

## MICROTHREADING A MODEL FOR DISTRIBUTED INSTRUCTION-LEVEL CONCURRENCY

CHRIS JESSHOPE

*Department of Computer Science, University of Amsterdam, Kruislaan 403,  
Amsterdam, 1098SJ, NL*

Received April 2004

Revised February 2006

Communicated by Kemal Ebcioglu

### ABSTRACT

This paper analyses the micro-threaded model of concurrency making comparisons with both data and instruction-level concurrency. The model is fine grain and provides synchronisation in a distributed register file, making it a promising candidate for scalable chip-multiprocessors. The micro-threaded model was first proposed in 1996 as a means to tolerate high latencies in data-parallel, distributed-memory multi-processors. This paper explores the model's opportunity to provide the simultaneous issue of instructions, required for chip multiprocessors, and discusses the issues of scalability with regard to support structures implementing the model and communication in supporting it. The model supports deterministic distribution of code fragments and dynamic scheduling of instructions from within those fragments. The hardware also recognises different classes of variables from the register specifiers, which allows the hardware to manage locality and optimise communication so that it is both efficient and scalable.

*Keywords:* Chip multiprocessors, Microthreads, Concurrency, Scalability, Dataflow

### 1. Introduction

It is now abundantly clear that within a decade we will have chip-packing densities that will support massive on-chip concurrency. 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 floating-point processors. The problem is that most computers are designed to support binary code compatibility and this yields microprocessors that execute multiple instructions out of order from a sequential stream of instructions. Although this approach supports implicit parallelism within the binary code, it is not at all scalable and an alternative model, microthreading, is presented here with much better scaling properties. This approach also allows backward compatibility of binary-code with no speed-up but does provide speed-up if that code is recompiled (or translated from the binary) only once! The new code uses a few additional instructions in the base ISA to support the explicit description of concurrency (using dynamic meta-data). In addition, it provides synchronisation primitives; a global synchronisation to support bulk-synchronous update of shared or distributed memory and synchronising registers that block instruction issue and

allow true data dependencies to be expressed between concurrently executing code fragments without speculation. This paper takes a broad look at the issues involved in this model with the perspective of some notable milestones in parallel processing.

Parallel processing is a subject that never really seems to mature, even though it has had a great deal of time to do so; the earliest texts on this subject were published more than two decades ago [1]. One reason for this lack of maturity is the relationship between parallel computer architecture and silicon technology, which over the same period has changed so much as to appear discontinuous on a wider historical time scale. We know this change is not discontinuous but exponential and quite predictable. However, this complex relationship has provided a less than stable foundation on which parallel processing has been built (and rebuilt!). There is also considerable inertia in chip architecture, as it is constrained by the economics of a global user base with a considerable investment in legacy software, mostly in binary form. This paper suggests that the move to on-chip concurrency has been sidestepped due to the inherent difficulty in controlling concurrency dynamically. However, as a consequence of the technological advances, this situation cannot be ignored for long. The paper presents microthreading as an emerging paradigm that has its roots in much of the prior work on concurrency. It provides the scalability and code stability required from the commercial perspective in an exponentially changing on-chip environment.

### *1.1. Vector and array processors*

The 1970s and early 1980s saw the first appearance of parallel computers, which exploited loop-level parallelism. These computers were organised to process vectors or arrays of operations efficiently, either in pipelines (vector supercomputers) or using arrays of interconnected processors working in lock step (array processors). These systems exhibited a wide range of parallelism with values of  $n_{1/2}$ , ranging from ten through to many thousands ( $n_{1/2}$  is a measure of concurrency defined on asymptotic performance models found in pipelined implementations and measures the number of operations required to achieve one half of the peak performance see [2] for a complete definition of  $n_{1/2}$ ). For small  $n_{1/2}$ , programmers would use existing sequential code and vectorising compilers but loops which contained dependent operations, could not be vectorised[3]. For high  $n_{1/2}$ , programmers would use explicitly data-parallel languages[4,5] but typically these were machine specific and code written for one target could not be used on another. This issue of portability has continued to plague parallel processing and even today, three decades later, we can still only support the portability of parallel code through the use of libraries of concurrency controls, such as MPI and others. These library solutions are not appropriate for on-chip concurrency, which is likely to have a significant component of concurrency at the instruction level. However, existing source code does provide significant concurrency at the loop level and this must be exploited in any model targeting on-chip concurrency.

In sequential computers, instructions are executed in loop-body order for each loop index but in a vector computer instructions are executed across loop-index for each instruction in the loop body. Both are unnecessarily restrictive and loop concurrency should be captured explicitly so that any schedule of instruction exe-

cution can be implemented within the scope of loop-index and loop-body, so long as all true data dependencies are honoured. This would capture loop concurrency, do-across concurrency in dependent loops and instruction-level concurrency within the loop body. Microthreading achieves this and its schedules are data-driven.

### *1.2. Multiple micros*

With the inexorable rise of on-chip packing density, the 1980s saw the integration of the first complete 32-bit microprocessors onto a single chip. This development led to the subsequent move from data-parallelism to process-parallelism, as these commodity processors began to be used to construct networks or clusters of microprocessors. One notable building block was the INMOS transputer, the T424[6] and later the T800[7]. The INMOS transputer included a 32-bit microprocessor, memory and inter-chip communication channels all on a single die. Transputers also provided hardware-supported concurrency controls in their ISA. A lightweight process model was implemented in hardware providing synchronisation and scheduling linked to the communication channels. Inter-process communication is mapped to either on-chip (shared memory) or off-chip communication across the built-in communication links. The choice was made at link/load time giving the model configuration-independent compilation of code. Commodity microprocessors are still pre-eminent in the design of supercomputers and in the top 500 (see <http://www.top500.org/>) most are based on off-the shelf microprocessors, although in 2003-4 the top slot was occupied by a multiprocessor based on a pipelined vector instruction set[8].

