

# Scalable Instruction-level Parallelism

Professor Chris Jesshope<sup>1</sup>

Department of Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ,  
Amsterdam, The Netherlands  
bossman@mac.com

**Abstract.** This paper presents a model for instruction-level distributed computing that allows the implementation of scalable chip multiprocessors. Based on explicit microthreading it serves as a replacement for out-of-order instruction issue; it defines the model and explores implementation issues. The model results in a fully distributed implementation in which data is distributed to one register file per processor, which is scalable as the number of ports in each register file is constant. The only component with less than ideal scaling properties is the switching network between processors.

## 1 Some Issues in Current Microprocessor Design

Over the last twelve years Moore predicts a packing density increase of 256 in silicon die with a corresponding speed increase of 16. Whereas we see speed increases better than predicted, the same is not true of system-level concurrency. The history of the PPC processor (see <http://www.rootvg.net/RSmodels.htm>) shows that clock speed has increased at twice the predicted rate, i.e. from 33Mhz to 1Ghz but that increases in system-level concurrency do not track packing density. The PPC microprocessor has evolved from a 32-bit single-issue to a 64-bit, five-way issue design. It is difficult to obtain high IPC in out-of-order issue microprocessors and so a factor of two in effective issue width and a factor of two in bit-level concurrency is we get from this 256 increase in packing density. It seems that our linear minds can not differentiate between  $e^t$  and  $e^{3t}$  as they both seem discontinuous to us.

The faster than predicted clock speed is due to a finer pipeline division. The smaller than predicted concurrency is simply because we do not understand concurrency well enough, especially at the instruction level. There are also many factors contributing to the problem; for example the need for binary code compatibility. Looking at modern microprocessors, more and more area is being used for on-chip memory to mitigate against the high latencies of off-chip access, typically 25-33% of the chip area[1]. Also instruction issue logic is becoming more and more complex and typically takes as much silicon area as the on-chip memory. Instruction issue complexity is due to a poor scaling of the out-of-order approach, which grows with at least the square of the issue width[2]. Finally the register file is not scalable[3]. Register file capacity is related to issue width and the cell size to the square of the issue width. This means that as issue width

is increases, there is a cubic scaling of register file area, which will very quickly dominate chip area, speed and power considerations.

All three issues can be improved significantly. If a microprocessor can tolerate high latency, then it is possible to replace large on-chip caches with more processors. It will also be shown that instruction issue and register file area can be designed to be scalable. The solution requires explicit concurrency controls and distributed synchronisation to be added to an ISA and these in turn require a move away from binary-code compatibility. Backward compatibility can be retained at some cost in efficiency but recompilation or binary translation of legacy code is required to obtain speedup.

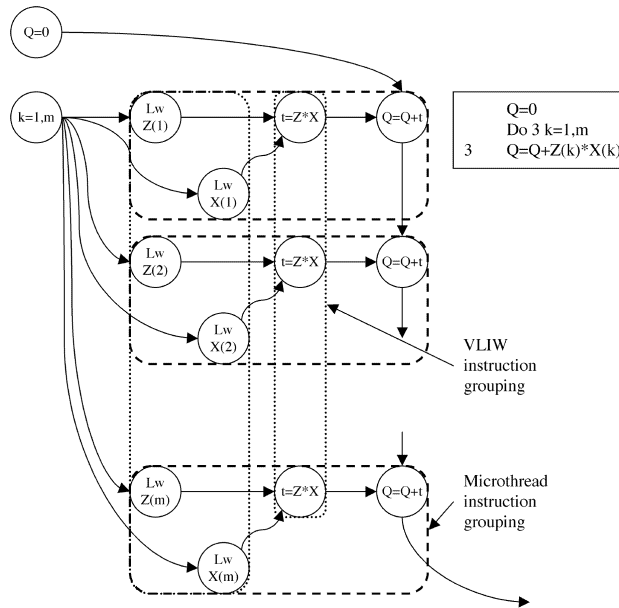


Fig. 1. Dependency graph for FORTRAN DO loop.

