

# A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors

T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol and L. Zhang  
Institute for Informatics, University of Amsterdam  
Amsterdam, Netherlands

**Abstract**—This paper presents a concurrent execution model and its micro-architecture based on in-order RISC processors, which schedules instructions from large pools of contextualised threads. The model admits a strategy for programming chip multiprocessors using parallelising compilers based on existing languages. The model is supported in the ISA by number of instructions to create and manage abstract concurrency. The paper estimates the cost of supporting these instructions in silicon. The model and its implementation uses dynamic parameterisation of concurrency creation, where a single instruction captures asynchronous remote function execution, mutual exclusion and the execution of a general concurrent loop structure and all associated communication. Concurrent loops may be dependent or independent, bounded or unbounded and may be nested arbitrarily. Hierarchical concurrency allows compilers to restructure and parallelise sequential code to meet the strict constraints on the model, which provide its freedom from deadlock and locality of communication. Communication is implicit in both the model and micro-architecture, due to the dynamic distribution of concurrency. The result is location-independent binary code that may execute on any number of processors. Simulation and analysis of the micro-architecture indicate that the model is a strong candidate for the exploitation of many-core processors. The results show near-linear speedup over two orders of magnitude of processor scaling, good energy efficiency and tolerance to large latencies in asynchronous operations. This is true for both independent threads as well as for reductions.

## I. INTRODUCTION

The enforced shift from implicit to explicit concurrency in processor design is giving the computer industry many problems for which it has few solutions. As has been shown by decades of research, concurrency is difficult. There is no doubt that solutions will require many simpler cores rather than a few complex ones in order to decentralise processor design and mitigate the power barrier. Indeed, this has already been achieved in the embedded systems domain [1], where chips already exist that can execute in excess of one thousand instructions in every clock cycle. The main problem is the ability to program these cores or, to be more specific, to program them safely. The embedded market has different demands in this respect to the commodity market, as it is focused on only a few applications. Another exception is the server market, where there is natural concurrency and the Sun UltraSPARC T1 (Niagara) processor [2] has been developed primarily for this market.

It is an open question whether we can program multi-

cores efficiently and safely for generic markets. This paper describes general-purpose solutions and builds on the premise that what is missing is a coherent model of concurrency and the assumption that such a model must be built into the ISA of the processor cores. However, any solution must respect the constraints that semi-conductor design engineers are now facing [2], namely: an increasing power density and the requirement for power management [3]; a reduction in reliability caused by vulnerability to soft errors [4]; and high latency in both on- and off-chip communication.

These predicate constraints in processor design and the approach taken must be scalable and also conservative in the execution of instructions (not speculative), in order to provide power-efficient solutions. Models based on transactional memory [5] do not respect producer-consumer synchronisation (i.e. dataflow) and hence do not guarantee either of these characteristics. Any approach must also reflect the asynchrony implied by highly latent communication. To be efficient, the unit of concurrency scheduled must be as small as possible and scheduling overheads should be low or well hidden. Superscalar processors use the data-flow firing rule and have fine-grain concurrency but fail, as the data-flow graph they operate on is limited by the physical register file size and uses unscalable, centralised scheduling. Moreover, it is speculatively derived. In [6], Papadopoulos illustrates that contextualisation of the synchronising memory is necessary for efficient dataflow implementation.

The dataflow model [7], is one of the major contenders in meeting the above criteria. Unfortunately this model is relatively inefficient and also has difficulty in capturing the imperative programming style. The former is illustrated in [7], which describes a loop summing its own index that requires seven instructions in its body, six instructions of overhead for a single operation. For comparison, the sequential model has an overhead of two instructions to implement the same loop and more to the point, the micro-architecture described here has a zero-instruction overhead. Dataflow instructions target other instructions, not registers, causing a logarithmic overhead to broadcast a value to  $n$  operations, whereas with register targets, a write and multiple reads is all that is required.

