some cases exceeded. On the other hand, the market is all about inertia or lack of change. At ACAC 2000, the invited speaker Rumi Zahir, who led the team responsible for the instruction set architecture of IA-64, told us an anecdotal story about the briefing his team had been given by Andy Grove. They were given a clean sheet to do whatever they wanted, but with one exception... the resulting microprocessor should be able to boot up a binary of DOS from floppy disc! In the event, Moore's law solved their problem and the Itanium core processor is not binary compatible with X86 processors, instead it has a separate compatibility unit in hardware to provide IA32 compatibility.

There are two routes to ISA development, evolutionary or revolutionary and it appears that the evolutionary route always relies on technological improvements and results in ever increasing complexity in design. We have good examples of this in current out-of-order issue superscalar microprocessors. Intel has demonstrated this approach, requiring each new ISA to be backward compatible with the previous one. On the other hand revolutionary change has been made, for example Motorola and IBM moved away from their respective CISC ISAs to the RISC-based Power PC architecture, first introduced in 1993. Such a major divergence in machine code forced Apple, a major user of the 68000 processor, to emulate the 68000 ISA on the Power PC for backward compatibility. Emulation has been used by a number of other microprocessor designs, including the Transmeta Crusoe, which was targeted at high performance but low-power applications. The benefits of speed and power savings made software emulation a practical alternative to hardware compatibility.

Perhaps we should first ask what the issues are that require changes to an ISA design as we follow the inevitable trends of Moore's law? In fact there is just one issue and that is in providing support for concurrency within the ISA. More and more gates mean increased on-chip concurrency, first in word width, now in instruction issue width. The move to a RISC ISA was revolutionary, it did not introduce concurrency explicitly, rather it introduced a simple, regular instruction set that facilitated efficient instruction execution using pipelines. In fact many people forget that the simplicity of RISC was first adopted in order to squeeze a full 32-bit microprocessor onto a single chip for the first time. RISC has also been introduced as an evolutionary development, for example, Intel's IA32 CISC ISA, which has a very small set of addressable registers, is implemented by a RISC engine with a much larger actual register file. This is achieved by dynamically translating its externally visible CISC ISA into a lower-level RISC ISA. Of course this is only possible due to the inexorable results of Moore's law. Intel was able to maintain backward compatibility in the IA32 from the 8086 in 1978 through to the Pentium 4 first introduced in 2000 but have now moved to a new ISA, which introduces a regular and explicit concurrent instruction set.

2 Concurrency in ISAs

Concurrency can be introduced into a computer's operation via the data that one instruction processes or by issuing instructions concurrently. In this paper we do not consider the data parallelism found in SIMD or vector computers, although we do look at a vector model of programming that is supported by wide instruction issue. Neither do we consider the data flow approach. This leaves just two ways in which concurrency can be introduced explicitly into conventional ISAs, through VLIW or through multi-threading. There is a third way, which is that currently used by most commercial microprocessors. This is to extract the concurrency from a sequential instruction stream dynamically in hardware. We will look at each of these in turn beginning with the excesses of the latter in terms of consuming silicon real-estate.

2.1 Out-of-Order Instruction Execution

Out-of-order instruction execution can be seen as a theoretically optimal solution for exploiting ILP concurrency, because instructions are interleaved in the wide-issue pipelines in close to programmed order, whilst honouring any data and control dependencies or indeed any storage conflicts introduced by the out-of-order instruction execution. The major benefit is that it is achieved using the existing sequential instruction stream and therefore maintains code-base compatibility. In effect, the instruction stream is dynamically decomposed into micro-threads, which are scheduled and synchronised at no cost in terms of executing additional instructions. Although this may be desirable, speedups using out-of-order execution on superscalar pipelines not so impressive and it is difficult to obtain a speedup of greater than 2, even on regular code and using 4- or 8-way superscalar issue, e.g. [1]. Moreover, they scale rather badly as issue widths are increased.

To understand why this is, let us first look at how a typical superscalar pipeline works. Instructions are prefetched, sometimes along more than one potential execution path. Instructions are then partially decoded and issued to an instruction window, which holds instruction waiting to be executed. Instructions can be issued from this window in any order, providing resource constraints can be met by register renaming. Instructions are then issued to reservation stations, which are buffers associated with each of the execution units. Here a combination of register reads and bypassing, using tagged data, matches each instruction to its data. When all data dependencies have been satisfied, the instructions can be executed. Eventually an instruction will be retired in program order by writing data into the ISA visible registers to ensure sequential execution machine state.

The first and most significant problem with this approach is that execution must proceed speculatively and even though there is a high probability of control hazards being correctly predicted [2], this must but put into context. As a rule of thumb, a basic blocks is often no longer than 6 instructions [3] and if we assume a 6-way instruction issue superscalar microprocessor with 6 pipeline stages before the branch condition is resolved [4], we are likely to have of the order of 6 branches unresolved at any time. Even with a 95% successful prediction rate for each branch, there is a 1 in 4 chance of failure in any cycle. With unpredictable branching, the situation is much worse and branch prediction failure is almost guaranteed in any cycle (98% chance of failure). These parameters will also limit multi-path prefetching, as instruction fetch and decode bandwidth is exponential in the number of unresolved brunches. In other words we could be fetching and decoding up to 64 different instruction paths in a multi path approach. A second problem is that of sequential-order or deterministic machine state, which lags significantly behind instruction fetch due to the many pipeline stages used in out-of-order execution. This means there are significant delays on non-deterministic events, such as on an interrupt or an error, caused by the miss prediction of a branch condition for example. Recovery for miss prediction therefore can have a very high latency. The final problem is one of diminishing returns for available resources [5], which we will look at in more detail below.

Out-of-order executions requires large register files, large instruction issue windows and large caches. As the issue width increases, both the number of register ports and hence the size of the register file must both increase. The physical size of the register file increases more than quadratically with instruction issue width [1] and this is largely due to the size of the register cell, which requires both horizontal and vertical busses for each port. The proposed Alpha 21464 illustrates this problem very well [6], its register file comprises 512 64 bit registers and occupies an area over four times the size of the L1 D-caches of 64KB. The area of the instruction window also grows with issue width. It can be thought of as a sliding window over the code stream within which concurrency can be extracted, it grows with the square of the number of entries due to the scoreboard logic that that controls instruction issue. The 21464 has 128 entries. It must be large so as to not unduly limit the potential ILP that may be exploited in an out-of-order issue. The problem is compounded because out-of-order execution introduces additional dependencies (WAR and WAW), which are resolved by register renaming and drive up the size of the register file. These are not real dependencies but simply resource conflicts. Again the proposed 21464 illustrates the problem well, the 128 entry out-of-order issue queue + renaming logic is approximately ten times the size of the L1 I-cache, also 64KB. Finally, out-of-order issue increases the complexity of the memory hierarchy, both in levels of cache implemented and in prefetching and cache management techniques. It is well known that caching produces only diminishing returns in terms of performance for chip area occupied and current L2 cache arrays will typically occupy between 1/3 and 1/2 of the total chip area [6].

Clearly something is very wrong with the out-of-order approach to concurrency if this extravagant consumption of on-chip resources is only providing a practical limit on IPC of about 2. Having said that further improvements in IPC have been observed in Simultaneous Multi-Threaded (SMT) architectures, which are based on the superscalar approach. However we have to consider whether this is the most appropriate solution, adding yet further hardware resources to an already non-scalable approach to increase instruction issue width still further. Note that the 21464 [6] is an SMT supported out-of-order issue architecture.