Although the INMOS transputer was as close as we have come to an explicitly concurrent microprocessor, it did not capture the concurrency of loops and the concurrency it did capture was at a relatively high level of granularity. Loop concurrency was captured by manually decomposing loops into processes that each executed a subset of the index space and thus programs had to capture scheduling as well as its functionality. Ideally process concurrency should capture concurrency at its lowest level of granularity and leave scheduling decisions to the implementation, which should be designed to efficiently schedule units of concurrency not much larger than single instructions. The scheduling decisions, or more accurately the tradeoff between distribution and scheduling, where the former describes true concurrency and the latter interleaved concurrency, can then be separated as a system concern and implemented dynamically, depending on whether throughput or latency tolerance is the primary concern.

Microthreading can also be viewed as a process model of concurrency. It can describe dynamic homogeneous concurrency as well as static heterogeneous concurrency down to the instruction level. Implementation studies have shown that families of threads containing a single instruction can be scheduled more efficiently in this model than in a conventional sequential processor, even when there are dependencies between each of the microthreads.

### *1.3. Implicit instruction-level parallelism*

During the 1990s and early 2000s little has changed in the field of concurrent systems. The drive has been to build faster and more powerful microprocessors,

based on instruction-level parallelism. If we look at the development of the micro-processor over the last twelve years, Moores law predicts a packing density increase of 256 and a corresponding speed increase of around 16 (the speed of a CMOS transistor is inversely proportional to its length, which is, to a first order approximation, the square root of packing density). If we look at the history of the Power PC processor as an example of progress (see <http://www.rootvg.net/RSmodels.htm>), clock speed has increased at about twice this predicted rate, i.e. from 33Mhz to 1Ghz over this 12 year period. Over the same period, circuit density has increased entirely as predicted (256 times) but if we look at how circuit density has contributed to instruction-issue width, we see only a factor of 10, with the PPC moving from 32-bit single-instruction issue to 64-bit, five-way, instruction issue. According to Moores law we should be seeing 25 times this increase in issue width, 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 issue width, 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 25-50% 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[9,10,11].
- (iii) Finally the register file is not scalable[12]. 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 system area, speed and power considerations.

Rixner et. al.[13] recently looked at the issue of register file scaling from the perspective of streaming media applications. They analysed the relative area of a number of chip-parallel designs and concluded that fully distributed solutions, where each ALU port has its own register file with another pair of ports providing connectivity through a fully connected switch gave the best results in terms of area and power. Microthreading adopts a similar approach, although it uses a conventional single-issue, 3-port register file with additional ports for casynchronous communication. The difference is that microthreading can execute a wide range of applications using only a ring network, which provides even better scaling properties than the fully connected networks proposed by Rixner. Executing independent loops is mapped to networks that need only broadcast and logical reduction for distributing work and signalling termination of a concurrent section respectively. Executing loops with regular dependencies, such as loop-carried dependencies, requires ring connectivity in addition to the above. If arbitrary dependencies are required, which has not yet been demonstrated, a fully connected network would be required but the asynchronous nature of the communication in microthreaded processors and the latency tolerance would support packet routing over planar to-

pologies. Much research has already been performed in this area and much of it is applicable to the move from multi-processors in general to chip multiprocessors, e.g. [14].

It should be noted that there are numerous other approaches to achieving concurrent instruction issue and scalability, and many of these are based on data-flow. These include the TRIPS architecture[15], Wavecache [16] and MIT's RAW architecture [17]. Each of these has a number of properties in common, for example, they all (including microthreads) use static placement (distribution) of instructions coupled with dynamic or data-driven scheduling. It remains to be seen which of these new approaches will lead to the most flexibility in terms of scaling and the most compatibility in exploiting legacy code, both now and in the future.

#### *1.4. Synchronisation*

Efficient and scalable synchronisation is a key issue in the implementation of concurrent instruction issue. Synchronisation is required between ALUs at the instruction level, between clusters of processors at the thread level and between the chips and other external devices such as memory and networks at the system level. Ideally these should all use the same mechanism. For minimising latency, synchronisation should be achieved at the lowest level, i.e. in the registers used by the ALUs as operands to its instructions. The problem here, is the limited namespace used in most ISAs, usually a 5-bit address.

To overcome this limitation in out-of-order concurrent instruction issue, register renaming is used. This provides a new context for each use of a given register specifier and can be expensive. A more organised approach to contextualising the register file has been used in the SUN SPARC ISA and more recently in the IA 64. These contexts are sequential rather than concurrent ones and managing them is relatively cheap.

In microthreading, register remapping is achieved with contextual pointers into the register file. These contexts are created dynamically with the threads and instructions use both contextual pointer and register specifier to address operands and results. They are also created empty and can not be read unless written to. Using this technique, it becomes possible to use a large distributed register file as the synchronising memory. Such an approach is entirely scalable and with thousands of processors on chip the the register file could be many MBytes in size and completely replace the first level of cache. Other approaches have used full/empty bits on registers to achieve synchronisation and have all derived from the Delencor HEP[2], which was a notable and visionary processor architecture designed by Burton Smith.

## **2. The Micro-threaded Model of Instruction-level Concurrency**

### *2.1. Related work*

Micro-threading was first proposed in 1996 [18] and has been refined and developed in a number of subsequent subsequent papers, e.g. [19,20]. A number of other papers have also considered similar models, the earliest being nano-threads in [21], a limited form of micro-threading 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 [22,23,24,25,26]. More recently, in [27], a thread model called mini-threads has been evaluated with a view to increasing concurrency in a SMT architecture, without increasing register-file size as SMT uses threads that require their own context and architectural register set. Last but not least, threading has been used in a broader context with a single shared register set in dataflow architectures, see for example [28]. A good review on multi-threading is given in [29].

