

# *A Concurrency Model for Instruction-level Distributed Computing*

Professor Chris Jesshope

Department of Computer Science  
University of Amsterdam, Amsterdam  
The Netherlands  
bossman@mac.com

## **Abstract**

*In this paper a model for instruction-level distributed computing is described. This model allows the implementation of scalable chip multiprocessors. It is based on microthreading and operates within a single context. It is a direct replacement for the out-of-order issue of instructions but is based on explicit concurrency controls. As such it would not yield speedup unless existing sequential code were recompiled using the concurrency controls. An analysis of the model, in particular the communication and synchronisation, shows that the model can be implemented in a distributed manner in which instruction-level data is held in distributed register files, one per processor. This implementation is scalable as the number of ports in each register file is constant. Only the switching network between processors would have less than ideal scaling properties.*

## **1. Introduction**

There is something terribly wrong with today's microprocessors. People have been talking of billion transistor chips[1] for seven years but even now, are unable to exploit the high levels of transistor-level concurrency available on today's silicon die. We are told that within a decade we will have chip-packing densities that will support massive on-chip concurrency but even today, if microprocessors had not become so burdensomely complex, we would still be looking at levels of on-chip parallelism of the order of 100s of processors and yet no one seems to know how to build general purpose microprocessors that support this level of concurrency at the instruction level. This paper proposes such a model.

Looking at the development of the microprocessor over the last twelve years, Moore's law predicts a packing density increase of 256 and a corresponding speed increase of around 16 (a CMOS transistor's speed is inversely proportional to its length or the square root of the packing density). Looking at the history of the PPC processor (see <http://www.rootvg.net/RSmodels.htm>), we see that clock speed has increased at about twice this rate, over the period i.e. from 33Mhz to 1Ghz. Circuit density has also grown roughly as predicted but if we look at how circuit density has contributed to on-chip processing concurrency, we see an increase of only a factor of 10 (from a 32-bit single-instruction issue to a 64-bit, five-way issue) a typical IPC of around two means that we do not even see much benefit in system speed from packing density. According to Moore's law, we should have an increase in concurrency and hence performance of 256 times over this period, independent of clock speed, so what has happened?

The faster than predicted clock speed is no doubt due to a finer slicing of the pipeline. The smaller than predicted concurrency, a factor of 25, is more worrying and is due to a number of architectural factors:

- i. More area is being used for on-chip memory. Typically 25-33% of the chip area in a modern microprocessor will be second level cache, which exists only to mitigate against the memory wall, as current processors cannot tolerate the high latencies between processor and memory.
- ii. Instruction issue is not scalable in out-of-order issue microprocessors, as the issue logic area grows at least as the square of the issue width and currently consumes an area similar in size to the L2 cache in wide-issue designs [2-4]
- iii. Finally the register file is not scalable [5]. Register file capacity is related to issue width and cell size grows with the square of the number of ports, which in turn is linearly related to issue

width. This means that as we increase issue width, we typically have a cubic scaling of register file area, which will very quickly dominate 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 however, requires explicit concurrency controls and synchronisation added to the instruction set architecture and these in turn require a move away from binary-code compatibility to gain this improvement. It will be shown that backward compatibility may be retained at some cost in efficiency but that recompilation or binary-to-binary translation is required for speedup.

Work has been reported, which mitigates against register file scaling [e.g. 4,6] but they only provide a small constant improvement in the area of the register file, which although will delay the time at which the register file scaling properties will dominate area, speed and power considerations, does so only by a very short period of time. A good example is the half-price architecture [7], which modifies the instruction issue stage and adds an additional pipe-line stage for register read, in order to reduce the number of read ports to the register file from two per instruction issued to one per instruction issued. For a given number of registers this has the effect of approximately halving the size of the register file. However, this is a one-off solution and, according to Moore's law, will not significantly impact the problem of register file scaling. In about 18 months, circuit density will double, if issue width follows circuit density, the register file will have grown by a factor of 8 again! Thus the half-price architecture does not solve the fundamental problem and is not even much of a delaying tactic. The problem needs to be addressed from the perspective of developments over the next decade, by which time, if we rethink our basic architecture, we could be seeing thousands of processors on a single chip.