## 2 The Concurrency Model and its Restrictions

### 2.1 Introduction to the Model

The model proposed in this paper is one based on the concept of micro-threading. It uses multiple threads within a single context and provides a schedule-independent description of concurrency. VLIW or the more abstract EPIC also define explicit concurrency but over a fixed schedule of instructions, giving problems with non-deterministic timing, as any delay in one operation delays the entire schedule.

Figure 1 shows a dependency graph for a simple FORTRAN loop illustrating how instructions are grouped in VLIW, where each group is issued simultaneously and must complete before the next is issued. EPIC differs from VLIW in that it could encode the schedules shown, whereas VLIW would sequentialise these packets into issue-width units.

Micro-threading groups instructions in sequences, where each group is called a *micro-thread* and the collection of groups, is called a *family of microthreads*, in this case over all iterations in the FORTRAN loop. Instructions in different threads can be executed concurrently in any schedule subject to data dependencies; they can be interleaved arbitrarily in a single pipeline to give latency tolerance or can be executed on multiple pipelines in a chip multiprocessor. Instructions issue in a single thread is sequential with the overall schedule determined by data availability. Micro-threads are short sequences of instructions created dynamically with their own register window, which execute and terminate on the processor they are allocated to.

## 2.2 Background

The microthreaded model was first proposed in 1996 [4] and has been refined and evaluated over time[5]. A number of other papers have also considered similar models, the earliest being nano-threads in [6], a limited form of microthreading using only two contexts to tolerate memory latency. There is now a relatively large body of similar work describing the usage of threads for pre-fetching and tolerating memory latency, e.g. [7] and [8].

## 2.3 Concurrency Controls

Micro-threading can be based on any ISA and can be designed to maintain full backward compatibility, allowing existing binary code to run without speedup. Alternatively, binary-to-binary translation can be used to obtain backward compatibility with a more efficient implementation and potential speedup. For backward compatibility, just five instructions are required to implement the concurrency controls. These are:

- an instruction to create a family of threads **Cre**
- an instruction to effect a context switch **Swch**
- an instruction to terminate a thread (and invoke a new context) **Kill**
- a instruction that waits for all other threads to terminate **Bsync**
- an instruction to terminate all other threads **Brk**

Using these instructions, the FORTRAN code in figure 1 can be translated into the microthreaded code shown below, where the main thread creates a family of *m* threads to execute the loop, defined by the *create control block* at the label **do3**. It should be noted that this family carries a dependency defined by the scalar variable *Q*, from the main thread in **\$\$0** via the **\$\$0/\$D0** pair in each thread, through to the last iteration, where the value of *Q* is written to memory.

## IV

The dependency distance from the create control block determines how these pairs of registers are bound over the family of threads.

```

do3:      .data          # create control block
         .word 1        # start index
         .word m        # last index
         .word 1        # skip between indices
         .word 1        # distance between dependencies
         .word 2        # number of $L registers per thread
         .word 1        # number of $S registers per thread
         .word body     # pointer to the code
         .word last     # optional code for last thread (used)
main:    mv $S0 $G0     # initialise Q to zero for first thread
         cre do3       # create family of threads for loop
         bsync        # wait for threads to complete
         finish      # done!
body:    lw $L1 Z($L0)  # for all but last index value do
         lw $L2 X($L0) # load two values - $L0 contains index
         mul $L1 $L1 $L2 # multiply
         swch        # context switch as loads may miss cache
         add $S0 $D0 $L1 # add result to value from previous thread
         kill       # terminate thread
last:    lw $L1 Z($L0)  # last is as above except...
         lw $L2 X($L0)
         mul $L1 $L1 $L2
         swch
         add $S0 $D0 $L1
         swch
         sw $S0 Q($G0) # ...it stores the result to Q
         kill

```

Note that the index value is written to the first local register for a thread when it is created ( $\$L0$ ). A number of local and shared registers (also defined in the control block) are allocated dynamically when the thread is created so long as there are sufficient registers on the target processor. Instruction interleaving in a pipeline is defined by explicit context switching between the threads allocated to it and is required after an instruction that has a data dependency on any of its operands. In this code there are two examples, using the result of the `lw` instruction, which is dependent on a cache hit and reading the  $\$D0$  register, which is dependent on a previous thread writing the corresponding  $\$S0$  register. Explicit context switching at the instruction fetch stage avoids flushing the pipeline at the register read stage if either operand is found to be empty. When a register read fails and the pipeline is executing instructions from another thread, the micro-context of the instruction is stored in the *empty* register to suspend that thread. When data is written to the register, that context is reactivated. This action implements synchronisation as a blocking read between threads or between a thread and the memory system.