### 2.2. Abstract model

A microthreaded pipeline comprise three major components. An instruction selector, which selects instructions for execution from a number of concurrent code fragments called microthreads. A synchronising memory that suspends instructions until their operands are available and finally one or more functional units that complete the instruction. Each component accepts new instructions/data on every cycle but may take more than one cycle to complete. This is illustrated in Figure 1. Instruction sequences in the microthreads execute strictly in order, so blocking an instruction at the synchronising memory must also suspend any subsequent issue from the same thread.

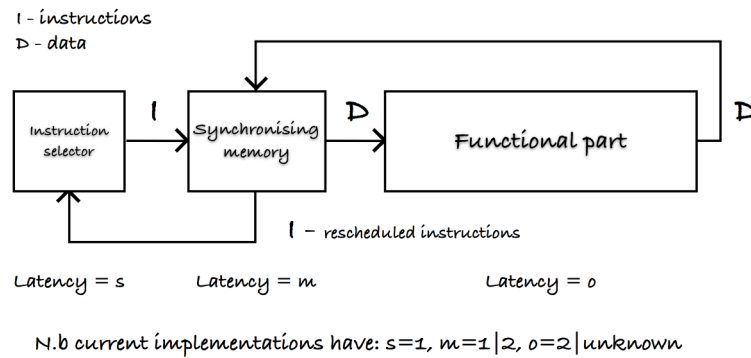


Figure 1: A microthreaded pipeline showing the three components and the flow of data and instructions.

Microthreaded programs are derived from sequential programs by defining a number of stages  $k$ , where execution is passed to a set  $M_k$  microthreads of cardinality  $\mu_k$ . A stage begins by creating the set of microthreads and ends with a synchronisation that defines the termination of all  $\mu_k$  threads. Within a stage, concurrent microthreads communicate values only via the synchronising memory. Non-synchronising memory can be written by microthreads but values written in one stage may only be read by microthreads from a subsequent stage. This defines a bulk-synchronous model of consistency in main memory. The definition of a microthread depends on the implementation of the instruction selector. In its simplest form a microthread is a sequential program with a single entry point, one or more

termination points and no function calls. By providing support for multiple families of microthreads within the instruction selector, microthreads can be defined recursively so that a microthread may also contain concurrent stages. This also allows functions to be defined as microthreads, where parameters and results are managed using dataflow synchronisation in the synchronising memory.

The concurrency in a microthreaded program can be dynamic and is distributed dynamically (although deterministically) to multiple processors. Stage  $k$  of a microthreaded program is executed on  $\pi_k$  pipelines ( $\pi_k \leq \mu_k$ ), where thread  $i$  ( $0 \leq i \leq \mu_k - 1$ ) is distributed to processor  $j$  ( $0 \leq j \leq \pi_k - 1$ ) using a known function  $\delta(i, \mu_k, \pi_k)$ , and executes there to completion. All active microthreads in a pipeline (i.e. those microthreads not suspended in synchronising memory) are stored in an active queue from where they are selected for execution. This selection is data-driven and hence the scheduling of instructions is dynamic and non-deterministic.

Preemption is not implemented at the thread level and all threads execute until they block. Thus the *currently-executing microthread* executes until either it terminates or one of its instructions is suspended. By partitioning instructions by their delay, i.e. whether they can be statically scheduled or not, a compiler is able to annotate instructions in a microthread with a flag indicating context switch points. Annotated instructions trigger the selection of the head of the active queue as the new currently-executing microthread. The old microthread is forwarded to the synchronising memory with the annotated instruction and waits there until the required data becomes available, when it will be rescheduled by adding it to the active queue again. If the annotated instruction does not suspend the microthread it is immediately returned to the tail of the active queue as soon as its data has been read. Stages 1 and 2 of a microthreaded pipeline are illustrated in Figures 2 and 3.

Note that this annotation is an optimisation that removes a control hazard from the pipeline, i.e. a flush on data non-availability at the synchronising memory. As context switching is frequent, this is an important optimisation, which provides efficient pipeline utilisation so long as threads are not starved of data.

To implement microthreading over a conventional ISA requires five new instructions to be added to it. The **Cre** instruction defines a family of microthreads and returns an identifier to a specified register. It also uses a control block that contains meta data defining the family of microthreads and its context. An advantage of this model is that this meta data is dynamic and can be manipulated at run time or even extended without changing the ISA, although extension will require a redesign to the pipeline. It may for example capture operational parameters and constraints related to resource issues. **Cre** has non-deterministic delay in execution as there are global implications for its efficient execution. The **Sync** instruction waits for the termination of the family of microthreads specified in its operand register and returns a value to its target register. For normal termination that value is `Maxint`. This instruction also has non-deterministic delay, as would be expected. A **Brk** instruction forces the termination of all microthreads within the specified family and is provided to support infinite families of threads. This instruction passes a return value to the **Sync** instruction as its termination value. Thus a family of threads either terminates when all threads have terminated normally and gives a return