## **2. The Issues**

The two major issues that must be considered in designing a model to provide scalable, instruction-level concurrency are the register file and instruction issue scaling. The register file provides global communication via a shared memory and is an example of a problem of broader significance, namely global communication in concurrent systems. Scaling independent computations is trivial, scaling fixed patterns of communication is easy and even scaling general communication is not difficult and has been solved at the system level. Although solutions exist, communication does not scale so well as computation. If we ignore wire delays, then to obtain a constant switching delay between  $n$  processors requires  $O(n^2)$  bus switches. This is the requirement of a full cross bar switch. Only if we are prepared to tolerate a switching delay which grows logarithmically with  $n$ ,  $O(\log n)$ , can the number of switches be reduced to  $O(n \log n)$ . Even  $O(n)$  networks can be used to solve the communication problem but their cross section bandwidth will not scale for all communication patterns.

Instruction-level concurrency exhibits exactly the same problems. Out-of-order issue requires global communication between instructions issued concurrently using a shared register file. Multiple ports are required as each instruction issued may require two reads and one write to the register file. The adverse scaling manifests itself through these multiple ports, which have a cross-bar arrangement from word lines to bit lines giving  $O(n^2)$  area per bit. There is also a requirement for larger register files as more concurrency is exposed, which means the number of bits required in total is  $O(n)$  giving an overall scaling of  $O(n^3)$  for the register file. (N.b. here  $n$  represents issue width.)

Ideally, to solve this problem a distributed rather than a shared-memory solution is required and then communication scaling is defined by network design, which has an  $O(n^2)$  upper bound. At an instruction level, what is required is a change to the ISA in order to tackle this problem. The ISA specifies the communication required through the instructions executed and their register specifiers. Current out-of-order issue microprocessors use a legacy ISA with a limited namespace for communication, e.g. 32 or less names! Another issue is the localisation of communication, which is

lost in translation from the high level language. To solve these problems requires an ISA with explicit concurrency controls and a large namespace for communication.

The second issue is one of synchronisation, which in a distributed system is achieved through constraints on communication, such as a blocking read or a blocking write or both [8]. Synchronisation is required between ALUs at the instruction level, between clusters of processors on the same chip at the thread level, as well between the chip and external devices such as memory and networks at all levels. It would be simplifying if these were all based on the same model. In an out-of-order issue processor, instruction-level synchronisation is managed by the issue logic, which keeps track of resources and data dependencies prior to instruction issue. Only when all dependencies have been resolved can an instruction be safely issued. Again an  $O(n^2)$  scaling is inherent in this solution and additional hardware is also required for branch predictors, register renaming and in managing threads in simultaneous multithreading[9]. This model inevitably blurs synchronisation with concurrency controls, as it is based on an ISA with no explicit concurrency.

Perhaps we should consider carefully what is happening in the out-of-order model of concurrency. If we relax the constraint implied in sequential execution, constraints on instruction execution are defined in the binary code by register specifiers, which identify the dependencies between instructions. As already indicated, the namespace is unfortunately very limited. When executing instructions therefore, two things must happen. First, register specifiers must be renamed, in order to remove any introduced dependencies from name-space overloading. Next, every instruction fetched must be checked against the other instructions in the instruction window to identify what, if any, dependencies or resource conflicts constrain its execution. The ISA's namespace, the instruction window size and data dependencies all limit the available concurrency. Synchronising between instructions by restricting issue is inefficient and synchronisation is best distributed to whatever structures are used for communication. For example, using some form of full/empty tags on registers. This will have a minimal impact on scalability, adding a few bits to every register in the register file. More importantly it requires that the ISA include some form of concurrency control with mechanisms for blocking instruction issue based on data availability. Previous fine-grain shared memory architectures have implemented i-structures on registers for instruction issue control. Lists of continuations are maintained to synchronise with the data. A good example is the Delencor HEP. This approach however, would have a significant impact on register-file size and the model described here does not require such a general synchronisation at the instruction-level.