Another research question answered in this paper is whether the beneficial characteristics of the dataflow model, namely scalability, asynchrony, latency tolerance and conservative

instruction execution, can be implemented with the efficiency of a classical, in-order RISC processor. To answer this question we present and evaluate a concurrency model and its implementation as a micro-architecture. The model is derived from the DRISC processor proposed by Bolychevsky et. al. in 1996 [8] and extended by Jesshope [9], as a general concurrent computational model. It captures all data-dependencies in a parallel program but does so in a more constrained manner than dataflow. What it gains from this loss of concurrency is an efficient implementation as a RISC processor. The novel aspect of the model is that it is based on hierarchies of families of identical threads, where data-dependencies are captured by shared register variables between threads in addition to bulk synchronisation on memory. The remainder of this paper describes the model and related micro-architecture and presents simulation results that demonstrate the scalability of this approach and an analysis of the control structures required for its implementation.

## II. THE MICROTHREAD MODEL

### A. Families of threads

The microthread model is based on named families of identical, blocking threads, where each thread is distinguished by a unique index value in some predefined range, which may be unbounded. The family of threads (rather than the thread itself) is the managed unit of work, which is created by the models *create* action. A family may be asynchronously terminated or pre-empted by two further actions, *kill* and *squeeze* respectively. The advantage of this approach is twofold. It allows for the management of concurrent programs from workflow to ILP with a uniform set of actions and it amortises the cost of concurrency creation when many small and homogeneous microthreads are required, i.e. in loops. An implementation will partition a thread's context into two components; there is a shared part stored by family, which includes information about the code, various parameters describing the scope of the family and its global state. In addition, information is stored per thread on its execution state including how threads communicate with each other.

### B. Microthreads

Microthreads are strict sequences of actions (instructions), which block if their operands are not available. Each thread has a context of synchronising variables supporting asynchronous communication. This context is divided into different classes of variables. To support communication between threads, a *shared* variable is written by one thread and is read in one other thread, its successor thread in the family. An instruction attempting to read an empty shared variable will block the thread to which it belongs until the data is written. These variables may be in one of three states: empty, full and waiting. They are initialised in the empty state, become full when written and contain a thread reference when empty for the purpose of continuing that thread on a data write. These register variables are dataflow i-stores, restricted in that only a

single continuation is allowed and relaxed in that once written they can be read and written any number of times.

Another class of synchronising variables are *local* and these can be used for managing asynchronous actions within a single thread, typical examples include loading external data or acquiring resources, where delays are not known and can not be statically scheduled. These actions will set their target variable to empty when executed, decoupling the execution of that action, which then writes the variable asynchronously. The final class of variables is *global* and these are read-only values shared by all threads in a family.

Communication using shared variables is restricted to give the model its freedom from deadlock and to capture locality. A creating thread may share a variable with the first thread in the family it creates and any thread in the family may share a variable with its successor thread in the family defined by its index sequence. Finally the last thread in canonical order makes its shared variable available to the creating thread but only when the family has completed.

### C. Family management

Threads may also read and write regular memory, which is asynchronous and would typically be implemented as a distributed-shared memory. Synchronisation on this memory is provided on family termination, i.e. when all threads in the family have terminated and when all writes to memory by those threads have completed. This bulk synchronisation, together with the parent-child and intra-family synchronising communication, is sufficient to capture all dependencies in a program. For determinism however, two concurrent threads may not read and write or both write the same location in memory.

Termination is achieved by making the create action asynchronous and having it set a local synchronising variable in the creating thread with a return code, which identifies how the family terminated, i.e. resulting from normal termination, or being terminated by a *break* action, or the asynchronous squeeze or kill action.

Executing a break action in a thread terminates its own family and any subordinate families and provides a return value, which sets a local in the context of the creating thread. The squeeze action also provides in a return value, which is the squeeze index. The squeeze action pre-empts a family and its subordinate families by terminating a family and identifying the index in each family (the squeeze index) where the execution was terminated. This is captured as if threads were executed sequentially in order to minimise the state required to capture the state of the pre-empted unit of work. This squeeze index identifies the starting point for resuming the pre-empted unit of work in every family recursively.

### D. Places

The model described so far is abstract and deterministic. However, all models must deal with physical resources and this concept must be managed uniformly in the model's various implementations. The model therefore introduces the

abstraction of a place at which a create action is executed. The implementation of the model then defines the specific format and semantics of a place. However, in order to enforce static analysis of resources, two model-defined places are identified. The *default* place is resource naive and an implementation is free to choose how threads are distributed. In the implementation described here threads will be distributed to the same processors as its parent family. The *local* place, on the other hand, forces thread creation to be confined to the same processor as the creating thread. Any other place is a variable of an implementation-specific type, set by a place server and used by create, which forces thread creation to be delegated to the place named in that variable. In the implementation described in this paper, places are clusters of DRISC processors configured into rings and threads are distributed deterministically to those processors using a block parameter and a block-cyclic distribution. Further information about the model, which is also called SVP (SANE virtual processor), is given in [9].