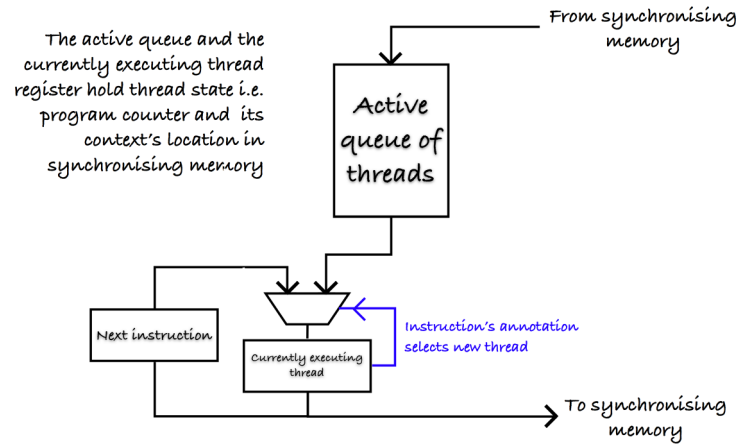


Figure 2: The instruction selection stage of a microthreaded pipeline

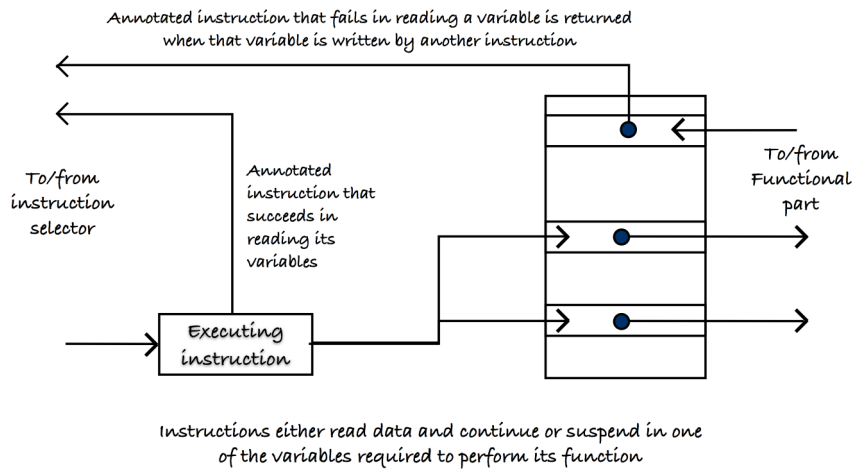


Figure 3: The synchronising stage of a microthreaded pipeline

of Maxint or that family is forcefully terminated by a **Brk** instruction and returns another value.

Two further instructions are provided to preempt a whole family of microthreads. The first **Kill** stops thread creation and terminates all executing threads destroying any synchronising state that may exist. The other **Sqz** stops thread creation and allows all currently executing threads to complete any synchronisation. In this case the family can be restarted as **Sqz** returns the index value of the first thread in the family that was not executed. Thus a squeezed family can be restarted from this index value, perhaps on different resources.

In addition to the instructions defined above, a means of annotating instructions must also be defined. Both context switch and termination points must be identified in a microthread's code. This may be achieved by a control stream identified along with the instruction stream in the meta data for the family or, as in the example below, by using the pseudo instructions **Swch** and **End** that annotate the prior instruction. These pseudo-instructions may be prefetched by the instruction selector concurrently with the annotated instruction, acted on and discarded.

To illustrate the use of microthreaded instructions to create nested concurrent stages, consider the following FORTRAN code performing a matrix-vector multiplication:

```

      m = 1024
      DO 1 i = 1,m
        y(i) = 0
        DO 1 j = 1,m
1       y(i) = y(i) + A(i,j)*x(j)

```

This is translated into assembler in Figure 4 using these new instructions. Note the assembler is MIPS like, with the target register as the first specified.

This binary code defines a million concurrent microthreads which are schedule independent. It can be implemented on from 1 to several thousand processors giving linear speedup. The i-loop microthreads are all independent and the j-loop threads each contain a dependency chain but can provide a limited amount of do-across concurrency. The meta data for these families defines the dynamic loop bounds and specifies a partitioning of the microcontext for each thread into global, shared and local registers. More information about the register specifiers is given in section 2.3 but can probably be ignored on first reading.

The actual number of pipelines used to execute this code would have to be defined before executing the first **Cre**, either statically or dynamically with reference to some operating environment, where an allocation could be done from a pool of processors for each concurrent stage in a program.

On executing the **Cre**, the globals are copied from the top locations in the creating environment's local registers and broadcast along with a pointer to the family's meta data to all processors participating in this concurrent stage. Each processor independently accesses the metadata and, knowing the loop bounds ( $\mu_k$ ), the number of processors allocated ( $\pi_k$ ), and its position in those processors, can create the subset of microthreads it must to execute. These threads are created

*Parallel Processing Letters*

```
i-loop: .data          #Create control blocks
        .word 0       #start index
        .word 0       #finish index
        .word 1       #G, the number of globals registers defined
        .word 0       #S, the number of shared registers defined
        .word 4       #L, the number of local registers defined
        .word 8       #B, the number of threads per allocation round
        .word i-body  #pointer to code for family of threads
j-loop: .word 0       #start index
        .word 0       #finish index
        .word 1       #number of globals registers defined
        .word 1       #number of shared registers defined
        .word 4       #number of local registers defined
        .word 0       #default block size
        .word j-body  #pointer to code for family of threads
main:   Mv $L31 1023   #main code set m-1 in $L31
        Sw $L31 i-loop+4 #set dynamic bound for i-loop threads
        Sw $L31 j-loop+4 #set dynamic bound for j-loop threads
        Mv $L31 1024   #1024 to $L31 is $G0 in i-loop threads
        Cre $L1 i-loop #create a family of microthreads using meta data at i-loop
        Sync $L2 $L1  #wait till all threads in family $L1 are complete
        End           #and that is it!
i-body: Mv $L3 0      #initialise y(i)=0 ($D0 in 1st j-loop thread)
        Mul $L4 $L1 $G0 #set L4 to (i-1*m), top location of locals is global to j-loop
        Cre $L2 j-loop # create inner loop over j
        Sync $L4 $L2   # sync loop identified by $2 return code to $L4
        Sw $L3 y($L1)  # store y(i) $L3 is the last $S0 in the j-loop
        End
j-body: Add $L2 $L1 $G0 #complete index into A, (i-1)*m+j
        Lw $L3 x($L1)  #load x(i)
        Lw $L2 A($L2)  #load A(i,j)
        Mul $L2 $L2 $L3 #note $L2 reused - does not require synchronisation
        Swch          #Both L2 and L3 are from memory - delay is non-deterministic
        Add $S0 $D0 $L2 #this line captures the thread-to-thread dependency
        End
```