2.2 VLIW ISAs

Let us now consider explicit concurrency in an ISA using VLIW, which is both synchronous and static. VLIW encodes a number of operations into one long instruction word and these operations are executed in lock step on parallel functional units. The approach was originally called horizontal microcode as early designs used microcoded pipelines to execute the operations simultaneously. Later the name very long instruction word (VLIW) was coined. The origins of this approach can be traced back to signal processing solutions of the late 1970s and the Floating Point Systems AP120B [7] is a good early example. Although called an array processor the instruction set is wide and it executes several operations simultaneously. Array processing applies to the mode of programming, which used libraries of array-based operations. True VLIW computers were built without cache and exploited loop-intensive code. A fixed memory latency and branch behaviour that was predictable at compile-time enabled these devices to function effectively in their domain. They were not however general purpose computers. Moreover the limitation of cacheless architecture is a significant problem with modern technology, where processor speeds are significantly higher than memory speeds.

The most notable recent adoption of VLIW is Intel's new IA-64 ISA [8], renamed again to EPIC. This is a generalisation of VLIW and differs from it in a number of ways. Firstly the instruction set is designed to be future compatible. It does not describe explicit hardware resources but the extent of software concurrency. Thus each instruction packet can contain an arbitrary number of operations that are executed concurrently or sequentially depending on the extent of the instruction issue width. Secondly it provides greater flexibility than earlier VLIW ISAs by providing support for the two key problems in VLIW, namely, keeping the processor running in the presence of non-determinism in both data and control. The use of predicated instruction execution overcomes many control hazards and an explicit prefetch instruction, followed by a check when the data is required is used to avoid non-deterministic latency in memory loads.

These problems are universal but the adverse results are particularly severe in VLIW architectures, as any failure in these mechanisms can kill the schedule and force all units to wait for one hazard to be resolved. This is as a result of the lockstep nature of the ISA. This solution also comes at a cost, which is redundant computation. Predication is a form of multiple-path execution, where the compiler determines the extent of redundant computation in order to maintain the static schedule in the presence of what would normally be considered branches. Clearly any form of multi-path execution is a form of speculation, which consumes hardware resources and perhaps more importantly, energy. There are also limitations on what Intel calls data speculation, i.e. hoisting speculative loads high enough in the instruction stream to overcome potential memory latency problems, which include memory aliasing problems. Prefetches can be hoisted above conditional branches but if each branch path requires different data, speculative memory bandwidth requirements would increase exponentially with the number of branches over which the prefetch was hoisted.

2.3 Multi-threaded ISAs

We have seen that both VLIW and out-of-order issue require some form of speculation in order to operate effectively. Multi-threading on the other hand makes any form of speculation unnecessary, although some multi-threaded approaches do rely on speculation [9]. Multi-threaded instruction execution need not suffer from the problems encountered using speculative execution, with one exception and that is fundamental, it is synchronising across many concurrently issued instructions and requires a large register file. In a threaded microprocessor, it is not necessary to issue instructions in a thread out-of-order and hence we need only deal with true data dependencies. This can simplify processor design considerably, see [11], which considers a range of processor designs in developing chip multi-processors, it suggests a packing density difference of a factor of 8 between in-order issue and out-of-order issue processors. However a chip multiprocessor based on a threaded scheduling will also require additional hardware to support context stores and mechanisms for scheduling and synchronising inter-thread dependencies.

The major benefit of multi-threading is tolerance to latency in memory accesses, true concurrency and other non-deterministic events. It can even be used to avoid speculation on conditional branches [12], thus making branch prediction unnecessary, in all but single-threaded code.

In Multi-threaded code, even if a compiler decides where context switches occur, the instruction schedule is dynamic, as ready threads depend on non-deterministic events and then can be scheduled in any order. High-latency memory events, such as cache misses, true data dependencies and conditional branches are triggers which can be used to determine when to context switch, which provides a new source of instructions to be executed while the event is resolved and data produced. There is some cost for this but the cost can be made small. The result is, that instructions from more than one thread can be executed in one or more pipelines. But what impact will this have on the ISA design?

The most flexible approach is to have dynamic thread management, where instructions are added to some base ISA to provide for some or all of the following actions:

- thread creation
- thread termination
- thread synchronisation and related initialisation

Initially, this approach seem to have only an incremental impact on the ISA, leaving it backward compatible with the base ISA on single-threaded application code. We will see later however, that this is not necessarily true and in the example below we see that instruction tagging for context switches and register specifiers are also likely to change in the Multi-threaded ISA.

Multi-threading has been applied in a variety of different ways and for a variety of applications and programming paradigms. These include multiprocessor supercomputers, such as the HEP [12], Horizon [13] and Cray MTA [14], an alternative approach to the implementation of data flow computers (see [10] for the rationale) and more recently for Java byte-code engines in micro controllers [15] and streaming applications [16]. One of the more interesting recent developments is the use of threading in order to develop so called network processors [17]. This approach has been adopted by both Intel [18] and IBM [19]. It is clearly a well suited application as the low context-switching overhead of a thread microprocessor can be used meet the real-time demands of network switching.

The extent of any taxonomy in multi-threaded architecture is also dependent on the base micro architecture, instruction issue, e.g. out-of-order issue or in-order, number of instructions issued simultaneously, the extent of sharing of various resources, e.g. superscalar or multi-processor approaches, programming model, etc. Suffice it to say that most combinations have been explored. An excellent survey of processors with explicit multi-threading can be found in [20], which covers most, if not all, different approaches to multi-threading. This survey provides a number of taxonomic distinctions in Multi-threaded architectures:

Blocking and non-blocking - typically non-blocking threads are used in data flow architectures (but not exclusively so), a non-blocking thread will resolve all depend-

encies prior to launching the thread by decoupling the memory accesses from the computation, e.g. [21].
Explicit and implicit - implicit approaches attempt to increase performance of sequential code by thread-level speculation, e.g. [22]
Block-threading, interleaved multi-threading and simultaneous multi-threading - in block multi-threading instructions are executed until some event causes a context switch. Typically there will be support for a small, usually fixed number of threads, each of which has its own register set and stack pointers to maintain its context without spilling to memory. Interleaved threading is where a context switch takes place on every cycle, as instructions are interleaved from multiple threads into a pipeline that assumes no structural hazards. Finally when instructions are issued simultaneously from multiple threads to a superscalar pipeline, this is called simultaneous multi-threading. This should not be confused with multi-thread support for chip multiprocessing, where many processors without shared resources may use multiple threads to support concurrency. For many examples of each approach see [20].

3 Micro-threaded Execution Model

In this section we take a look at one particular model in detail, that based on [12]. This approach uses a block-threaded approach with a blocking model for threads. Unusually however for a block-threaded approach, it is possible to interleave threads on a cycle-by-cycle basis, as its context switching overhead is zero cycles. It is based on a simple in-order issue pipeline and is designed to support wide instruction issue using a chip-multiprocessor approach. Before we look at the model in detail and evaluate its costs, we give some results of recent simulations. More detail of simulation conditions are given in previous papers presented at this conference [24, 25].

3.1 Simulation Results

The first results show vertical threading, on one single-issue pipeline and illustrate the tolerance to latency that can be achieved. The comparisons are between the base architecture and the same architecture augmented with a micro-threaded scheduler. This simulations use a level-1, cache-miss latency of 5 processor cycles and a level-2 cache-miss latency of between 10 and 1000 processor cycles, representing a range of memory architectures from tightly coupled through to distributed. First we show the performance of micro-threading on the K3 Livermore loop, which is an inner product calculation. The thread is a loop body comprising just 4 instruction, i.e. not much opportunity for parallelism, as each thread is dependent on the previous one. The 4 instructions are two load words, which will usually miss cache as no prefetching is assumed, a multiply requiring both loads which are independent in each thread and an add instruction which is dependent on the result of the multiply and the result of the add from the previous iteration. This is in executed with a thread-based vector instruction, which generates a family of threads for the entire recurrence loop. The conventional code would have 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 the micro-threaded pipeline using its vector thread create instruction.