Because the model is defined on a single context, transfer of control between functions results in a single thread of control and threads other than that executing the call or return are killed. The synchronisation model works only at the instruction level and there is no synchronisation on memory. Concurrent writes to memory must be controlled by synchronisation at the register level. In the example, the loop-to-loop dependency is carried in registers and only written to  $Q$  in the last thread. If every thread wrote  $Q$ , the order of execution of the threads

might result in the wrong value being stored in memory. Finally, a dependency distance of zero defines a family of independent threads, which have no dependency relation between them nor to the thread that created them. A non-zero dependency distance defines a chain of dependencies that follows iteration order with thread  $i$  being dependent on thread  $i-d$ , where  $i$  is the  $i$ th thread created and  $d$  is the dependency distance defined in the create control block.

## 2.4 Register Partitioning

The register namespace in the microthreaded model is partitioned logically according to the type of communication being performed. An implementation would map each partition onto an appropriate register file and associated access mechanisms. The microthreaded model identifies four classes of variables, which are:

- *invariants*, which give rise to broadcast communication. Such variables are global and can be read and written by any thread. They are represented by a specifier  $\$Gi$  in the assembly language;
- *dependencies*, which give rise to pair-wise communication between two threads. They are written by one thread and read by another. They are called Shared in the producer thread and represented by a specifier  $\$Si$  and are called Dependent in the consumer thread and represented by a specifier  $\$Di$ . It should be noted that  $\$Si(\text{producer})=\$Di(\text{consumer})$ ;
- all other variables are called Local and used for communicating data within a single thread or between the memory system and that thread. These are represented by  $\$Li$ .

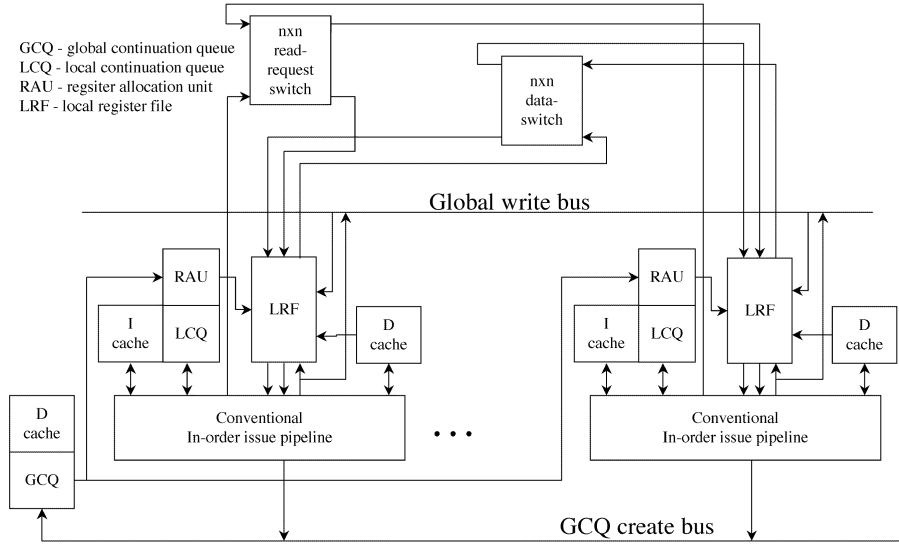
Each class of variable is allocated as a logical register window as mechanisms for access to each will differ. The state of a thread includes information to locate the various register windows in the register file(s).

## 3 Implementing the Model

A recent article [9] reviewed billion transistor architectures, seven years on from a special issue on the same topic. It highlights two areas where current approaches have failed to provide solutions to the trends in chip architecture. The first and perhaps the most important is the shrinking percentage of the chip that is reachable within a single clock cycle, this has become more critical with the superclocked processor cores that are now being designed. It mandates a partitioning that allows multiple, independent clocking zones that communicate asynchronously. The other factor is that of power dissipation and any implementation of a chip multiprocessor must address both of these issues.