Figure 4: Micro-threaded assembler code for matrix vector multiply.

one per cycle to match the highest possible context switch rate. Creation involves dynamically allocating registers for the microthread, creating an entry in the data structures used by the instruction selector, initialising this, checking the I-cache for a hit on the code pointer and adding the microthread to the active queue. The creation also initialises the allocated registers to empty and sets the first (\$L1) to its index value in the family.

Although this example uses homogeneous concurrency, had the loop body been complex enough, further concurrency could be expressed by defining the loop body as a set of concurrently executing heterogeneous microthreads. Similarly, heterogeneous concurrency at a high level can be defined in the same way.

### 2.3. Synchronisation name space

Synchronisation defines a partial order over a given namespace. In the example above, that namespace is very large. Each of the 1024 i-loop microthreads requires 5 register variables and generates 1024 j-loop microthreads requiring 7 registers. This defines a namespace of some 7 million register variables, which must be addressed using a 5-bit register specifier. The solution to this problem is simple, each microthread maintains pointers to the location of its register variables. These are defined on register allocation. The currently-executing microthread provides this information to the pipeline as it executes its instructions. Operand and target registers of an instruction use an address formed from the base address provided by the microthread and the 5-bit offsets from the instruction. In this way, the namespace can be much larger than the physical register pool, which in any case will vary with the number of processors allocated to a stage. One potential problem of this resource limitation is resource deadlock, where a dependency can cause a thread to be unable to release its resources and where it may not be possible to create the dependent thread because of those resources being tied up. In the example above a resource deadlock is possible due to the creation of subordinate threads. It is solved by limiting the resources used by the outer family's threads. The metadata specifies that no more than 8 threads per processor will be created in allocating resources to this family of threads. This allows the subordinate (dependent) threads to be created. This is very similar to k-bounded loops in dataflow terminology.

#### 2.3.1. Partitioning the register file

The microthreaded model captures a number of types of communication, which naturally emerge from a range of high-level language constructs. We can identify this in the example above. The first type of communication is the broadcast of invariant data to all processors involved in executing the family of microthreads. This is the only communication required in executing independent microthreads.

Both families use global data (broadcasts). The loop bound, 1024, is defined in the main body and is used in each outer thread to define the address of the  $i$ th row of A. 1024 is set as \$L31 in the creating thread and read as \$G0 in each outer-loop thread, which after multiplying it with  $i$  (\$L1) ) passes it to each of its  $j$ -loops \$L4 to \$G0 in order to address the  $i,j$ th element of A. The number of globals, G, shareds, S, and locals, L, are all specified in the metadata when creating a family

of threads. In general globals are defined as the top G locations of the creating thread's microcontext.

Globals are loop invariants and may not be redefined in any created thread, as there is no guarantee that other threads on the same processor will have read the global value, as all microthreads executing on the same pipeline will share the same locations for their globals, even access them via the bypass busses. A single offset is required for each family of threads on each processor to access the globals.

Figure 5 illustrates a mapping of six independent contexts onto two processors. Note that two addresses into the register file are required for each context. On processor 0, the family uses the top G locations of the local context of the creating thread as its globals and threads acquire a unique address for their locals when it is created. On processor 1 however, the globals must be dynamically allocated and written to before the threads are created but again this space is shared by all threads in the family on that processor.

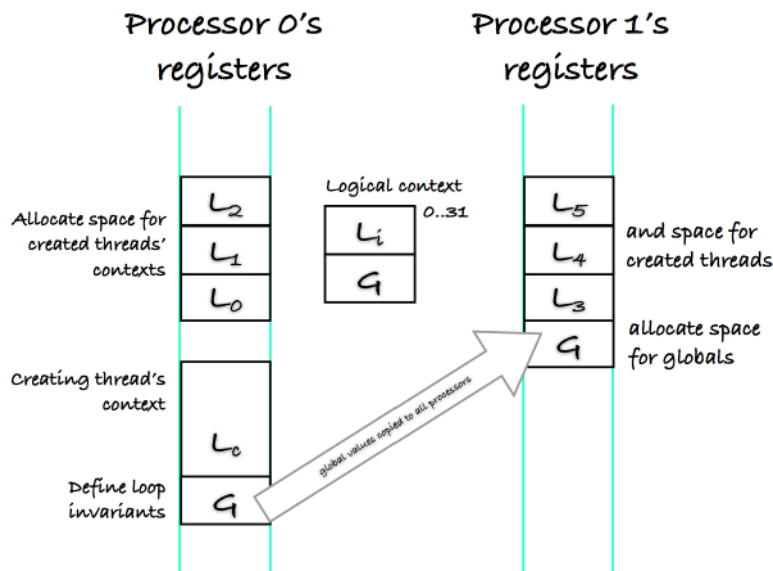


Figure 5: The mapping of independent contexts on multiple processors showing global and local windows within the context and the relationship between the local context of the creating thread and the global context for the created threads.

The j-loop in the example above is a dependent family, where variables are shared between one thread and the next. This regular communication requires two further windows to be defined. The shared and the dependents, represented by \$Si and \$Di in the assembler respectively. The shared registers are written in one microthread and read as dependent registers in the dependent microthread (the next in index order). The shared and dependent variables define a dependency chain through the family of threads. This chain must be initialised before the first

thread and consumed following the last thread. As with globals, the creating thread defines a part of its local context to initialise and consume values at the beginning and end of this dependency chain. Again the number of shared variables is defined in the meta data associated with the creation of a family of microthreads.