### **3. The Microthreaded Model of Instruction-level Concurrency**

#### **3.1 Related work**

Micro-threading is a model of concurrency limited to a single context, which shares the registers allocated to that context. It can support instruction level concurrency across all iterations of a variety of loop structures, including *for* loops and *while* loops. The model was first proposed in 1996 [10] and has been further refined and evaluated in subsequent papers[11-13]. A number of other papers have also considered similar models, the earliest being nano-threads in [14], 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 [15-19]. More recently, in [20], a thread model called mini-threads has been evaluated with a view to increasing concurrency in an SMT architecture, without increasing register-file size. SMT uses threads that require their own context and architectural register set. Mini-threads are a means of increasing concurrency without increasing the size of the physical register pool. The argument given is that by sharing the architectural register set of one thread between two or more mini-threads (two were investigated in [20]), concurrency is increased without requiring additional registers. It is not at all clear, that increasing concurrency by the use of mini-threads does not put pressure on the register pool. However, this is an application-related rather than an architectural constraint. Last but not least,

threading has been used in a broader context with a single shared register set in dataflow architectures, see for example [21].

### 3.2 Concurrency controls

Micro-threading requires additions and possibly modification to a base ISA in order to implement its concurrency controls. This concurrency can then be used to provide latency tolerance in a single microprocessor or to schedule instructions simultaneously in a chip multi-processor. It is therefore necessary to recompile source code in order to obtain speed-up using this model but it is possible to design a microthreaded microprocessor that executes legacy binary code without speed-up or indeed provide binary-to-binary translation on legacy code. Very few new instructions are required to implement these controls. We need an instruction to create one or more threads (*Cre*) and either two further instructions or an additional field in all instructions to signal when to context switch and when to terminate a thread. To maintain full backward compatibility with legacy code, additional instructions must be used rather than tags, so as not to change the fields in the base ISA. Tagging could be used if legacy code was transformed using a binary-to-binary translation. As an example, the C code:

```
struct box {
    int next;
    int x1;
    int x2;
    int y1;
    int y2;
}

struct box start;
struct box locate( int x; int y; start)
{ while (start.next != 0)
    {if (x >= start.x1)
        if (y >= start.y1)
            if (x <= start.x2)
                if (y<= start.y2)
                    return start;          /*match*/

        start := start.next
    }
return -1;
}
```

would be translated to a generic micro-threaded assembler shown in Figure 1.

In this code it is assumed that the function's parameters are passed via registers, i.e.  $x=\$G1$ ,  $y=\$G2$  and  $start=\$G3$ . It is also assumed that the return value (the address of the element located) is passed via  $\$G6$  and that  $\$G7$  holds the return address. The peculiar register specifiers and the *Bsync* instruction are defined later in sections 3.3 and 3.4 respectively but can probably be ignored on first reading. Looking at the concurrency controls, the *Cre* instruction (emphasised) has a label, which locates an 8-word control block in data memory. The parameters in this block define a family of threads that is created when this instruction executes. The parameters are defined as follows:

- the first three words give the start, limit and step values that define the loop/family bounds;
- the next word gives a constant stride defining dependencies between instances of these threads;
- the next two words define the number of local and shared variables allocated to each thread;
- the last two words are the address of the code executed for all threads with the possible exception of the last, which may use the optional address if not zero.

A microthreaded microprocessor creates one thread for each index value defined by the first three parameters, which, although not required in this code, is written to the first of the local registers allocated to each thread. As a while loop has no limit determined, the bounds here are defined by the

<pre> while:  .data         .word 1          #start         .word 5n        #limit         .word 1          #step         word 1           #dependency distance         .word 4          #locals         .word 1          #globals         .word body       #code         .word last       #optional last iteration locate: Mv \$S0 \$G3         <b>cre while</b>      #create a threads         bsync         mv \$G4 -1         jr \$G7 body:   mv \$L2 \$D0        #get addr of item         swch             #context switch         lw \$L3 next(\$L2) #get addr of next         beq \$L3 \$G0 kill #Kill if end of list         mv \$S0 \$L3       #else pass to next         lw \$L1 X1(\$L2)   #get lower X         bge \$G1 \$L1 fail         swch         lw \$L1 Y1(\$L2)   #get lower Y         bge \$G2 \$L1 fail         swch         lw \$L1 X2(\$L2)   #get upper X         ble \$G1 \$L1 fail         swch         ... </pre>	<pre> ...    lw \$L1 Y2(\$L2)    #get lower Y         ble \$G2 \$L1 fail         swch         mv \$G6 \$L2         jr \$G7          #success return addr fail:  kill kill:  mv \$S0 0         #propagate 0 to kill other thrds         kill last:  mv \$L2 \$D0       #get addr of item         swch         lw \$L3 next(\$L2) #get addr of next         beq \$L3 \$G0 fail #Kill if end of list         mv \$S0 \$L3       #else pass to next         cre while       # last thread creates         lw \$L1 X1(\$L2)   #get lower X         bge \$G1 \$L1 fail         swch         lw \$L1 Y1(\$L2)   #get lower Y         bge \$G2 \$L1 fail         swch         lw \$L1 X2(\$L2)   #get upper X bound         ble \$G1 \$L1 fail         swch         lw \$L1 Y2(\$L2)   #get lower Y         ble \$G2 \$L1 fail         swch         mv \$G6 \$L2         Jr \$G7 fail:  Kill </pre>
--	---