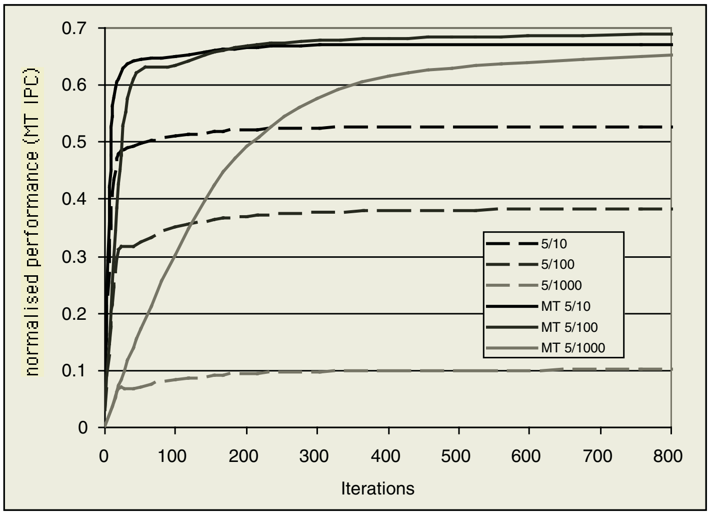

Fig. 1. Relative performance of micro-threaded (solid lines) vs conventional pipeline (dashed lines) on Livermore K3 loop. Each line shows a different cache delay e.g. L1/L2.

Figure 1 shows the performance of the micro-threaded pipeline for the Livermore K3 loop kernel. This shows that a micro-threaded pipeline achieves the same asymptotic performance (IPC = 0.7) regardless of the cache delay but requires more iterations to achieve it. For a 1000 cycle L2 cache miss penalty, the half performance vector length is 120 iterations. What is significant is that for 240 plus iterations, the micro-threaded 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!

This result assumes unlimited registers, which is an unreasonable assumption, the simulation was repeated with a fixed number of registers (128) and the results are shown in figure 2. For the 1000 cycle L2 miss penalty, performance is register limited and it is only marginally better than the conventional pipeline. Less than 32 iterations can run in parallel in this configuration, which is insufficient to tolerate cache misses of 1000 cycles and also has an impact on the 100 cycle L2 miss performance as well.

The final simulation we present here shows the scalability of the model as a chip multi-processor (CMP) with multiple instruction issue per cycle, based on a 1- 2- and 16-way CMP, each with a cache delay of 5 and 10 cycles respectively for L1 and L2 cache. This is shown in figure 3. In these results, 1 and 2-pipe simulations use 128 registers and the 16 pipe simulation uses 4096. The results show normalised performance, which is the IPC per pipe and this is plotted against iterations per pipeline, giving a normal form for each result. Ideally, with perfect scaling, all results should be coincident, which is just about what is observed. Admittedly the results are based on the K7 Livermoore loop, which is a parallel loop with no dependencies. However, peak performance has an IPC of just below 1 instruction per cycle per pipeline and the half performance vector length is about 5 iterations per pipeline, showing that at least a 95% utilisation can be achieved with this model even on issue widths of 16.

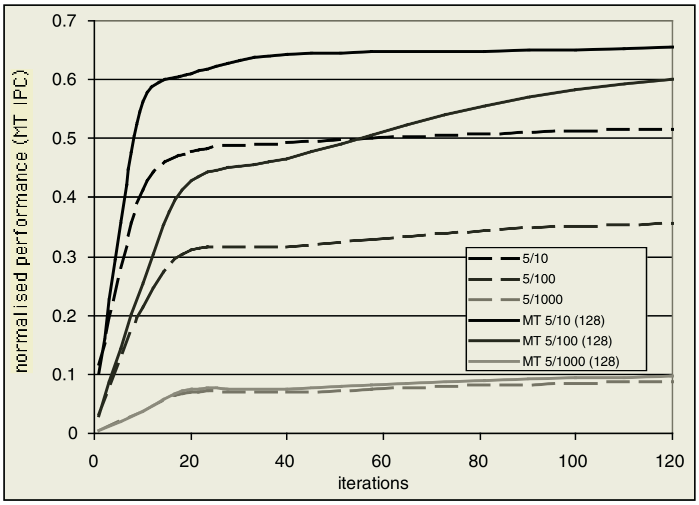

Fig. 2. Relative performance of micro-threaded (solid lines) vs conventional pipeline (dashed lines) on Livermore K3 loop with 128 registers.

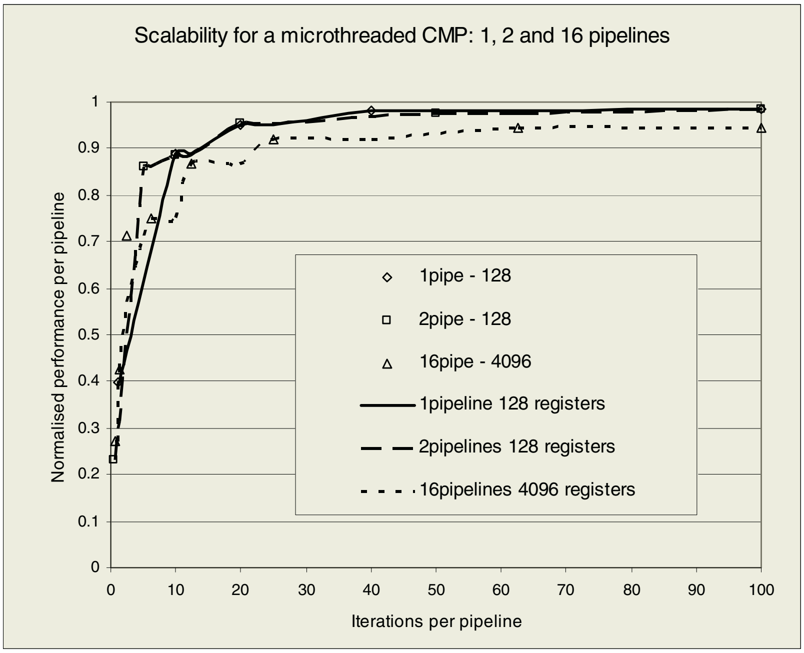

Fig. 3. Performance of a micro-threaded CMP of 1, 2 and 16 processors on Livermore K7 loop.

3.2 Micro-threaded Model

Now let us look at the model these results are based on in more detail in order to understand how and why these results are possible. Figure 4 shows a modified pipeline, with shared and duplicated parts indicated by shading. This is a very conventional in-order issue pipeline with micro-threading components added. These will be described as we outline the micro-threaded abstract model and its implementation.
The model supports a number of concurrent threads all drawn from the same context, these were called micro-threads to distinguish them from other multi-threaded techniques. The term micro-thread captures the notion of this approach, that of creating, interleaving and terminating very many small sequences of instructions efficiently, perhaps just a few machine instructions each. One disadvantage of micro-threading as proposed in [12] and shared in nano-threading [23] is that they both require the allocation of registers to threads at compile time. This is a major disadvantage for micro-threading where the aim is for general computation using ILP and data parallelism. Dansoft's nano-threaded approach has just two nano threads and allocation of registers is trivial, a subordinate thread would typically be used to preload values from memory into a registers for later use by the main thread.

The solution to this problem in micro-threading was reported in [24 and 25], which describe a dynamic register allocation scheme combined with a thread creation mechanism that produces families of threads, based on the same fragment of code. Without this solution, threading a number of iterations from a loop would require different instances of code with unique registers allocated to each instance. Using this approach, one thread-create instruction generates a family of threads across a loop-like triple that defines the start step and limit of the index value for each thread created. This is very similar to a vector instruction set. Each family of threads can iterate a loop concurrently to the maximum extent of resources available. Thread creation thus becomes a two-stage process:

- *stage one* creates a descriptor for a family threads, which waits until resources are available, and
- *stage two* allocates each thread in the family to a set of resources as they become available. The thread is now able to execute.