The dependency chain maps  $S$  local variables from the creating thread to the dependent variables in the first created thread; the shared variables from the first thread to dependent variables of the second thread and so on; until the shared variables from the last thread are mapped to the same local variables of the creating thread again. In the creating microcontext, the  $S$  local variables below the globals as those shared by the family of threads created. Note that the local created context now comprises  $S+L$  variables when the previous thread is on the same processor or  $2S+L$  when it is not. Communication in this case is implemented as nearest neighbours in a ring for all but the last thread, which may need to forward data over a number of links in a ring if  $\pi_k$  does not divide  $\mu_k$ . Note that this use of overlapping windows also allows bypassing of values between dependent microthreads when they are mapped to the same processor.

Figure 6 illustrates the mapping of four dependent contexts onto two processors. Note that now three addresses into the register file are required for each context. Pointers for global and local contexts are required as before but in addition, a pointer must be stored for each thread in order to access its dependent registers and a bit must be stored to define whether they are on the same processor or not, bit  $R$  in Figure 7.

The communication defined above is static, or to be more precise defined at create time, when the number of processors is known. A generalisation of the communication model to support arbitrary communication between threads in a family can be implemented but this would be dynamic and must use a remote move instruction *Rmv*, which would copy a value in a register location from one context to a location in another context, where the remote context was identified by a dynamic index value (or difference in index values) specified in a second operand register. The implementation of this is more complex any analysis of resource deadlock becomes impossible in such a dynamic model. It would require some form of dynamic deadlock avoidance, which would free resources when a potential deadlock was detected, requiring the spilling of synchronising state to main memory (and managing synchronising events on that state).

### 3. Research directions

#### 3.1. Simulators

The microthreaded model is a promising candidate for chip multiprocessor organisation and has great potential for generating concurrent code from legacy applications. It is currently being investigated in the *Microgrids* project with funding from the NWO and also within a collaborative EU project *AETHER*. These projects approach this model from two different perspectives, namely in the foundations of the model in the former and its application to self-adaptive computing environments in the latter. A simulator for a chip multiprocessor of the model defined above has

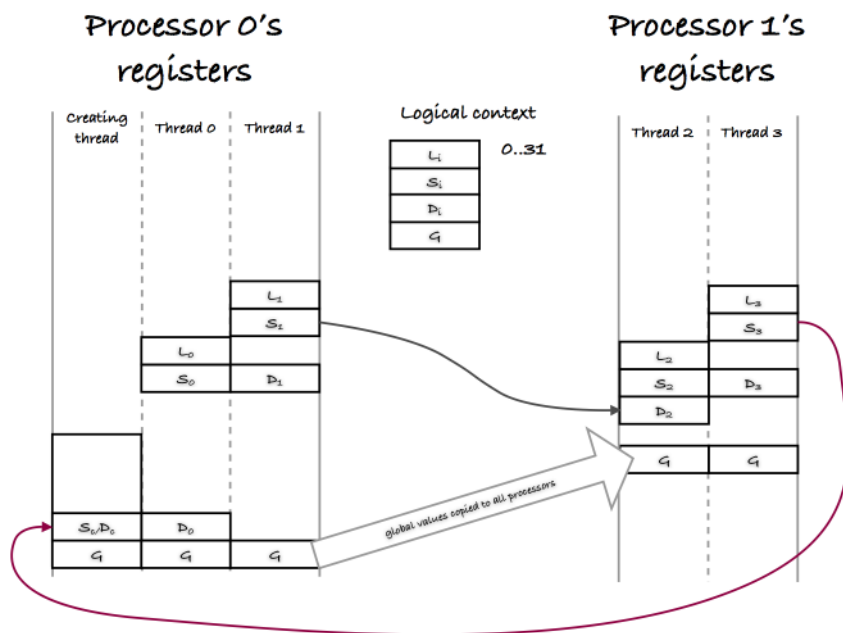


Figure 6: The mapping of dependent contexts on multiple processors showing global, shared, dependent and local windows within the contexts and the relationship between the shared and dependent contexts in a dependency chain.

already been completed and is currently being used to investigate loop kernels and other small code fragments by compiling them by hand. The simulator can run binary code on an arbitrary number of independent, asynchronously-communicating, microthreaded pipelines. The processor currently implements the Alpha instruction set augmented by the instructions defined in this paper. It has been used to produce some preliminary results, published in [20]. These results showed a near linear speedup for independent microthreads over many orders of magnitude by simply executing the same binary on different numbers of pipelines. A maximum IPC of around 1,600 was obtained for 2,048 processors, for a fixed-sized problem. The inefficiencies come only from the amount of work performed per processor, approximately 900 pipeline cycles in the case of 2048 processors and the overhead of distributing the work, which is also of the order of 100s of cycles in the model we were simulating.

We have also investigated energy efficiency by exploiting the fact that we can easily detect when a pipeline's active queue is empty. In this state its clocks can be disabled and its power put into standby mode. The energy dissipated executing the same concurrent stage on different numbers of pipelines was constant over two and a half orders of magnitude, i.e. from 1..256 processors, with minor inefficiencies as the overheads begin to dominate.

### 3.2. Compilers

In order to perform more extensive evaluation of the microthreaded model, compilers have to be developed. The model, being data driven and asynchronous, is more amenable to parallelisation than a model that requires static scheduling to be managed in the code generation phase of the compiler. Even dependent loops can exploit "do-across" concurrency when executed on multiple processors. In this case, the number of processors that can usefully be used is related to the number of independent instructions in the static schedule of the microthreads. The number of processors can be selected dynamically and is dependent on the resolution of dynamic issues, such as pointer disambiguation, which may or may not result in loops being dependent. Such dynamic resolution is only possible because of the dynamic distribution and scheduling in this model.