Figure 1. Compilation of function with a while loop into microthreaded code.

target hardware and the number of threads created at a time is a multiple of the number of processors. The last thread of each family creates a new family until either the search succeeds or the end of the list is encountered. In a multi-processor implementation, these threads would be issued one to each processor per clock cycle and executed concurrently. Thread creation occurs only when resources (e.g. registers and continuation slots) are available. Execution is subject to some restrictions; namely the instructions in each thread must be issued in strict sequence but threads can be distributed arbitrarily to multiple processors and instructions from multiple threads in a single pipeline can be interleaved arbitrarily, subject only to the constraints defined by the data dependencies.

Interleaving is defined by explicit context switching, which is required when there is data dependency in an operand of an instruction. In this code it is defined by a *Swch* or *Kill* following that instruction. In this case dependencies exist in reading shared variables e.g. \$D0 and in data loaded from memory, which is decoupled. *Swch* and *Kill* may either be encoded as new instructions or may be encoded in the operation itself with a 2-bit field that specifies how the next instruction is chosen: {*normal*, *switch*, *kill*}. A normal tag will take the next instruction from the current thread, switch and kill tags take the next instruction from another thread if this is possible and where the kill tag also indicates termination of the current thread. When tagging is used, the *Swch* and *Kill* instructions in figure 1 become pseudo instructions that define these tags for the preceding instruction; they would not therefore require a slot in the pipeline. 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. On single threaded code, or when no other thread is active, the pipeline will stall on reading an empty register until either the data becomes available, or another thread is reactivated, in which case the pipeline must first be voided up to the register read stage. When a register read fails and the pipeline is not stalled, a single micro-context may be stored in the empty register to suspend it. When data is written to that register, the context is reactivated. This implements a blocking read and, at the instruction level, makes perfect sense, as we

do not want to halt instruction execution to provide a bidirectional synchronisation such as implemented in the transputer[8].

It must be understood that this model is defined over a single context and that any transfer of control between functions must result in a single thread of control. In figure 1, any microthread can execute a return from the function and this action will kill all other threads. It must also be recognised that the model works only at the instruction level. There is no synchronisation on memory and concurrent writes to memory must be controlled by synchronisation at the register level. Dependencies are carried between threads by registers. The pattern of dependencies between the threads in a family is defined by the dependency distance in the control block. Every thread, except the first in the family, is able to read data from registers in a prior iteration at a fixed distance from it (the dependency distance). The first thread in a family can also read registers from the thread that created it but only if a dependency distance other than zero has been defined. A dependency distance of zero defines a family of independent threads and these have no dependency relation between them nor to the thread that created them. Thus any chain in the dependency graph follows iteration order and starts from the thread issuing the *Cre* instruction, through the threads it created in the order defined by the loop bounds and with a skip distance defined separately from those bounds.

### 3.3 Synchronisation name space

Synchronisation defines a partial order over a given namespace. This partial order defines the concurrency implicit in that code and the order of computation where there are sequential constraints. In a superscalar architecture, the namespace is defined by the physical register pool but is communicated by the compiler through the register set defined in the ISA, usually significantly smaller. Renaming recreates a partial dependency graph based on the physical register pool but only on those instructions (in execution order) that fit into the instruction window.