#### E. Programming in the model

The strategy for programming any implementation of this model, whatever the level of granularity, is to use a sequential language, such as C. Parallelising compilers are normally considered difficult but this is because they must map and schedule any concurrency they extract. In this model, mapping and scheduling of threads is managed entirely in the implementation of the create action and parallelisation is significantly simplified. The strategy for compilation is to capture all loops and function calls using the create action. For functions, shared variables will be used to synchronise calling and called threads. Independent loops are not a problem but for dependent loops, equivalence transformations are introduced to force the code conform to the model's restrictive communication. These transformations use to either shared variables or bulk synchronisation on memory to capture the dependencies in the loop. These include:

- non-local regular dependencies, e.g.  $a[i - k]$ , which use a parallelising transformation yielding an independent family of order  $k$  and a subordinate dependent family with a local dependency that can use shared variables;
- multiple dependencies on the same index, e.g.  $a[i - 1] + a[i - 2]$  use multiple shared variables, with static routing between them;
- dependencies on different indices, e.g.  $a[i - 1] + a[j - 1]$  use a parallelising transformation yielding a dependent family on the diagonal forming the computational wave-front, in this case over  $k$ , where  $i + j = k$ , and an independent one over the orthogonal hyper-plane that is the computations wave-front. The dependent family synchronises the sequence of independent families in memory.

A parallelising compiler is under development based on the CoSy framework, an industry-standard C compiler. The target for this compiler is the  $\mu TC$  language. Microthreaded C ( $\mu TC$ ) [13] is based on the C99 standard by extending it

with new types and constructs, which capture all the concepts and actions in the microthread model described above. Two implementations of  $\mu TC$  now exist. The first is based on version 4.1 of the GNU C compiler and compiles  $\mu TC$  to produce assembly code that can be run on Microthreaded processors [15]. This development is almost complete. The second implementation is based on a tool chain that translates  $\mu TC$  programs into C++ code that can be linked with a library based on POSIX threads. This allows  $\mu TC$  programs to be run on conventional processors, albeit with substantially increased thread creation and synchronisation overheads than possible in a DRISC processor, i.e. some four orders of magnitude slower!

### III. MICROGRIDS OF MICROTHREADED PROCESSORS

A microgrid is a chip-multiprocessor, using processors that implement the microthread model directly and in this section both micro and chip architecture are described. The processor is based on a DRISC version of an Alpha processor, which uses in-order issue pipelines with both in-order (synchronous) and out-of-order (asynchronous) completion of instructions. L1 D-cache misses, thread creations and FPU operations are all implemented asynchronously and dependent instructions are synchronised using the local registers. The processor has small caches and a relatively large register file that implements the threads contexts of synchronising variables. The latter has two additional state bits to implement i-store functionality. The processor also has structures to allocate and manage both families and threads.

#### A. Thread management

A family of threads is created with an upper bound on the number of concurrent threads per processor, which is the block size specified in the create instruction. On executing the create, registers for that number of threads are allocated dynamically as a single contiguous block from unallocated registers in the local register file, further constraining the block size if fewer registers are available. Once the create instruction has a block of registers, it can create a new thread every cycle until either there are no more thread-table entries or the allocated registers have been exhausted. As soon as a thread terminates its registers and thread-table entry are reused until all threads in the family have been created or the family is terminated by a break, squeeze or kill instruction. The create action requires an instruction to allocate a family table entry, a number instructions to set the family's parameters (common cases require fewer instructions, e.g. a function call requires none) and an instruction to initiate the create, which contains a code reference.