The resources required are a continuation queue slot and a contiguous set of registers defined by the thread header.
A major benefit of this dynamic allocation is that it supports a model that can trivially schedule work on multiple processors in a CMP. A potential problem is that the compiler must be aware of resource deadlock issues, for example an inter-thread dependency that spans more than the available chip resources.

A secondary problem introduced by dynamic allocation of registers is that of binding between allocated registers in dependent threads. In an inter-thread synchronisation, one thread will produce a value and another will consume it. In the micro-threading micro-archtecture, synchronisation is performed using full/empty bits on registers and the problem of binding between dynamically allocated registers is solved by allocating threads in strict sequence and by providing an offset in the thread header between producer and consumer within that sequence. This allows runtime

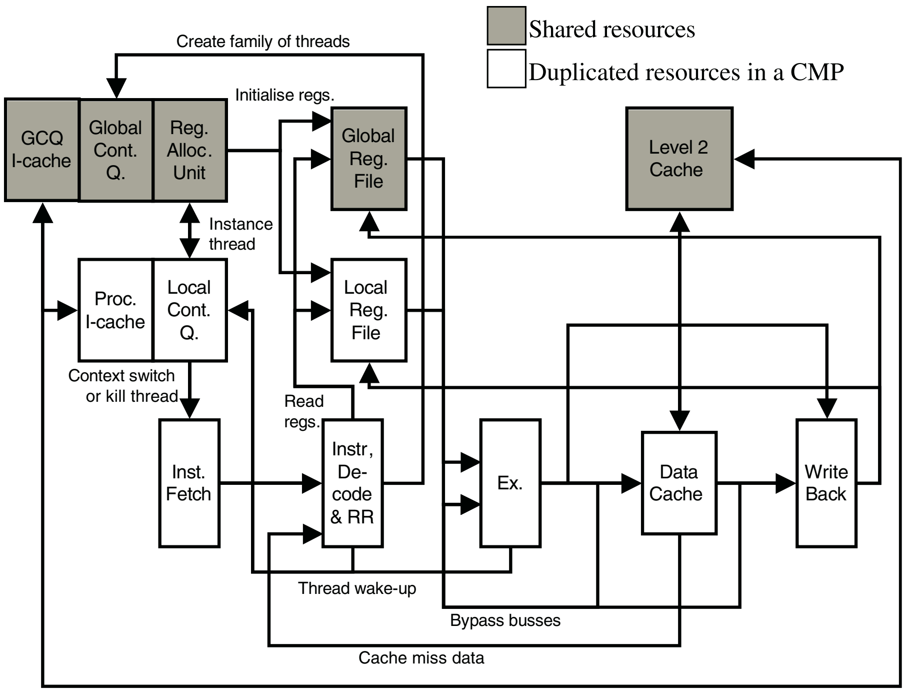

Fig. 4. CMP showing shared and duplicated resources for a wide-issue Multi-threaded architecture

structures to be maintained that allow the sharing of registers between threads, even if allocated to different processors. More detail on this is given in section 4.3.

3.3 Context-Switching Model

In [20], our micro-threaded model is classified as static, block-threaded and with explicit switching. This is because the compiler explicitly tags instruction where a context switch should take place. The context switching overhead is zero cycles, as the tagged instructions trigger an exchange of PC at the instruction fetch stage of the pipeline and then continue to execute. The next cycle sees the first instruction executed from the new context. Context switching can occur on every cycle if so tagged. Tagged instructions include conditional branches and any instruction that reads a register, where the compiler cannot guarantee that the data will be available. Deterministic delays can be compiled into sequences of instructions in the normal manner but for non-deterministic delays a context switch is signaled. There are two kinds of synchronisations, where a context switch is required. The first is intra-thread synchronisations where the result of an earlier load word is required and where there is no guarantee that the load hit the cache. The second is an explicit synchronisation between instructions in different threads, where one thread produces data and another consumes it.

A context switch will only occur when there is at least one other thread ready to run. In the case of single threaded code, or where all other threads are suspended or otherwise unable to run, the current thread will continue to issue instructions as there is a chance that the synchronisation will succeed. If it does, then we avoid a bubble in the pipeline while the thread is suspended passed down the pipeline and cycled back to the continuation queue on a successful synchronisation. If it does not succeed, the thread will suspend on the empty register and await its data and any subsequent instructions issued will be aborted. In this case, the instruction issue stage will have to wait for a thread to be made active before continuing. In the case of a branch instruction, it is possible to add branch predictors but on such a simple pipeline, this is probably not an optimal solution and we use branch delay slots in single threaded code.

Instructions are also terminated for thread termination by the compiler. This means that any instruction can be tagged as being the last in its thread. Both context switch and kill therefore are implemented at zero overhead as both overload otherwise useful instructions. A thread kill tag is similar to a context switch in that it forces a context switch as well as signaling the LCQ that this is the end of the current thread.

3.4 Synchronisation Model

Synchronisation between threads occurs on registers using a three-state model (full/empty/waiting). In the waiting state, the thread reference is stored in a previously empty register and awaits data before being rescheduled. Synchronisation between registers and memory can be added using full/empty states and this may be required in a massively parallel system. Memory synchronisation will cause the consumer thread to wait in the register while the memory system awaits synchronisation with another context. Of course higher levels of concurrency may require software scheduling mechanisms and the full state of the registers must be saved in this situation. The LCQ state can also be used to trigger software context switches, instead of having the pipeline idle.

The justification for using registers as synchronisers for micro threads is to provide a very low-latency synchronisation mechanism within a single context and this model requires that all registers in the micro-architecture implement a modified i-structure [26]. A successful synchronisation incurs zero overhead and recycles the suspended thread to a runnable state within just a few cycles (e.g. the number of cycles to the register read stage in the pipeline + 1, assuming an I-cache hit on rescheduling). Thread suspension occurs at the register read stage when a read is attempted on an empty register. In this case the instruction reading the register is transformed into an instruction that writes the thread reference into the empty register. To do this, the thread's reference travels down the pipeline with each instruction executed. A subsequent write to that register will extract and reschedule the thread whose reference is waiting there. In this way, neither suspend or wakeup require any additional pipeline stages and only a failed synchronisation will require an additional cycle to re-launch the incomplete instruction. The instruction that writes to a waiting register first reads and reschedules the waiting thread before writing to the register. An extra cycle is also required when a deferred memory access is made, as this must insert a new write back instruction into the pipeline, or when all both read ports are used in the instruction that writes the waiting register (e.g. and R op R -> R instruction). In this case a one cycle stall is required to extract the waiting thread reference or an additional register port is required.

Each register implements a modified i-structure that is allocated in the empty state on resource allocation. It has two operations i-store and i-read. I-store updates a specified register with a value and sets the register to the full state. Normally only a single i-store operation is allowed on a given i-structure but this single write is not enforced, to allow the registers to be used in a conventional manner if required. The i-read operation either suspends the thread containing it, if the register is empty, or it returns the value stored. Note that no further i-read instructions can take place on a register that contains a suspended thread. The compiler must therefore enforce binary synchronisations. If there is a requirement for multi-way synchronisation, i.e. many threads suspended on one event, the solution is to create a single guard thread that performs the synchronisation and then creates any number of other continuations. Note that the guard thread's only actions are to await synchronisation, to create a number of other threads and to terminate. This could require just two instructions with a vector create.