The microthreaded model also uses registers as its namespace for synchronisation as this is the only option that can provide efficient control over the order of execution at the instruction level. Its namespace is large and dynamic and is defined as the sum of all registers allocated to all threads in a given context. It is dynamic because thread creation is dynamic (loop bounds can be variable and threads can beget other threads recursively). The namespace can be much larger than the physical register pool and this gives portability across a range of target architectures with different levels of concurrency. One potential problem is that deadlock can result from a namespace physically larger than the resources used to implement it. The problem is that resources for a thread cannot be released until any dependent thread has completed and the resources tied up may prevent the dependent thread from being allocated. This is not a principle problem however, as dependencies usually jump small distances in the loop index space and this situation can be detected at run-time.

Unlike out-of-order execution, which enforces dependencies by constraining the issue of instructions, the microthreaded model has a distributed model of synchronisation using a blocking read on every register in the physical register pool. In dataflow terms, each register implements an i-structure that is allocated in the empty state when a thread is allocated to a processor, or when a load word instruction is issued to it. 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. I-read suspends the issuing thread if the register is empty or returns the value stored if it is full. Normally, a list of deferred continuations is associated with an i-structure but this would increase the complexity of the register file, something we are aiming to reduce. Instead, the compiler must enforce binary synchronisation, i.e. there cannot be concurrent readers of a register not known to be in the full state. This is not a principle restriction, as multi-way synchronisation can be achieved using thread creation; a single guard thread reads the variable in question and then creates multiple threads, which can then read the data safely and concurrently.

In addition to pair-wise synchronisation the microthreaded model requires a barrier synchronisation (*Bsync*) and a Break (*Brk*) instruction for efficiency. The *Bsync* instruction suspends the issuing thread

until all other threads have completed, whereas a *Brk* instruction would simply kill all other threads. Both might have implementations that only affected the descendants of the issuing thread, if such an implementation were feasible.

### 3.4 Partitioning the register namespace

Rixner et. al.[6] show that the ideal scaling of the register pool in a chip multi-processor distributes all registers between the ALU ports, with a switch providing routing of results to the appropriate ALU/port. Although such an implementation is appropriate for the streaming applications considered in that paper, it does not support a general-purpose model of computation where a static allocation of instructions to ALUs is not possible. On the other hand, a single global register file has a scaling of  $O(n^3)$  for n-way concurrency (multiple-issue and/or multiple processor). A register file partitioning between processors therefore is the goal of any implementation of this model.

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

- invariants give rise to broadcast communication and such variables can be written to and read from in any thread. They are called *Global* registers in this model and are represented by a specifier  $\$G_i$  in assembler;
- dependencies give rise to pair-wise communication between two threads and these variables can be written by one thread and read by another. They are called *Shared* in the producer thread and represented by a specifier  $\$S_i$  and are called *Dependent* in the consumer thread and represented by a specifier  $\$D_i$ . It should be noted that  $\$S_i(\text{producer}) = \$D_i(\text{consumer})$ ;
- all other variables are called *Local* and used only for communicating data within a single thread or between the memory system and that thread. These are represented by  $\$L_i$ .

In this model, each class of variable is allocated as a logically-separate register window as mechanisms for access to each class of variable will differ, depending on implementation. A scalable implementation for allocation and access is discussed in section 4. In the assembler the different classes of variables are specified using a \$ sign, a letter (from G, S, D or L) and a range  $0 \leq i \leq R$ . The state of a thread must now include information such as base addresses to locate a thread's register windows in the register file. The global window is accessible to all threads and its location would be held as a part of the context's state.  $\$S$ ,  $\$D$  and  $\$L$  windows are allocated dynamically when the thread is created, as and when resources become available. The number of local and shared variables for a given thread is defined in the thread control block (see *Cre* instruction in section 3.2). The dependency distance in the thread control block determines the relationship between the producer and consumer threads in a given family of threads.

### 4. Distributing the Register File in the Microthreaded Model

An ideal implementation of the microthreaded model would have a completely distributed register file organisation. That implementation would then need to manage the three different types of communication specified by the partitioning of the register namespace as defined in section 3.4. This section looks at the mechanisms that would enable this model to be implemented in a distributed manner.