All information required to execute a thread is stored in the fixed-size thread table. The entries in this table contain information, such as program counter, state, and register-file addresses. At any given time, thread table entries can be in one of six states. These states, and their transitions are illustrated in Figure 1. For three of these six states, empty, waiting and ready, the threads are managed by putting them on a linked list. To this end, every thread has a next field in the Thread

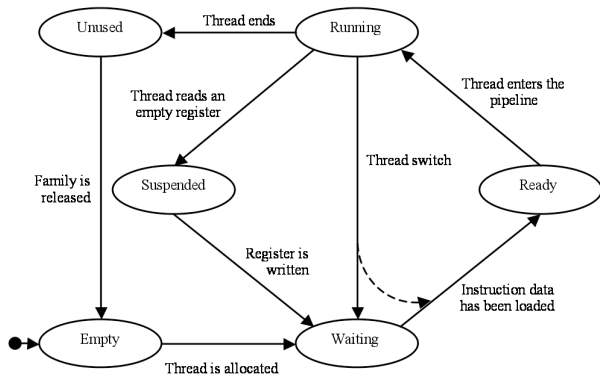


Fig. 1. Thread states and their transitions

Table. Both empty and ready threads are put on processor-global linked lists, where the empty list contains entries that can be used to allocate new threads and the ready list contains threads that can be executed in the pipeline. Waiting threads are threads that are waiting on the I-Cache line that holds their next instruction. They are linked in a list with a pointer stored in the cache-line. This allows the I-Cache, when a line has been read, to immediately append all threads waiting on that line to the ready list, making them available to the pipeline. Threads are also maintained in a per-family membership list (requiring a second next field for each thread), to which they are added when they are allocated. This list allows the processor to append all threads of a family, regardless of their state, to the empty list when the family is forcibly terminated. The other three thread states are implicit in that the thread is not specifically marked or put on a list for those states. Running threads are those that are currently in the pipeline. From this state, those threads can terminate, proceeding to the unused state where it only exists on the per-family membership list, or proceed to the suspended state, where a reference to the thread is held in a register that was read as empty, or to the waiting state following a context switch when the thread has to re-check its I-Cache line. Note that if the I-Cache line is present the waiting state is bypassed and the thread is appended to the ready list.

### B. Family management

All family related information is stored in the family table. This includes the number of threads created, parameters for creating new threads, including the pointer to the code and pointers into the thread table for the membership list. The allocate instruction identifies and initialises with defaults a free entry in this table. Further instructions may then overwrite these defaults and when set up, the family's threads will be created.

When the family has terminated and no more references to it exist, the entry is cleaned up and can be reused for another family creation. Because the family table is of a fixed size, a possibility of resource deadlock can occur when there is no more space left to create a family and all existing families require one to be created (e.g., recursive family creation). To

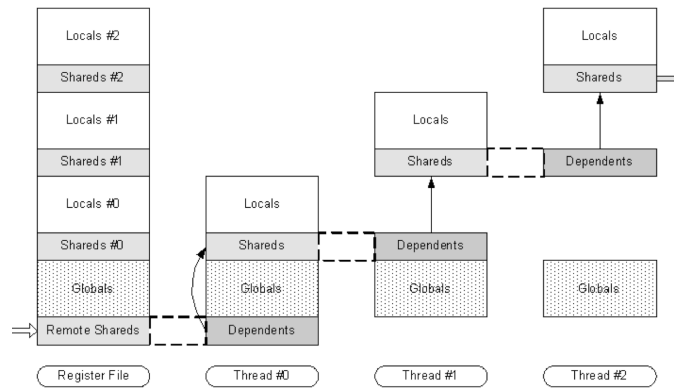


Fig. 2. Register allocation and context mapping for a block of three threads per processor. Arrows represent a thread reading a dependent and writing a shared, which is the only communication pattern through a family of threads via shared synchronising registers.

avoid this problem, family creation events can be delegated to another group of processors. This delegation can be done explicitly by the program or implicitly by the hardware. This allows family creation to continue as long as there are processors available, somewhere.

### C. Register contexts

Each processor has a large register file to allow many concurrent threads to have their own context of synchronising registers. As already described, when a family is created, a single block of registers is allocated in the register file. This space is divided into regions for each thread, with partitions for the model's different register classes and pointers to these are stored in each thread's thread table entry. These partitions are illustrated in Figure 2. A block for remote shares is also allocated but not initialised. The globals are then allocated and initialised from a subset of the creating threads context. Each thread then has its own block, which is divided between locals, shares and dependents. The shares are written by one thread and the dependents provide the address for the dependent thread to read those values. The implementation manages thread mapping and if both threads are created on one processor, shares and dependents share a single location in the register file. However, if producer and consumer are mapped to different processors, reading or writing the model's shared variable will induce a communication between the shared and dependent registers where the remote shares are copies of the remote shared registers. On allocating a thread, all of its registers are set to empty, and the thread's index in the family is written to the first local register of that thread. The thread code specifies the number of local, global and shared registers when the family is created, whose sum can be less than the ISA's address range.