Figure 2 shows a distributed implementation of the microthreaded model. In this architecture, the GCQ iterates the *create control block* to individual processors, where registers are allocated and state is created for the threads. Thread state is stored in the LCQ and this and the register files are fully distributed. A

single bus provides access to the GCQ for creating threads. Although only one processor may access this bus in any cycle, analysis of code in [10] shows that thread creation is a low-frequency event. The GCQ iterates the  $m$  iterations, to  $n$  processors so that each processor has at most  $\lfloor \frac{m}{n} \rfloor_c$  threads allocated to it. The iteration proceeds so long as all processors have LCQ slots and registers available, otherwise iteration waits until resources are released. Registers are allocated by the RAU using parameters passed to it by the GCQ, and those registers are initialised to empty. The first of the  $\$L$  registers is also written with the thread's index value. Threads are scheduled from the LCQ to the local pipeline only when the required code is in the I-cache, so no I-cache miss stalls will occur on a context switch.



**Fig. 2.** A distributed implementation of the microthreaded model.

Each register in each LRF implements an i-structure having two operations, i-store, which sets the register to the full state and writes data to it and i-read, which, if the register is full, reads the data stored otherwise saves a reference to the thread reading it. The i-read instruction must therefore reactivate any suspended thread it finds waiting there. These operations requires a two-bit overhead on each register to encode the full, empty and waiting states, and a state machine controller for each port of the register file.

The model defines three different types of communication specified over the four classes of register window, where each must have an access mechanism that can be mapped onto this distributed implementation. It is trivially easy to distribute the local register window, as accesses to it are from only one thread

but there is still a requirement for synchronisation. It is possible for both ALU and memory system to deliver data in an unscheduled manner, e.g in an iterative operation or a cache miss. Distributing the global window is not so simple, as it is shared between all threads. As any thread may write to this window an implementation must include arbitration to resolve this. The implementation shown, replicates the global window in all processors and provides a single *global write bus* with arbitration between processors for writes to this window. Reads then become local and require no special implementation. Writes occur locally as normal but get reflected in the other processors asynchronously.

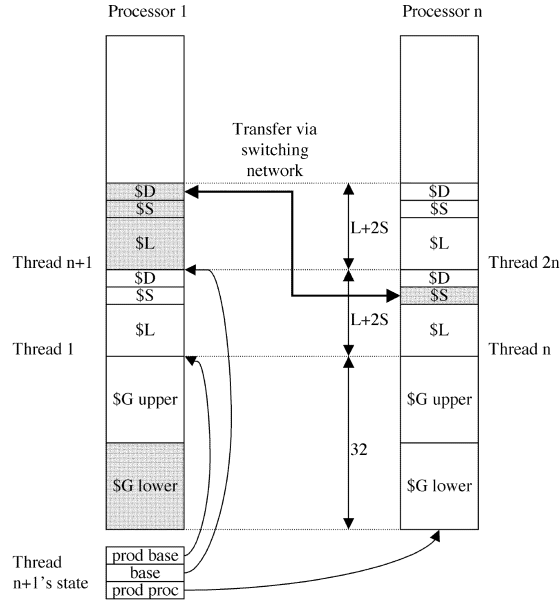
The final question to be asked is whether the shared and dependent windows can also be distributed to local register files and the answer to this will depend on the number of concurrent reads to these registers. If this number grows with the number of processors, there is nothing to be gained from distributing these windows. The  $\$S$  and  $\$D$  windows allow only pair-wise communication, the  $\$S$  window is in the namespace of a single producer thread and the  $\$D$  window is in the namespace of one or more consumer threads. A  $\$D$  window is mapped onto the  $\$S$  window by the dependency relationship between the threads. Many-to-one mappings between  $\$D$  and  $\$S$  windows are possible but the multiple threads that define these  $\$D$  windows cannot concurrently read the same location in their respective  $\$D$  windows, as this would violate the i-structure limitation, of holding only a single continuation. This restriction must be enforced by the compiler. What it means is that the number of concurrent reads to the  $\$S/\$D$  window is bounded above by the range of the register specifier and this provides a model constraint that allows distributed implementation. Note that the  $\$D$  and  $\$S$  windows may be allocated to different processors so thread state must also contain the processor id of the producer to its  $\$D$  window.