It is trivially easy to distribute the local register window, as the accesses to it are local to one thread, which is allocated to one processor. There is still a requirement for synchronisation on this window, as in the ALU, some iterative operations may take a variable time and similarly on memory reads, a cache miss will also be asynchronous. Thus the  $\$L$  window can be allocated to a local register file but it requires synchronisation and so must implement the simplified i-structure model proposed.

Distributing the \$G window is not as simple; it is shared between all threads and can include two types of communication. The first are loop invariants that are assigned prior to the execution of a loop and are read by all threads implementing it. The second use is in dependencies that flow against the order imposed by thread creation, for example in terminating a dependency chain. An example is given in figure 1, where any thread can set \$G6 that the function returns. This results in non-determinism, as many processors may succeed in their search on different list elements simultaneously. This code is a relaxed implementation and it returns the first search success anywhere in the list. A strict sequential compilation would need to pass a token as another dependency between threads to enforce sequential-order termination. This would resolve the non-determinism but result in less efficient code. The fact remains however, that any implementation of the global register file must include arbitration to resolve any non-determinacy. One implementation would be to replicate the \$G register window in a local register file on every processor and to provide a single global bus with arbitration between processors for writes to this window. Reads are local and require no special implementation. Writes occur locally as normal but get reflected in the other processors asynchronously, depending on bus implementation. Synchronisation is obviously required on reads as we cannot rely on issue sequence, except in the thread writing the value.

The final question to be asked is whether the \$S window can also be distributed to a local register file. The answer to this will depend the number of concurrent reads or writes to the \$S windows mapped onto one processor. If this grows with the number of processors, there is nothing to be gained from distributing the \$S windows.

The \$S and \$D windows allow pair-wise communications to be defined by the dependency relationship between threads. The \$S window is in the namespace of a single producer thread and only the producer may write to the \$S window. The \$D window is in the namespace of one or more consumer threads and in a global register file, proposed in earlier papers [11,13], both producer and consumer would reference the same location. The \$D window is said to be mapped onto the \$S window by the dependency relationship between the threads. Many-to-one mappings between \$D and \$S windows are possible but in this model, 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 restriction. This means that the range of the register specifier in the \$D window gives an upper bound on the number of concurrent reads to any given \$D window and provides us with a model constraint that allows distributed implementation. In fact the only way to create multiple dependencies on a \$S window is to create multiple families of threads, each having a dependency other than zero. In this case the first thread in each family has a dependency on the thread that created it. In this model, there is no systematic way to create a family of threads where every thread has a dependency on the thread creating it. Such multi-cast must either be achieved by broadcast using \$G variables or must be implemented by a chain of dependencies running through a family of threads.

In a distributed implementation, the \$D window(s) may be associated with different processors to the \$S window. In this case, a thread's state must also contain the processor id and base address where its shared window is allocated. Reads and writes to the \$S window occur in just the same way as to the \$L window, they are local with no requirement other than the implementation of synchronisation. Reads to the \$D window, on the other hand, would trap to a remote request to be satisfied by a switching network between the processors. To make a scalable CMP, it is imperative to decouple the operation of the local pipeline with the remote register read, otherwise the pipeline would have an arbitrarily long register read stage due to any switch delay and arbitration. There is a mechanism to achieve this, which is already used successfully to tolerate non-deterministic access times to memory. The thread reading the \$D register must context switch and be suspended locally. To do this requires the \$D window to be allocated locally in the consumer thread as well as being mapped remotely to the \$S window in the producer thread. A read to \$D therefore suspends the reading thread if the location is empty and this initiates an asynchronous read to the remote \$S register via a switching network. Any delay in network implementation can be tolerated by context switching, given sufficient local