### D. Caches and memory interface

The instruction and data caches in a microthreaded processor are conventional caches, with some additional fields in each cache-line to allow for decoupled memory accesses.

These caches are smaller than in conventional processors, since the microthreaded processor is latency tolerant. Each line in the Instruction Cache has a pointer into the Thread Table to allow the processor to construct a linked list of threads waiting on that cache-line. Whenever a thread is activated, the I-cache is checked to see if the threads instruction-line is present. On a miss, a line is selected for replacement, cleared and the thread is put into the linked list for that line. Should the line already be present, but is still being fetched from memory, the thread will simply be appended to the linked list for that line. When the cache-line has been read from memory, the processor can append the entire linked list for that line to the ready list, from which the pipeline will select threads to run. To prevent cache-lines from being evicted while being used by threads in the ready list, a per-cache line counter counts the number of threads in the ready list that still need this line. The line-replacement algorithm (i.e., LRU) is adjusted to only select lines, which have a counter value of zero.

The D-cache supports decoupled memory operations by using a pointer into the Register File for each cache-line. When a read from memory into a register misses the cache, or hits a cache-line which is still being fetched, the read information (number of bytes and offset in the cache-line) is stored in the target register, and the register is appended to the linked list of registers for that cache-line. Thus, the empty register acts as an entry in a memory read buffer. When the cache-line has been fetched from memory, it is appended to the processing list, a linked list of data-cache-lines that require their reads to be processed. Every cycle, the processor will service a read request from the linked list of reads writing the data to the register files asynchronously.

Since memory operations are asynchronous, all operations are tagged with a value that identifies the operation. The response for each memory operation must contain the same tag as its request. This allows the processor to properly handle the request when it is received. Since there are three kinds of requests, I-cache line reads, D-cache line reads and Data writes the tag is merely two bits identifying this type, and an index into the cache lines for the I-Cache or D-Cache, depending on the response type. Using these tags, the interface to memory is very simple; all that is required from memory is that it can accept and respond to these tagged requests; the responses may arrive in any order, are identically tagged.

### E. On-chip memory model

The memory system in a microgrid must provide the abstraction of a shared memory but achieve this across potentially thousands of processing cores, while providing scalable throughput both on and off chip. The ameliorating factor in this difficult design is that DRISC processors tolerate a large amount of latency, which has led us to adopt and specialise the COMA memory model [10][11]. The memory system includes an on-chip cache and one or more off-chip communication interfaces to a multi-bank memory system. On chip there are just two levels of cache. Each core has its L1 data and instruction caches and the COMA system supports multiple