Of course any microthreads whether dependent or not, when mapped to a single processor will provide tolerance to latency through dynamic interleaving. The compiler exploits this and will generate static sequential schedules for instructions with known delay but annotate instructions to context switch when delay is non-deterministic. The static schedules are executed in order, whereas execution following annotated instructions is scheduled by the instruction selector.

Our strategy for the compiler is to enable the development of a system and tool-chain incrementally and we are currently investigating a number of different compiler frameworks as a basis for the compilation. In order to minimise time to solution we have adopted a low-level, intermediate language based on C, which implements the additional instructions defined in this paper as keywords,  $\mu TC$ .

Our compiler tool chain will then make use of various front-end compilers (as well as manual generation of this dialect of C). Candidate high-level compilers include compilation from sequential C and compilation from data-parallel languages,

such as single-assignment C (SAC) [30]. This strategy allows us to investigate and evaluate code-generation techniques by hand. For example, we will investigate translating benchmarks in C to  $\mu$ TC, evaluating these in our simulator environment and building a set of analysis techniques and optimisations that can be applied automatically. To illustrate  $\mu$ TC, the example coded in FORTRAN above has been translated into the following  $\mu$ TC thread definition, which accepts arrays A, x and y and array size m:

```
thread matvec(real *A, *x, *y; int m);
{
  create (i_fid; 0; m-1; 1; 4)
  {
    index int i;
    real s=0.0;
    create(j_fid; 0; m-1);
    {
      index int j;
      shared real s;
      s=s+A[i][j]*x[j];
    }
    sync (j_fid);
    y(i)=s;
  }
  sync (i_fid);
}
```

It can be seen that this code reflects the loop structure of the original code and the transformation from sequential C requires only that the loop control constructs are replaced by `create` constructs, where concurrent execution is possible. This is true for both dependent and independent loops. The `create` construct supplies parts of the meta-data required for the concurrent sections. The remaining meta data, such as the extents of the various register windows, will be generated by the  $\mu$ TC compiler.

### *3.3. Implementation*

The design and implementation of a microthreaded pipeline in VHDL has already begun. The work to date has focussed on the support structures, e.g. the instruction selector, the synchronising memory and their interaction. The data processing part of the pipeline is no different to a conventional design. One of the fundamental issues that underpins this work is to ensure that implementations are scalable. By this we mean that the area occupied on chip is proportional to the number of instructions issued per cycle (physical concurrency) also that the area occupied is proportional to the amount of latency tolerance required (virtual concurrency) and that both can be scaled over orders of magnitude. This can only be achieved if both the instruction selector and synchronising memory can be distributed to each pipeline on the chip multi-processor. In synchronising memory, communication between processors

is required to distribute a family and in communicating dependencies between its threads. An investigation in [20] looked at the distribution and frequency of the accesses to the asynchronous (non-pipeline) ports in the synchronising memory. It concluded, using a static analysis of a range of different code kernels, that a distributed-shared synchronising memory could be implemented with 5-ports per processor, where three ports provided single instruction issue per cycle and the other two asynchronous ports were able to manage all other demands on the local register file. We have concluded from this study that each pipeline could support 512 synchronising registers in an area less than a 64-bit FPU, which would support of the order of 100s of local concurrent threads. Area estimates for the register file and thread management can be found in [31].

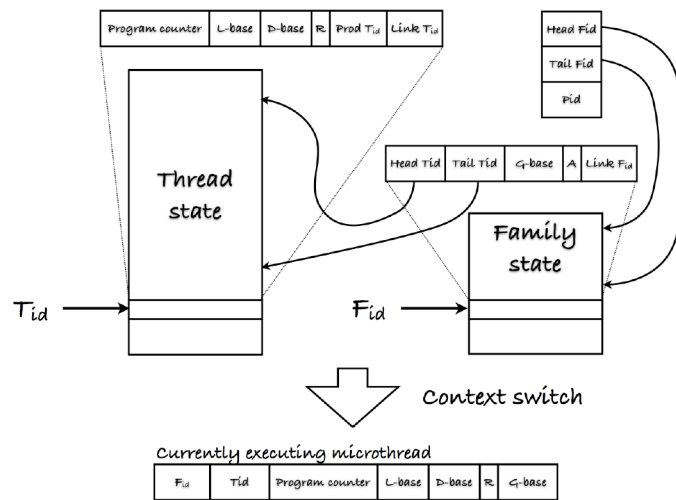


Figure 7: The data structures managed in the instruction selector, both of which are random-access memories.

The ISA and its implementation have been designed to support the concurrent creation and execution of microthreads. A design goal has been to support concurrently, one thread creation, one context switch and one thread reschedule from the synchronising memory per cycle, as we have written and simulated code that requires this in order to maintain a full pipeline. This stringent limitation, coupled with the requirement for scalability means that all thread management should be performed using direct access memories for all state accesses. Figure 7 shows the structures and state used in thread manipulations. Active queues are managed in the thread state memory, called CQ in [31]. Notice that both thread and family id ( $T_{id}$  and  $F_{id}$ ) are in fact addresses into the respective memories. The active queue, a queue of queues, is maintained using pointer fields in the state stores with head and tail pointers stored for each family. The memory required is similar in complexity to the synchronising memory and [31] shows that even considering all concurrent accesses specified, the size of a 256 entry thread-state memory is smaller than the

synchronising memory.