Reads and writes to the  $\$S$  window are no different to the  $\$L$  window, as they are local to that thread, but reads to the  $\$D$  window must trap to a special access mechanism to the producer's  $\$S$  window. It is important to decouple the local pipeline from this potentially remote register read and this requires the thread reading the  $\$D$  register to context switch and be suspended locally, which in turn requires that the  $\$D$  window, conceptually just a mapping onto the producer's  $\$S$  window, be physically allocated to the consumer thread. Each register location in the  $\$S/\$D$  window thus becomes a shared pair, connected asynchronously by the switching network. A read to  $\$D$  suspends the thread locally but only if the location is empty. This initiates a read to the potentially remote  $\$S$  register, where the request may itself suspend waiting for data. Any network or synchronisation delay is tolerated by executing other microthreads, given sufficient local concurrency. Note that after data has been delivered, any subsequent read to the same location becomes local, so multiple reads to a shared variable incur only one remote access.

### 3.1 Allocating Registers

The abstract model partitions the registers at the ISA level in order to differentiate different access mechanisms, namely local, broadcast and pair-wise com-

munication. An implementation requires this partitioning to identify the special access requirements. There are two cases that require special treatment. The first is a write to the global window, which must trigger an arbitration for the global write bus. Note that a local buffer can avoid any processor stalls due to peaks in traffic. The second is a read from the  $\$D$  window, which, on finding an empty register, must trigger a read to the corresponding  $\$S$  window wherever it is located. If it is local, it will cause an access to the producer thread's  $\$S$  window locally, if it is remote it will do this via the switching network on a remote processor. Note that the base address for the producer thread must be available to the consumer thread in all such binary communications, regardless of the location of the producer thread.



**Fig. 3.** Register allocation showing the state of thread  $n+1$ . Note that thread  $n+1$  is dependent on thread  $n$  and all registers accessible to thread  $n+1$  are illustrated by shading.

Using a conventional RISC ISA as a basis for the implementation gives a five-bit register specifier. A balanced implementation, based on the analysis in [10], would divide this address space equally between global and local variables for any created thread, which means that they would access to 16 locations in the  $\$G$  window and 16 locations shared between the  $\$L$ ,  $\$S$  and  $\$D$  windows. Assuming that a thread requires  $L$  locals and  $S$  shared variables (these are specified in the create control block), then a single window is allocated for every

thread with  $L+2S$  locations, where  $L+2S \leq 16$ . Each thread requires  $S$  registers for its own  $\$S$  window and another  $S$  for the  $\$D$  window that is linked to its producer's  $\$S$  window. To maintain backward compatibility for legacy code, the main thread would have to have access to all 32 addressable locations and, as the main thread in micro-threaded code has no dynamically allocated registers, its  $\$G$  window could also have 32 locations. Only the first 16 of these registers, would be available to threads that were dynamically created.

In thread code, the compiler would translate addresses to  $\$G$  to the lower half of the register specifier range and to  $\$L$ ,  $\$S$  and  $\$D$  to the upper half. In the main thread, the compiler would also translate the  $\$S$  window to a location in the upper global window. The hardware would trap writes to  $\$G$  using the ms bit of the specifier and a single base address associated with each thread would allow access to  $\$L$ ,  $\$S$  and  $\$D$  windows. For reads to empty locations greater than  $L+S$ , the hardware would trap to a mechanism to read the producer's  $\$S$  window. The thread's state also needs the base address of its producer window and the processor the producer thread is running on. This is illustrated in figure 3. Note that when a read to  $\$D$  is accessed from the producer's  $\$S$  window it is read from an address  $S$  locations lower in that window. The three components of thread state must be passed through the pipeline to the register read stage. These are the base address for the thread (this is also required at the writeback stage), the base address for its producer thread and the processor on which the producer thread is mapped.