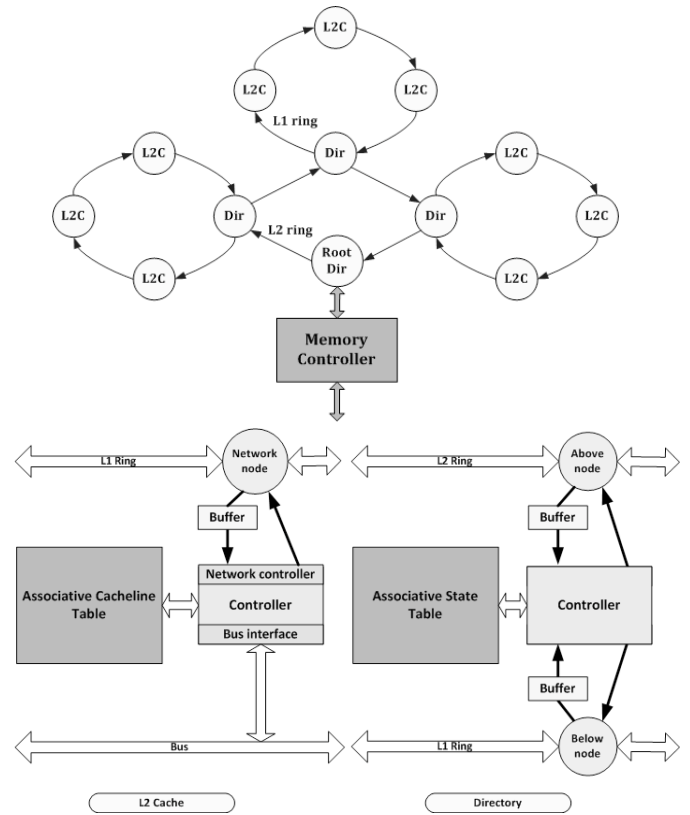


Fig. 3. 2-level hierarchical ring network of L2 caches and L2 cache structure with bus interface to cores, network interface to other L2 caches or directory and directory structure with below network interface connecting L2 caches and above network interface connecting other directories.

L2 caches each of which is connected via a snooping bus to a number of DRISC cores, as shown in Figure 4. Scalability constraints limit this to a small number of cores and 4 or 8 are assumed in the memory simulation. To support thousands of processors on chip, a large number of L2 caches are distributed across the chip in a hierarchical ring network, as shown in Figure 3, where L1 rings connect L2 caches and L2 rings connects directories.

When an L2 cache suffers a cache miss, it will place a request on the ring network and a responding cache will replace this with a miss-reply request containing the requested cache line, if it exists on this ring. Directories only hold state information cache lines and use this information to check whether the request can be served within the local ring or not. They therefore direct the request to a higher ring, if necessary, where it may be passed down again or off chip to be satisfied. The on-chip COMA for microgrid has one significant difference with the KSR [10], where a cache is blocked on an outstanding request. The microgrid memory must implement a lock-up free cache and its protocol implements several transient states on top of a basic MOSI protocol, in order to identify outstanding requests. More details on the protocol and its verification are given in [12].

The memory consistency model plays an important role in balancing programming complexity and system performance.

Since the microthread model only requires memory synchronization on a family create and termination, the memory model implements location consistency (LC) [14]. This is one of the weakest consistency models, where instead of enforcing a unique stored value to be observed by other processors, LC allows processors to read from a set of recently stored values. This combination of on-chip COMA and LC provides effective utilisation of the necessarily limited off-chip memory bandwidth.

#### IV. CHIP ARCHITECTURE

The chip architecture for the microgrid is designed to support thousands of processing cores in a single chip, connected by a variety of on-chip networks and is illustrated in Figure 4. This pool of DRISC cores will be configured (either at design time or at run time) into clusters, which implement the models abstraction of a place. Clusters may be of any size and, if preconfigured, will be heterogeneous. A thread may then delegate a unit of work, e.g. a family of threads and any subordinate threads, to a place or cluster using a chip-wide, packet switching network, the delegation network. This network must address every pre-configured place or every configurable component in order to perform delegation. Since the information content for this is comparatively small, this network has a low-bandwidth and small buffer-size. For a delegation, a number ( $\geq 2$ ) of 90-bit messages are required and this will be implemented with a 9-bit, dimension order, VCT grid network, requiring just 180 bits of buffering per node.

The second network, the COMA network, provides most of the on-chip bandwidth. This is a cache-line wide hierarchy of rings connecting the L2 cache blocks. This network implements the cache-coherency protocol described in Section 3.5 and [12]. It also provides the interface to the off-chip world.

The final network, the protocol network, is local to a processing cluster and implements the models actions over the cluster of processors, i.e. family creation and termination. It is a ring network between each processor in the cluster that uses token propagation to initialise each processor on create, including the global variables, and to signal termination, however it occurs (i.e. normal, break, kill or squeeze). Within a cluster, each processor has a dedicated bus to its neighbouring processors to implement register sharing and a cache-line bus to share an L2 cache between a number of processors.