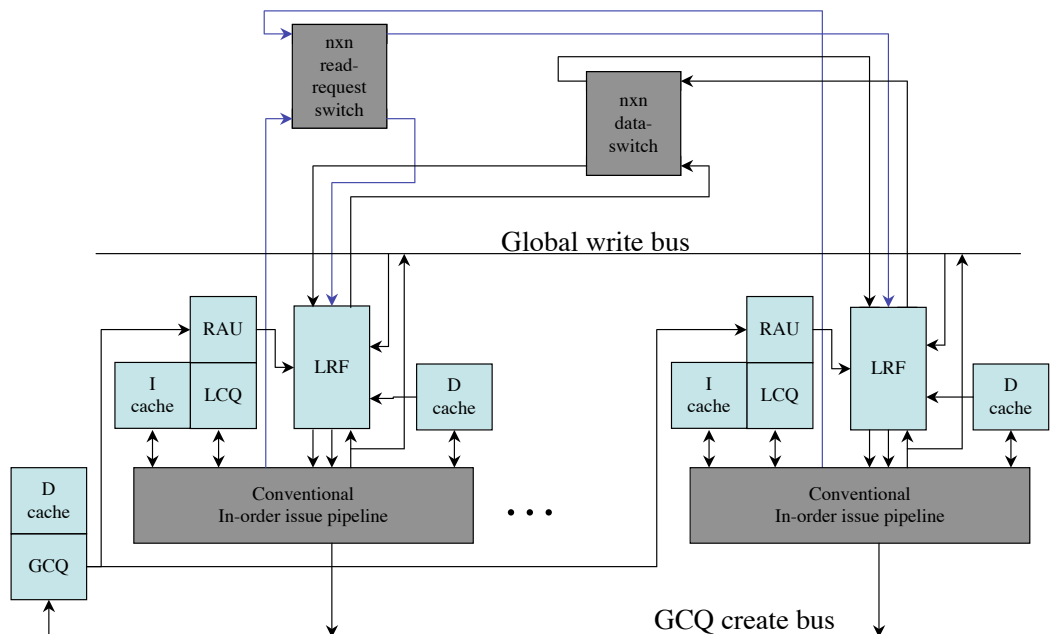


Figure 2. A chip shared-register distributed chip multi-processor based on the microthreaded model of instruction-level concurrency

concurrency and any subsequent read to the same location once the remote fetch is satisfied becomes local.

### 5. A Distributed Chip multiprocessor

Figure 2 shows a distributed implementation of a chip multiprocessor based on the microthreaded model of instruction-level concurrency. It reflects the three different types of communication defined in section 4. Local communication, whether within a thread or between threads allocated to the same processor, is via the local register file, LRF. Global communication is via a window in the local register file that is duplicated on all processors and where consistency is maintained between them by writes to a *global write bus*. Arbitration is required for use of this bus and hence a remote write may be delayed with respect to the local write but where the blocking read on all registers avoids any inconsistency in program state. Finally pair-wise communication between threads, where the producer and consumer threads are allocated to different processors, is achieved by allocating  $\$S$  and  $\$D$  registers to separate windows on the respective processors and suspending reads to  $\$D$  until a remote fetch can make the  $\$D$  window consistent with the remote  $\$S$  window. There is one further remote bus required to implement this model and that is the GCQ create bus. This arbitrates between processors on each cycle allowing only one processor to create a family of threads. The issue of memory hierarchy is deliberately avoided here, as it is not clear that a conventional L2/L3 cache hierarchy is what is required for this model.

The local register file has a maximum of 8 ports, these comprise:

- three ports for the conventional pipeline operation;
- one port to support decoupled access to memory on a cache miss;
- one port for the register allocation unit (RAU) to initialise the first local register with the index value for the thread;
- one read and one write port to support remote register requests; and
- finally a write port from the global write bus.

An implementation might optimise these ports depending on requirements using two techniques. By stalling instruction issue for one cycle and inserting an appropriate move instruction into the pipeline, any of the 4 asynchronous write ports can be removed and the writes made using the standard three

ports. Similarly by creating appropriate control, a stolen cycle can be used to satisfy the remote read request. Alternatively, arbitration can be provided to a single write port without stalling instruction issue. Priority would have to be given to \$L initialisation. An analysis of a variety of compiled loop kernels in [22] shows that only 9% of instructions executed read the \$D window and that for problem sizes of greater than 64, less than 0.5% of instructions executed write to the global register file. Thread allocation on these codes would occur in only 4-33% of instructions executed as the kernels generate between 3 and 25 instructions per thread. Finally the decoupled memory access would be dependent on cache miss rate. Even allowing for a 50% miss rate, a single write port would not on average be fully utilised based on this analysis.