## 4 Conclusions

This paper has introduced a model of explicit concurrency, that is dynamic and which can be used as a target for the compilation of existing sequential languages. The code produced exploits instruction-level concurrency across all kinds of loop structures. In order to schedule code using these explicit concurrency controls the model also requires distributed synchronisation on registers, implemented as an i-structure, which is restricted to binary communications, i.e. communication between one producer and one consumer thread. This model supports a general model of sequential computation at the high-level language with concurrency produced automatically by the compiler. Such a model was very successful on the vector supercomputers of the past. Unlike the vectorisation in these predecessors, this model will still generate concurrent code even if the sequential source exhibits dependencies between the iterations of a loop. Excess concurrency on each processor is available to allow data-driving scheduling with efficient mechanisms for context switching and synchronisation.

It is also shown in this paper that such a model may be implemented on a distributed collection of in-order pipelines, with dynamic allocation of registers to threads. Unlike an out-of-order issue processor, where the synchronisation namespace is much smaller than the physical register pool, in this model the synchronisation namespace is potentially larger than the physical register pool. This means that concurrency is no longer limited by hardware implementation

constraints but is limited only by the concurrency exposed in the loop structures in the source code.

The distributed implementation proposed is totally asynchronous, allowing for the implementation of all global communication by asynchronous means. Three global bus/network structures are required. The first is a switching network to perform the binary synchronisation, the second is a broadcast bus for broadcasting global variables and the last is a bus for creating a global description of the families of threads that implement the loop concurrency. Global communication is known to be a big issue in current and future systems and papers on asynchronous interconnect are already appearing in the literature[11]. This model and outline implementation therefore provides a way forward from the current problems faced in out-of-order, speculative instruction processing. The only potential problem with this model is that existing sequential code will not provide speedup unless some form of binary-code translation is provided that performs a parallelisation on legacy binary code. Finally power need only be dissipated on those processors executing threads, with control being determined by the instruction scheduling mechanism and no power is lost due to speculation errors, which must surely be a problem in current processor designs.

## References

1. R P Peterson et. al. (2002) Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading, *ISSC Digest and Visuals Supplement*.
2. J Burns and J Gaudiot (2001) Area and system clock effects on SMT/CMP processors, *Intl. Conf. on Parallel Architectures (PACT 01)*, pp211-221, IEEE.
3. I Par, M Powell and T Vijaykumar (2002) Reducing register ports for higher speed and lower energy, *Proc. 35th annual ACM/IEEE international symposium on Microarchitecture*, pp 171 - 182 , ACM ISBN ISSN:1072-4451 , 0-7695-1859-1
4. A Bolychevsky, C R Jesshope and V B Muchnick, (1996) Dynamic scheduling in RISC architectures, *IEE Trans. Computers and Digital Techniques* ,**143**, pp309-317.
5. C R Jesshope (2003) Multithreaded microprocessors evolution or revolution, *Proc. ACSAC 2003: Advances in Computer Systems Architecture*, Omondo and Sedukhin (Eds.), pp 21-45, Springer, LNCS 2823 (Berlin, Germany), ISSN0302-9743, Aizu, Japan, 22-26 Sept 2003.
6. L Gwennap, (1997) DanSoft develops VLIW design. *Microproc. Report*, **11**, 2 (Feb. 17), 1822.
7. Y Solihin, J Lee and J Torrellas, (2003) Correlation Prefetching with a User-Level Memory Thread, *IEEE Trans. on Parallel and Distributed Systems*, **vol. 14**, no. 6.
8. C Zilles and G Sohi (2001) Execution-based prediction using speculative slices, *Proc. Intl. Symposium on Computer Architecture*.
9. D Burger and J R Goodman (2004) Billion-transistor architectures: there and back again, *IEEE Computer*, **37**, 3, pp22-28.
10. C R Jesshope (2004) Microthreading, a model for distributed instruction-level concurrency, submitted to *Parallel Processing Letters* (on-line at: <http://www2.dcs.hull.ac.uk/people/csscrj/papers.html>).
11. A Lines(2004) Asynchronous interconnect for synchronous SoC design, *IEEE Micro*, **24**, 1, pp32-41.