#### V. EVALUATION

An initial evaluation of the microthread model has been undertaken using a cycle-accurate, software emulation of the micro-architecture described in Section III. This executes of a unit of work (family and any subordinate families) on a variable sized cluster of processors connected in a ring network. It does this from a cold-start and characterises the delegation of code to a place in the microgrid. Since the  $\mu TC$  compiler is still being developed the results presented emulate small hand-compiled kernels, which nevertheless are representative of the computation found in many large-scale applications. Our test

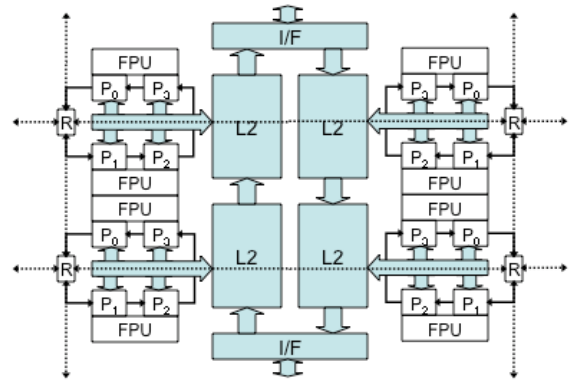


Fig. 4. Microgrid tile of 4 clusters of 4 cores ( $P_0..P_3$ ) sharing an FPU between 2 cores and an L2 cache between 4. It illustrates all on-chip networks, two local rings between processors and L2 caches and routers (R) for the chip-wide delegation network.

kernels include several Livermore kernels, both independent and dependent ones, a micro-threaded version of the sine function using a Taylor series expansion, which is small and very sequential and the fast Fourier transform.

The motivation for this work is both the validation of the models implementation as well as an initial evaluation of its performance, scalability and latency tolerance. The COMA memory system described in Section 3.5 is not yet incorporated into this emulation and the results presented here uses two extremes of memory implementation. The first is sequentially and the second is an idealised parallel memory that is capable of handling multiple requests in parallel. Both have parameterised latency and the latter a parameterised number of banks. Unless otherwise indicated, each processor in the cluster has a family table with 64 entries, a thread table size of 256 entries and a register file with 1024 registers. L1 I- and D-cache sizes were set to 1KB per processor for these results, except where indicated for comparison.

##### A. Data-parallel code

The first set of results is for the data-parallel Livermore kernels 1 and 7 executed over a range of processors. Compiling such loops for a micro-threaded processor is simple, the loop is captured by a single create and the thread code captures the loop body, minus the loop-control instructions (increment and branch). This code contains only local and global registers. The results record the execution time (in cycles) required for a problem size of 64K iterations. Figure 5 shows the speedup achieved for those kernels relative to its execution on a single processor.

As can be seen, the speedup scales almost linearly with a deviation from the ideal of 20-40% at 128 processors due to the start-up involved. Kernel 7 has more memory references than kernel 1 and saturates earlier. Both kernels were re-executed with a 32 KByte cache to show the effect of D-cache size on performance. As can be seen this improves performance but not significantly and shows the latency tolerance of the processor. Neither experiment is limited by memory bandwidth

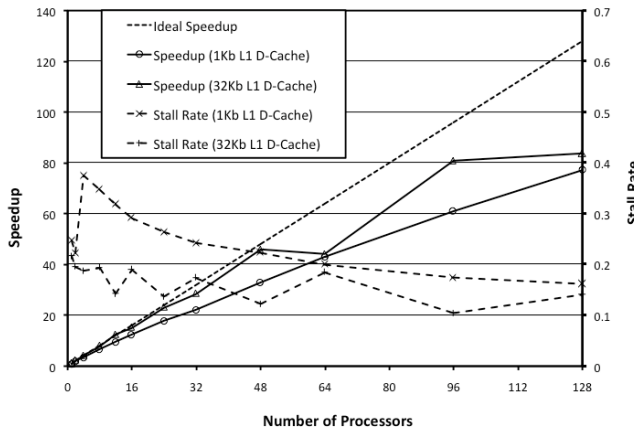
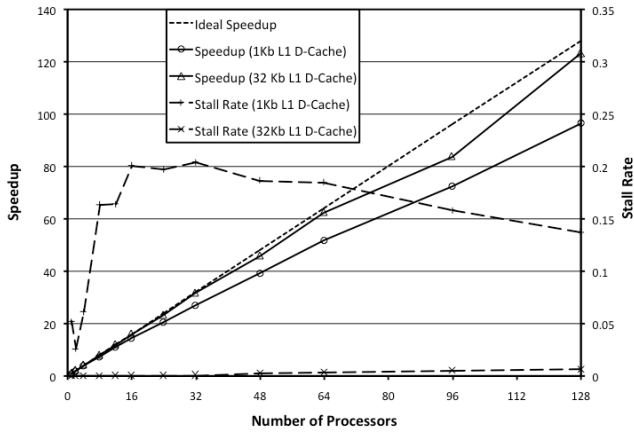