## 5. Conclusions

In this paper a model of instruction-level concurrency has been analysed for implementation as a scalable chip multiprocessor. Conventional instruction-level parallelism based on out-of-order issue uses instruction issue to control synchronisation between instructions and also uses a global shared register file for communicating values between concurrent functional units. Neither are scalable to wide instruction issue. The model proposed in this paper is based on microthreading, which uses explicit concurrency controls and registers implementing a restricted i-structure for synchronisation. It has already been shown that instruction issue in this model is scalable[13]. This paper analyses the communication and concludes that it may be implemented as a distributed collection of conventional in-order issue microprocessors communicating asynchronously via some kind of inter-processor switching network. This is a significant achievement as it is, the author believes, the first general-purpose model of instruction-level concurrency that allows a truly scalable implementation.

## 6. References

- [1] D. Burger and J. R. Goodman (Eds.) (1997) *IEEE computer*, Theme Feature on “Billion Transistor Architectures”, **30** (9).
- [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] R P Peterson et. al. (2002) Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading, ISSC Digest and Visuals Supplement.
- [4] V Agarwal, H S Murukkathampoondi, S W Keckler, and D C Burger (2000) Clock rate versus IPC: The end of the road for conventional microarchitectures, *Proc 27th International Symposium on Computer Architecture (ISCA)*, June, 2000.
- [5] 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
- [6] Scott Rixner, et al. (2000) Register organization for media processing, In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture* (January 2000), pp 375—386, IEEE.
- [7] Ilhyun Kim, Mikko H. Lipasti (2003) Half-Price Architecture, *Proc 30th Intl. Symposium on Computer Architecture*, p28, IEEE.
- [8] M. Homewood, D. May, D. Shepherd and R. Shepherd (1987) The IMS T800 Transputer *IEEE Micro*, October 1987, pp10-26.
- [9] Dean Tullsen, Susan Eggers, and Henry Levy (1995) Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp392-403.
- [10] 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.
- [11] C. R. Jesshope (2001) Implementing an efficient vector instruction set in a chip multi-processor using microthreaded pipelines, *Proc. ACSAC 2001, Australia Computer Science Communications*, **Vol 23**, No 4., pp80-88, IEEE Computer Society (Los Alimitos, USA), ISBN 0-7695-0954-1, Brisbane, Australia, 29-30 Jan 2001.

- [12] Luo B. and Jesshope C. (2002) Performance of a Microthreaded Pipeline, in *Proc. 7th Asia-Pacific conference on Computer systems architecture*, **Volume 6**, ( Feipei Lai and John Morris Eds.), pp83-90, Australian Computer Society, Inc. (Darlinghurst, Australia), ISSN:1445-1336 , ISBN 0-909925-84-4, Melbourne, Australia, 28 Jan - 2 Feb, 2002..
- [13] 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.
- [14] L. Gwennap, (1997) DanSoft develops VLIW design. *Microproc. Report 11*, 2 (Feb. 17), 18–22.
- [15] Yan Solihin, Jaejin Lee and Josep Torrellas, (2003) Correlation Prefetching with a User-Level Memory Thread, *IEEE Trans. on Parallel and Distributed Systems*, **vol. 14**, no. 6.
- [16] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi (2001) Dynamically allocating processor resources between nearby and distant ILP. In *Proc. Intl. Symp. on Computer Architecture*
- [17] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt (1999) *Simultaneous subordinate microthreading (SSMT)*. *Proc. Intl. Symposium on Computer Architecture*.
- [18] J. Redstone, S. J. Eggers and H. M. Levy, (2000) An analysis of operating system behavior on a simultaneous multithreaded architecture, *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.
- [19] C. Zilles and G. Sohi (2001) Execution-based prediction using speculative slices, *Proc. Intl. Symposium on Computer Architecture*.
- [20] J. Redstone, S. Eggers and H. Levy (2003) Mini-threads: increasing TLP on small-scale SMT processors, *Proc 9<sup>th</sup> Intl. Symp. On High Performance Computer Architecture (HPCA-9)*, p19, IEEE.
- [21] R. S. Nikhil, G. M. Papadopoulos and Arvind (1992) \*T: A multithreaded massively parallel architecture. *Proc. Intl. Symposium on Computer Architecture*.
- [22] 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>)