3.5 Subroutine Linkages

Micro-threading draws its concurrency only from within a single context and it relies on this fact to provide low-overhead concurrency controls for threads. Thus there must only be a single thread of control when performing subroutine linkages. The single persistent thread is called the main thread for identification purposes only. There are two general solutions to achieving this restriction in multiplicity of threads across subroutine linkages. The first is to make the concurrency user controlled, i.e. the compiler must generate instructions to synchronise to the main thread and to kill all other threads prior to a subroutine call or return. This can lead to large overheads in some programming paradigms, such as "winner-takes-all." Many synchronisations may be required to determine the winner and to signal this to all the other threads. The alternative solution, which is the one we prefer, is to provide a hardware imposed sequentiality across subroutine boundaries. This allows any thread to call or return and hardware cleans up any active threads and allocated resources as a part of the linkage.

As an illustration, assume that we link to a subroutine and the main thread creates a number of threads to search some space, each exploring a small part of that space. In our model any thread on gaining a solution can execute a return, which would kill all other active threads, relinquishing their resources in the process. It does not matter that we have redefined the main thread in this process, as a thread will use global state to communicate results.

3.6 Summary of ISA Requirements

In order to implement a micro-threaded ISA we have to add only four instructions to the base ISA. However, in order to make code more readable, we also add a number of pseudo instructions. The instructions added are:

- *Cre ref* - create a thread unconditionally where the long literal "ref" is a pointer or handle to the thread code;
- *Creq $a $b ref* - create a thread if the registers $a and $b are equal "ref" here is short and PC relative literal;
- *Crne $a $b* - create a thread if the registers $a and $b are not equal;
- *BSync* - suspends the current thread and awaits termination of all other threads before this thread continues.

In addition to the instructions above, three pseudo instructions are defined, which translate into executable instructions.

- *Wait $a* - waits for data in register $a and continues;
- *Setf $a* - signals register as full (n.b. the value in $a is undefined);
- *Sete $a* - signals register as empty.

Finally the compiler tagging of instructions for context switching is translated into pseudo instructions. Three distinct actions are encoded on any instruction requiring a two-bit extension field.

i. the next instruction comes from the same thread (*normal execution*);
ii. the next instruction comes from another thread, if one exists, otherwise from the same thread (*context switch*);
iii. the current thread is killed and the next instruction comes from another thread (*kill thread*).

In the original micro-threading paper these were called horizontal transfer, vertical transfer and kill respectively. Here we define them by pseudo instructions, which follow the instruction that they encode:

- *Swch* - switch context if any threads are waiting execution;
- *Kill* - kills the current thread.

Note that these instructions are not translated into executable instructions but simply encode the previous instruction with the additional action. For example:

```
add $a $b $c
kill
```

Generates one instruction, which performs a add operation and which is tagged to signal the IF stage to terminate this thread. The next instruction comes from another thread. Similarly:

```
mul $a $b $c
swch
```

Generates one instruction in the pipeline, which performs a multiply operation and is tagged to signal the IF stage to context switch. The next instruction comes from another thread but only if one is available.

4 Implementation Issues and Chip-Area Overheads

In this section we look at more details of implementation and compare the overheads of this model to the Alpha 21464 described in Section 2.1. Clearly the simulation results given above look very promising but what are the consequences on silicon real-estate and scalability in the micro-threaded model. In this comparison, we have adopted a MIPs-like ISA as a base architecture, implemented with a simple 5-stage pipeline, namely {Instruction fetch, Register read, ALU, Memory, Writeback}. A 5 stage pipeline is very simple by current microprocessor implementations. However, it illustrates the fact that much of the complexity of current microprocessors derives from the out-of-order issue and are simply not required by a micro-threaded pipeline, thus reducing its latency.

It is difficult to compare a micro-threaded pipeline to current practice in detail as that requires a detailed implementation of both. In this paper we look only at the instruction issue and register files and compare these to the out-of-order issue pipeline. Note that execution may not be optimal on the micro-threaded pipeline but optimisations, such as allowing 2-way (integer and floating point) in-order VLIW issue, would resolve any redundancy in execution units at a small additional cost. In both superscalar and micro-threaded architecture we are comparing wide instruction issue. In the case of the micro-threaded pipeline this is as an 8-way CMP. In the case of the 21464 it is an 8-way issue superscalar SMT extracting instructions from up to 4 threads.

4.1 Thread State and Register Allocation

In order to understand the implementation issues we must look at the state model of micro threads in some more detail. Table 1 shows the various states, events and representation of threads in the micro-threaded model.

The literal in the create instructions provides a pointer to the thread description block, which contains all of the parameters that define a family of micro threads. Figure 5 shows this data structure and also a schematic representation of the three major state changes from executing a create instruction to running the thread. The parameters are:

- the number of local and shared registers required by each instance of the thread {*local, shared*}
- a triple {*start, step, limit*}, which defines the number of instances of the thread and an index value for each
- the dependency distance {*dep*}, which links the consumer to the producer thread in the sequence of threads
- a pointer to the code for the body of the thread {*tp*}.

When a thread is created, the parameter block is copied into the GCQ, which is shared between all processors on a chip, see figure 4. The GCQ holds the abstract descriptions of all families of threads that have been created but not yet allocated to a proces-

Table 1. State transition table showing detailed state changes in the micro-threaded model.

Old state	Event causing transition	New state	Thread state stored in
Not-defined	Execution of: Cre I Creq I Crne	Created	Global continuation queue (GCQ)
Created	Resource availability on any processor	Active	Local continuation queue (LCQ)
Active	Thread is at head of LCQ and IF signals a context switch or kill	Running	LCQ + PC + LCQ slot no. in pipeline
Running	IF signals context switch	Suspended	LCQ + LCQ slot number in pipeline
Running	IF signals kill	Killed	LCQ
Suspended	Register read succeeds and instruction isn't a conditional branch	Active	LCQ
Suspended	Instruction at ALU stage is a conditional branch	Active	LCQ
Suspended	Register read finds one or more operands empty	Waiting	LCQ + LCQ slot no. in register
Waiting	Register written	Active	LCQ
Killed	Dependent thread is killed	not defined	All state is relinquished

sor. In each machine cycle an allocation will be attempted on each processor from one family of threads. Allocation requires a free LCQ slot on that processor and the required number of locals registers. In addition, for P processors, P times the number of globals must be allocated from the global register file. The result of the allocation is a set of base addresses, detailed below, the initialisation of the registers to empty, with the exception of the first local register, which is initialised to the loop count for that thread. Each allocated thread is uniquely identified during its lifetime by the tuple comprising its processor number and LCQ slot number.

4.2 Registers and Register Addressing

Register addressing uses a simple base + offset mechanism, where the base address is a part of a micro-thread's state and the offset is defined by the register specifier in the instruction executed. The ISA identifies four different types of register and a base address is required for each. Two bits of each register specifier define the register type, the remaining bits the offset. The register types are:

- *Global registers* - are allocated statically, stored in the global register file and read by any processor;