#### 4. Conclusions

This paper has presented a model of concurrency that supports a massive number of threads organised with no or little overhead in terms of thread creation, context switching and synchronisation. The model is incremental and thus offers backward compatibility with legacy binary code. Moreover, by recompiling or translating the binary code once only, schedule-invariant code is obtained that can be run on an arbitrary number of processors up to some dynamic, code-defined limit. The paper presents the model and the novel register mapping that supports efficient inter-thread communication. The model is scalable, being implemented on arrays of processors connected only by a broadcast network (with distributed synchronisation) and a nearest neighbour ring network for inter-thread communication. Moreover, results presented elsewhere have shown that implementations of the two major structures in a microthreaded pipeline are scalable and manageable. These properties, together with the potential for globally asynchronous, locally synchronous implementation make microthreading a potentially attractive model for future chip-multiprocessors.

#### 5. Acknowledgements

The support from the Dutch Research Council NWO for the *Microgrids* project and from the European Union under its Framework 6 research program for the *AETHER* project is gratefully acknowledged.

#### 6. References

- [1] R W Hockney and C R Jesshope (1981) *Parallel Computers*, Adam Hilger Ltd., ISBN 0-85274-422-6.
- [2] R W Hockney and C R Jesshope (1988) *Parallel Computers 2*, Adam Hilger Ltd ISBN 0-85274-811-6, ISBN 0-85274-812-4 (Pbk).
- [3] D Levine, D Callahan and J Dongarra (1991) A comparative study of automatic vectorizing compilers, *Parallel Computing* **17** (10-11): 1223-1244.
- [4] C R Jesshope (1982) Programming with a high degree of parallelism in FORTRAN, *Comp. Phys. Comm.*, **26**, pp237-246.
- [5] V B Muchnick and A V Shafarenko (1996) *Data Parallel Computing: the Language Dimension*, Thompson Publishing.
- [6] P Mattos (1984) The transputer, *New Electronics*, **17** (16) August 1984, pp4345.
- [7] M Homewood, D May, D Shepherd and R. Shepherd (1987) The IMS T800 Transputer, *IEEE Micro*, October 1987, pp10-26.
- [8] C Lazou (2002) The Japanese Earth Simulator: a challenge and an opportunity, *Primeur monthly*, <http://www.hoise.com/primeur/02/articles/monthly/CL-PR-06-02-1.html>
- [9] 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.
- [10] R P Peterson et. al. (2002) Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading, *ISSC Digest and Visuals Supplement*.
- [11] V Agarwal, H S Murukkathampooni, 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.
- [12] 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
- [13] S. Rixner, et al. (2000) Register organization for media processing, *In Proceedings of the 6th International Symposium on High-Performance Computer Architecture* (January 2000), pp 375386, IEEE.
- [14] A. Gupta and W. J. Dally (2005) Topology Optimization of Interconnection Networks, *Computer Architecture Letters*, Volume 4, July 2005.
- [15] D. Burger, S.W. Keckler, K.S. McKinley, et al. (2004) Scaling to the End of Silicon with EDGE Architectures, *IEEE Computer*, **37** (7), pp. 44-55, July, 2004.
- [16] S. Swanson, A. Schwerin, A. Petersen, M. Oskin and S. Eggers (2004) Threads on the Cheap: Multithreaded Execution in a WaveCache Processor, *Proc WCED in conjunction with ISCA*, at June 2004.
- [17] M. Bedford Taylor, W. Lee, J. Miller, et. al. (2004) Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams, *Proc ISCA*, June 2004.
- [18] 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.
- [19] 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, LNCS2823 (Berlin, Germany), ISSN0302-9743, Aizu, Japan, 22-26 Sept 2003.
- [20] K. Bousias, N. M. Hasasneh and C. R. Jesshope (2006) Instruction-level parallelism through Microthreading - a scalable Approach to chip multiprocessors, *The Computer Journal*, **49** (2), pp211-233.  
Computer Journal, 49 (2) pp211-233.
- [21] L Gwennap (1997) DanSoft develops VLIW design. *Microproc. Report*, **11**, 2 (Feb. 17), 1822.
- [22] 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.
- [23] R Balasubramonian, S Dwarkadas, and D H Albonesi (2001) Dynamically allocating processor resources between nearby and distant ILP, *Proc. Intl. Symp. on Computer Architecture*
- [24] R Chappell, J Stark, S Kim, S Reinhardt, and Y Patt (1999) Simultaneous subordinate microthreading (SSMT), *Proc. Intl. Symposium on Computer Architecture*.
- [25] 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*.
- [26] C Zilles and G Sohi (2001) Execution-based prediction using speculative slices, *Proc. Intl. Symposium on Computer Architecture*.
- [27] J Redstone, S Eggers and H Levy (2003) Mini-threads: increasing TLP on small-scale SMT processors, *Proc 9th Intl. Symp. On High Performance Computer Architecture (HPCA-9)*, p19, IEEE.
- [28] R S Nikhil, G M Papadopoulos and Arvind (1992) \*T: A multithreaded massively parallel architecture, *Proc. Intl. Symposium on Computer Architecture*.
- [29] T Ungerer, B Robic and J Silc, M (2002) Multithreaded Processors, *The Computer Journal*, **45** (3), pp320-348, British Computer Society.

- [30] S-B Scholz (2003) Single Assignment C - Efficient Support for High-Level Array Operations in a Functional Setting, *Journal of Functional Programming*, **13**(6), pp1005–1059.
- [31] Bell, I, Hasasneh, N and Jesshope C R (2005) Microgrids and Micro-contexts: Support Structures for Microthread Scheduling and Synchronisation, to be published in [?] (Special issue and Proc. 1st MicroGrid Conference, Amsterdam, July 2005). Preprint located at: <http://staff.science.uva.nl/~jesshope/Papers/IJPP.pdf>.