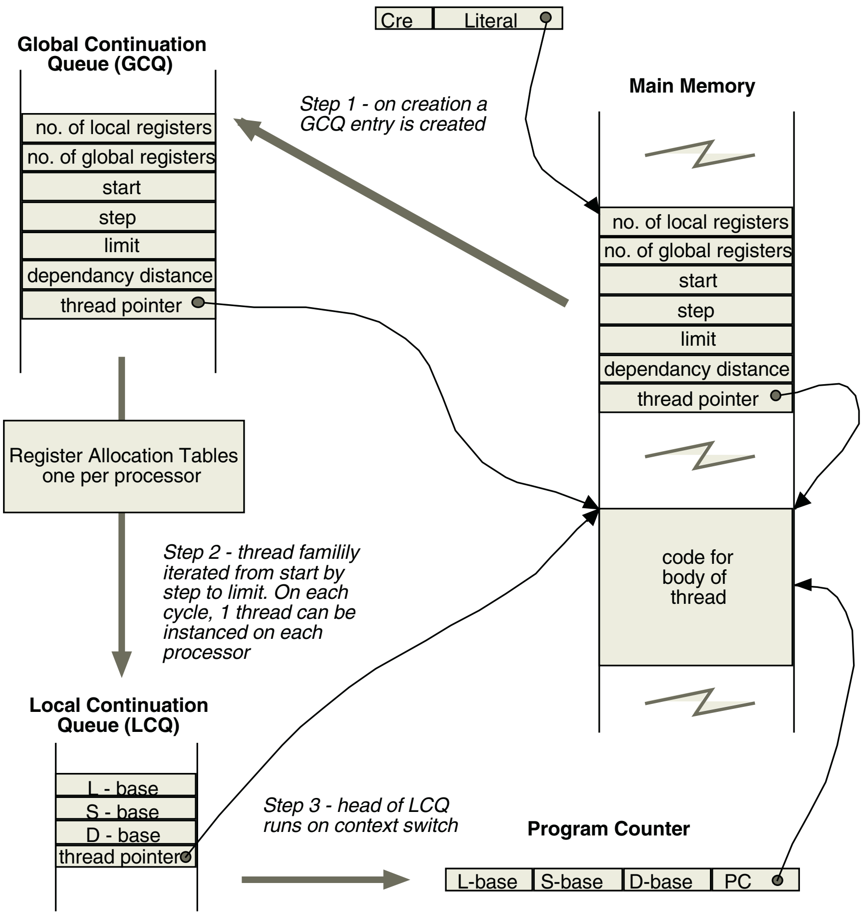
Cre
Literal
Global Continuation Queue (GCQ)
no. of local registers
no. of global registers
start
step
limit
dependancy distance
thread pointer
Step 1 - on creation a GCQ entry is created
Main Memory
no. of local registers
no. of global registers
start
step
limit
dependancy distance
thread pointer
Register Allocation Tables one per processor
code for body of thread
Step 2 - thread famlily iterated from start by step to limit. On each cycle, 1 thread can be instanced on each processor
Local Continuation Queue (LCQ)
L - base
S - base
D - base
thread pointer
Step 3 - head of LCQ runs on context switch
Program Counter
L-base
S-base
D-base
PC

Fig. 5. Data structures that define a micro-thread in its various states.

- *Shared registers* - allocated dynamically for each thread and stored in the global register file;
- *Dependent registers* - not allocated but refer to the shared registers of the thread this one is dependent on;
- *Local registers* - allocated dynamically for each thread in a processor's local register file.

The four different kinds of register in the ISA are identified in the assembly language by adopting a register specifier that uses $ followed by the first letter of the register type {G,S,D or L}, followed by the register number, e.g. $L0, is local register 0 and this is always initialised to the loop number.

We can immediately see some benefits of the micro-threaded approach over anout-of-order issue architecture. The micro-threaded ISA separates local and global registers. Each processor has its own local register file and these will only require 3 register ports in the implementation and hence the chip area required for local register files will be negligible compared to that the 21464 described in Section 2.1. Remember the 21464 has a single 512 register register file with 24 ports in total. It occupies an area some 5 times the size of the L1 D-cache, which is 64KB [6], a packing density hit of two orders of magnitude over the cache memory cell!

If we assume the same number of registers in the micro-threaded CMP as in the 21464, we would have 8 local register files of 32 registers each and a global register file of 256 registers. The issue we must resolve in order to compare the two is how many ports we require for the CMP's global register file. In our simulations [24, 25] we observed that on average, only two instructions in three read the global register file and only one in three write to it, even in threads which have a loop dependencies. Thus we can assume that the register file will require 9 ports, which more than matches the average number of hits required per cycle from 8 processors. We use this figure but note that it may result in some stalls due to conflict due to uneven load distribution and this is an issue we have yet to quantify in our simulation and one of the tradeoffs in any design.

If we assume the area of the 21464 is 1 and we assume a square law increase in register file area with number of ports and a linear increase one in number of registers, then the combined local register files of the CMP would require an area of only 0.8% of the 21464's register file and the CMP global register file would requires an area of 7% of the 21464's register file. There would also be a linear reduction in the area required for bussing data to and from the register files based on number of ports. So in the CMP we can reduce the area required for register file size by an order of magnitude in an 8-way issue processor but what of overheads for dynamic allocation of the registers.

4.3 Dynamic Register Allocation

The register allocation cycle is shown in figure 6. The Register Allocation Units (RAUs) maintain the allocation state of all register files. In each cycle registers are allocated on each processor's local register file and for each processor in the global register file. This is equivalent to 2 allocation units per processor, each maintaining 32 registers, where one allocates locals and one globals. In practice the global allocator will share some resources, so this is an upper bound.

Registers must be allocated in a contiguous block as we are using base + displacement addressing. If we assume a 3-bit displacement field, giving a 5 bit register specifier (i.e. 2 bits to select register type), then the maximum number of registers of one type that could be allocated to a thread is 8. The logic to implement an allocator is not complex. Even allocating to an arbitrary boundary in the register file would require little more than 1 bit of storage as an allocated flag, a 3-bit counter and a 3-bit comparator for each register. However, the area cost of the RAUs is linear in the number of registers with the constant being proportional to the displacement field, and moreover, it is small compared to the register file itself. There is an added complication, as we have to keep track of dependencies.

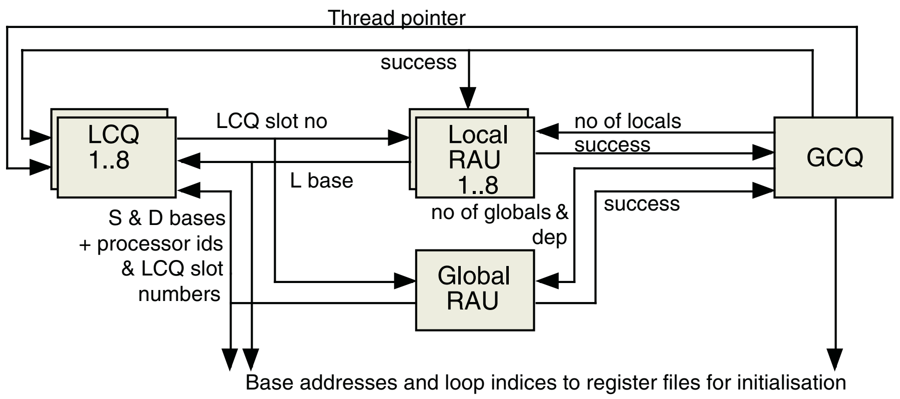

Fig. 6. Register allocation cycle showing the major components and interactions between them.

Synchronisation between two threads uses a shared register on which i-read and i-store operations can take place. If we call the thread performing the i-store the producer and that performing the i-read the consumer, then the shared register is allocated to the producer from the global register file and it is accessed using the shared base address, e.g. $S1. The consumer references the same register using the same offset but with the dependent base address, e.g. register specifier $D1, i.e. register $D1 in the consumer thread is the same register as $S1 in the producer thread. To achieve this dependency tracking we use the *dep* field from the thread header, which specifies the dependency distance between producer and consumer in terms of the thread issue sequence. The D-base of the consumer thread must be set to the same value as the S-base of the consumer thread. In order to locate and bind the producer and consumer threads in this way on a multi-processor chip, a number of rules must be followed in creating threads.

i. all thread families in the GCQ must be iterated in order of creation to various processors;
ii. the loop iterator must be defined such that the producer thread is allocated before the consumer thread;
iii. shared registers are allocated to all threads and a table of S-base, LCQ slot number and processor id are stored against the thread's sequence number modulo the maximum number of threads in the global RAU;
iv. the S-base, LCQ slot number and processor id of the producer thread are then determined from this table for each consumer thread using its sequence number minus dep, and the producer's S-base is copied and becomes the consumer's D-base.

Note that the processor id and LCQ slot number of the producer thread are required by the consumer thread's LCQ to signal the producer's LCQ when it has been killed, as only then can the producer thread's shared resources be released. If we assume that each LCQ has 64 slots, enough to have one thread waiting in each of the CMP's 512 registers, then keeping track of dependencies requires another 13 bits of storage per register and again is linear in the number registers in the CMP.

We have seen therefore that register allocation, including keeping track of dependencies has a chip area, which is linear in the number of registers in the CMP and which has a small constant. The area of this is negligible compared to global register file, which is dependent on both number of registers and number of ports squared.

4.4 LCQ, Thread State and I-Cache Prefetching

The LCQ is perhaps the major overhead associated with the micro-threading model in terms of chip area. We have already assumed that the number of threads in the CMP is equal to the total number of registers available. It cannot reasonably be more, as blocked threads wait in registers but it might be less. So we are looking at an upper bound here.

The LCQ is a linked memory structure and associated logic that implements a given thread priority, probably a FIFO, as our simulations have shown that scheduling priority has little bearing on the efficiency of execution [25]. This is hardly surprising due to the fine-grain nature of the threads involved. The LCQ is implemented as a store addressed by thread reference or slot number, which has two 6-bit fields for creating various priority queues. It requires a 3-bit field for thread state and a field, which points to the producer thread of any dependency, which may be on any processor and hence requires 9 bits. Finally it requires a thread pointer (PC), which we assume is 40 bits, giving a total of 64 bits per register in the CMP. The memory is likely to be multi ported but with a small number of ports, we estimate 4 read and 4 write ports as the LCQ interacts with RAU, pipeline, I-cache. This would mean the combined LCQ structures in the CMP are approximately equal to its global register file size, which we know is approximately 7% of the area required by the 21464's register file. What is important however is that it scales linearly with the issue width. The number of ports required is a structural and implementation issue and is independent of the issue width.

We estimate therefore, that the combined LCQs in an 8-way issue micro-threaded CMP would be about 3% of the size of the 21464's instruction issue logic.

There are more benefits and area savings in a micro-threaded model if we consider the LCQ's interaction with the I-cache in more detail. This can be used to avoid stalls in the pipeline due to I-cache misses. The state of a micro thread can be used to determine a prefetch and replacement strategy and conversely the state of I-cache lines can be used to set the thread state so that pipeline stalls on I-cache fetch can be avoided completely. The prefetch/replacement strategy is deterministic and very simple, each I-cache line only requires a counter of the number of active threads using that cache line. When a thread is allocated to a processor, a prefetch will be made to its thread pointer. If the prefetch hits, the line counter is incremented and the thread becomes active. If it misses, the memory block will eventually be fetched into any line with a zero counter. Until this happens the thread remains in a suspended state. When a thread enters the running state additional blocks may also be fetched to avoid I-cache stalls. Remember however, that any conditional branch will normally suspend the thread and we can ensure that the thread is not made active until the I-cache block along the new path has been fetched. A running thread either runs until it is killed or is suspended and in either case its code is no longer required in the I-cache and the I-cache line counter is decremented. Cache replacement therefore, is based on thread counters. Any line with a counter of 0 can safely be swapped and the cache will wait until this condition is eventually reached, as earlier threads suspend or are killed. When a thread is rescheduled after being suspended, the same process is followed as when it was created, i.e. it remains in the suspended state until the code in the cache. The overhead for this strategy is trivial, just a 6-bit counter and some associated logic per cache line.

5 Programming Model

Before we draw conclusions from the above analysis let us consider the programming model that might be used with a micro-threaded microprocessor. There are three issues here, binary code compatibility, sequential language compilation techniques and finally a concurrent programming models. These are each briefly discussed.

Concerning binary code compatibility, we have already said that only a small number of additions need be made to a base ISA in order to support the micro-threading model. We have also said that register specifiers and instruction tagging must be supported. It would be possible to use binary code translation to support backward compatibility by not using any of the additional instruction and by tagging everything to not context switch. This would leave us with single threaded code, which would not exploit the wide issue of a CMP. It is possible to develop techniques to do binary code analysis and create threaded code from sequential code by analysing dependencies in the instruction stream. This approach has not yet been studied in any depth.

If binary code compatibility is discarded it is possible to compile existing sequential source code to generate very efficient micro-threaded code. This is because the compilation can extract concurrency across loops, even in the presence of inter-loop dependencies. These techniques have been used in order to hand compile the code used in our simulations [24,25]. There are limitations on the complexity of loops that can be supported by a family of micro threads, because the synchronisation in this model requires a single constant dependency distance between loop iterations. For example, the two code fragments below could both be compiled in a single family of threads (vector instruction):

```
For i = 1 to n do
    x[i] = x[i] + x[i+j]
For i = 1 to n do
    x[i] = x[i] + x[i+j]
    y[i] = y[i] + y[i+j]
```

If j is a constant then the thread header is static, if j is a variable, the thread header would need to be constructed dynamically.

The following fragments however, could not be compiled to a single family of threads (unless k = j).

```
For i = 1 to n do
        x[i] = x[i] + x[i+j]
        y[i] = y[i] + y[i+k]
```

It could however be translated into two families of threads, with a global synchronisation before the creation of the second family, where each family performs just one assignment from all loop iterations. The overhead for this would not be large, just a few cycles amortised across n iterations.

New techniques need to be developed to fully understand the code generation issues but the basic code analysis is well understood and it comes from dependency analysis found in standard optimising compilers and vectorisation techniques used in compilers for vector supercomputers.

The final method of programming we consider is that for which this execution model was originally designed for [27], namely an explicit data-parallel model [28]. Such languages provide a simpler analysis and more information for optimisation in terms of the symmetries that they possess [29] and hence give us an easier route to generating efficient code, than in compiling sequential languages.

Note that loop parallelism other than indexed loops can be compiled with the micro-threaded model, including speculative techniques but these techniques tend to be marginally less efficient as they typically require one create instruction for every thread created rather than one instruction per family. Obviously the extent of the inefficiency is smaller for longer micro threads. These are all issues that are being considered in our current work but we have already simulated a pointer chasing loop based on a while loop and even with sequential order termination it provides better performance than a conventional single-issue pipeline.

6 Conclusions

It appears that Moore's law is a two-edged sword. The exponential growth of on-chip resources for storage and processing has tended to mask creeping inefficiencies in current microprocessor designs, such as out-of-order issue microprocessors, including those with multi-threaded instruction issue units. An exponential increase in gates with a short time constant can easily hide the underlying lack of scalability in any given approach. The problem is, that there are strong commercial pressures for an evolutionary development in microprocessor design. However, the fundamental scaling issues that have been highlighted are always going to be an issue at some stage, unless no other approach can be found with better scaling properties, and then a revolutionary change will be required. This has already happened with both radical and conservative microprocessor vendors but on different time scales (note for conservative read market leader).

The fundamental issues in out-of-order issue microprocessors are in the complexity of two main components in this design, namely the instruction issue logic, which grows as the square of the instruction window size (proportional to the issue width) and the register file, which is used for global communication and synchronisation between the concurrently issued instructions, and which grows with the square of the number of register ports (proportional to issue width). What is required to alleviate these problems is an instruction issue strategy that is linear in the issue width and a register file that is partitioned between local and remote synchronisations. Note that the global nature of communications is always going to grow with a square law if we want constant delay, as this is a connectivity problem. Thus the only option we have open to us here is to partition the register file into local and global parts, thus mitigating the square law scaling. Micro-threading does exactly this in a CMP.

This paper has therefore looked at multi-threading as an alternative approach to out-of-order issue. It investigates the diversity of multi-threaded design in achieving large instruction issue widths. Among these options we have shown micro-threading to be a particularly efficient form of multi-threading. We show this technique to be very effective in tolerating latency and in the avoidance of speculative execution. Moreover, it can extract concurrency from both ILP and loop-level parallelism by supporting a vector like instruction to generate families of threads for executing loops. Thus micro-threading can be used as the basis for wide instruction issue in a chip multi-processor.

The instruction issue mechanism distributes work between processors in the form of micro-threads that execute just a few instructions. This fine grain distribution enables a very even distribution of load, one of the factors, which determines the efficiency of a parallel system. The example simulated here on a micro-threaded CMP uses the Livermoore K7 loop and our compilation generates a family of threads, each of which comprises just a few instructions, all of them performing useful, rather than bookkeeping work. The second factor, which determines the efficiency of a parallel system is the overhead incurred in scheduling and synchronisation the threads. It is clear from this simulation that the multi-issue, micro-threaded CMP has negligible overheads. It achieves an asymptotic IPC of between 95 and 100 percent per pipeline, for up to 16 pipelines, demonstrating what has been said in previous papers [12,24,25], namely that the model has very small overheads for scheduling threads both on single and multiple issue pipelines. The half-performance vector length for achieving this is also just 3 iterations per pipeline but that would be expected for a highly parallel loop with no dependencies.

Finally we have considered the overheads of implementing the instruction issue logic and register files for a CMP, both are components that we know in an out-of-order issue pipelines are not scalable. In the CMP, we have shown that instruction issue scales linearly with the number of pipelines and we compared this with out-of-order instruction issue, which is responsible for the single largest component on the 21464 (with the exception of the L2 cache). We have also shown that the register file can be partially distributed to individual pipelines, again giving linear scaling, for the local parts. However, micro-threading also requires low-latency, inter-processor synchronisation and this is achieved in the model using a global register file. In the CMP however, many register references will be routed to the local register files and the number of concurrent accesses to the global register file will reduced. Hence we can reduce the number of ports in the global register file, which is the root cause of the bad scaling. This component is not scalable in the CMP but it is more scalable than in an out-of-order issue microprocessor. We estimate that a reduction in chip area for the global register file in a micro-threaded, 8-way CMP of 93% compared to an out-of-order issue pipeline of the same width. This is based on one micro-threaded instruction in three writing a word to the global register file and two in three reading a word from it.

References

1. Jessica Tseng and Krste Asanovic (2003) Banked multiported register files for high-frequency superscalar microprocessors, To appear, 30th International Symposium on Computer Architecture (ISCA-30), San Diego, CA, June 2003, http://www.cag.lcs.mit.edu/scale/papers/bankedreg-isca2003.pdf
2. K Skadron, P S Ahuja, M Martonosi and D W Clark (1999) Branch prediction, instruction-window size and simulation techniques, IEEE Trans. Comput., 48(11) pp 1260-81
3. D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report 93/6, Digital Western Research Laboratory, November 1993.
4. R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 Microprocessor Architecture. 1998 International Conference on Computer Design, pages 24–36, October 1998.
5. V Agarwal, H S Murukkathampoondi, S W Keckler, and D C Burger (2000) Clock rate versus IPC: The end of the road for conventional microarchictectures, Proc 27th International Symposium on Computer Architecture (ISCA), June, 2000.
6. R P Peterson et. al. (2002) Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading,ISSCDigest and Visuals supplement.
7. W R Wittamayar (1978) Array processor provides high throughput rates, Comput. Design, 17 (3), pp93-100
8. Intel, (2000) Intel IA64 Architecture Software Developer's Manual, Volume 1-4.
9. D M Tullsen, S J Eggers and H M Levy (1995) Simultaneous Multithreading: Maximizing On-Chip Parallelism. ISCA 1995: 392-403.
10. G M Papadopoulos and K R Traub (1991) Multi-threading: a revisionist view of dataflow architecture, Computations Structures Group memo 330, March 1991, MIT.
11. K Sankaralingam, R Nagarajan, H Liu, C. Kim, J Huh, D Burger, S W. Keckler, C R Moore (2003) Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture, to appear Proc ISCA 2003, San Diago, June 2003.
12. A Bolychevsky, C R Jesshope and V B Muchnick, (1996) Dynamic scheduling in RISC architectures, IEE Trans. E, Computers and Digital Techniques ,143, pp309-317.
12. B J Smith (1978) A pipelined shared-resource MIMD computer, IEEE Proc. 1978 Intl. Conf. on Parallel processing, pp6-8.
13. M Thistle and B J Smith(1988) A processor architecturefor Horizon. In Proceedings of the Supercomputing Conference (Orlando, FL). 35–41.
14. R Alverson, D Callahan, D Cummings, B Koblenz, A Porterfield AND B J Smith (1990) The Tera computer system. In Proceedings of the 4th International Conference on Supercomputing (Amsterdam, The Netherlands). 1–6.
15. R German, M GIiampapa, D Gresh, M GUupta, R Haring, H Ho, P Hochschild, S Hummel, T JOnas, D Lieber, G Martyna, U Brinkschulte, C Krakowski, J Kreuzinger and T Ungerer (1999) A multithreaded Java microcontroller for thread-oriented realtime event-handling. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (Newport Beach, CA). 34–39.

16. M Tremblay J Chan S Chaudhry A W Conigliaro and S S Tse (2000) The MAJC architecture: a synthesis of parallelism and scalability, IEEE Micro 20, 6, 12–25.
17. P N Glaskowsky (2002) Network processors mature in 2001 Microproc. Report. February 19, (online journal).
18. Intel Corporation (2002) Intel Internet exchange architecture network processors: flexible, wirespeed processing from the customer premises to the network core. White paper. Intel, Santa Clara, CA.
19. IBM Corporation (1999) IBM network processor, Product overview. IBM, Yorktown Heights, NY.
20. T Ungerer, B Robic and J Silc (2003) A survey of processors with explicit multithreading, ACM Computing Surveys (CSUR) March 2003 35(1), pp29-63.
21. K M Kavi, D L Levene and A R Hurson (1997) A non-blocking multithreaded architecture. In Proceedings of the 5th International Conference on Advanced Computing (Madras, India). pp171–177.
22. P Marcuello, A Gonzales and J Tubella (1998) Speculative multithreaded processors. In Proceedings of the 12th International Conference on Supercomputing (Melbourne, Australia) pp77–84.
23. L Gwennap (1997) DanSoft develops VLIW design, Microproc. Report 11, 2 (Feb. 17), 18–22.
24. C R Jesshope (2001) Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines, Proc. ACSAC 2001, Australia Computer Science Communications, 23, No 4., pp80-88, IEEE Computer Society (Los Alimitos, CA, USA), ISBN 0-7695-0954-1.
25. B Luo and C R Jesshope (2002) Performance of a Micro-threaded Pipeline, in Proc. 7th Asia-Pacific conference on Computer systems architecture, 6, (Feipei Lai and John Morris Eds.) Australian Computer Society, Inc. Darlinghurst, Australia, ISBN ~ ISSN:1445-1336 , 0-909925-84-4 , pp83-90.
26. Arvind and Thomas, R.E.,"I-Structure: An Effective Data Structure for Functional Languages" MIT,LCS- TM178, Lab. for Computer Science, MIT, 1978.
27. A Bolychevsky (1994) The fundamental Issues and Construction of a Data-parallel Dataflow computer, Technical Report. Computer Systems Research Group, University of Surrey.
28. C R Jesshope (1982) Programming with a high degree of parallelism in FORTRAN, Comp. Phys. Comm., 26, pp237-246.
29. A V Shafarenko (1995) Symmetries in data parallelism Computer Journal, 38, pp365-378.