Fig. 5. Speedup and stall rate (%) against number of processors in a cluster for Livermore kernels 1 (top) and 7 (bottom). Two sets of results are presented using a 1KByte and 32KByte D-cache.

by choice. The stall rate shown in Figure 5 measures the percentage of time when the pipeline is stalled due to hazards. However, it does not include the time when the pipeline is completely empty. For high IPC, this gives us an indication of the utilisation of the pipeline and since the model uses simple in-order pipelines without multiple instruction issue and branch prediction, this metric is very important. It shows that even without those features, the architecture is able use of the pipeline efficiently by interleaving many threads in the pipeline. We do not count completely idle cycles as this can be detected and the processor powered down. For low IPC therefore, this measures energy efficiency.

### B. Families with inter-thread dependencies

This section illustrates how the model captures and exploits dependencies. The code executed here contain one or more thread-to-thread dependencies that require shared registers, which constrain the execution sequence. The test kernels are a micro-threaded version of the sine function, implemented as a Taylor expansion of 9 terms (this was implemented with a single memory bank) and the Livermore 3, inner product of 64K iterations. As with the previous experiments the same

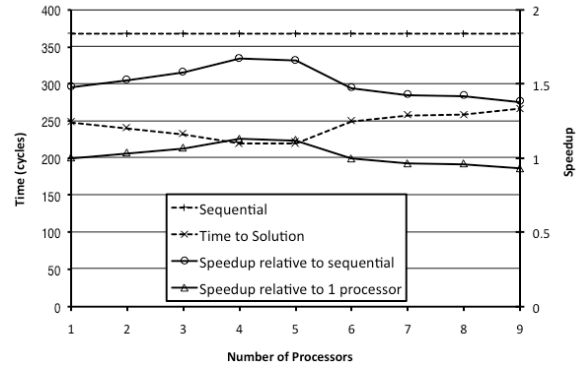


Fig. 6. Speedup of sine function.

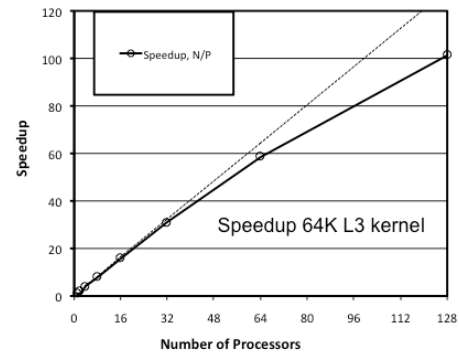


Fig. 7. Speedup of Livermore kernel 3, a reduction.

microthreaded binary code was executed on a variable sized cluster. Figure 6 shows the speedup against number of cores for the sine function. Threads are executed concurrently but dependencies between threads constrain its execution. Concurrency is still exploited locally in tolerating memory latency and do-across schedules exploit a small amount of ILP in the thread code, where speedup is proportional to the number of independent instructions (before or after a dependency). When compared to sequential code, microthreaded code is some 40% faster on a single processor and this increases to 70% faster on a cluster with 4 processors. This is due to the absence of any loop control overheads and high pipeline utilisation from the few local threads.

Loop-based code generally captures three classes of computation, data parallel where the iterations are independent, recurrence relations like the sine function and reductions. Because of the commutative and associative properties of the latter, these operations can be implemented in any order and concurrency may be exploited. Theoretically one can obtain  $O(N)$  speedup and a latency of  $O(\log_2 N)$  for binary operations. In the microthread model, several partial reductions can be performed in parallel with a final reduction giving the required result. The features of the model that allow this are the ability to create nested families of threads, the ability to specify where a family will be executed (a place) and the abstract

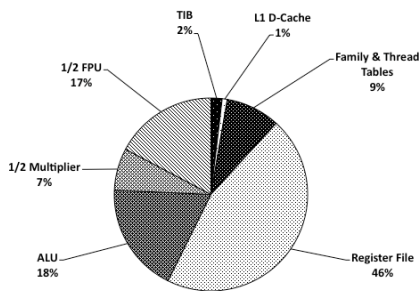


Fig. 8. The relative area of microthread core components for a core supporting 256 threads, where an FPU and multiplier are shared asynchronously between two cores.

manner in which this is all captured. For an  $O(P)$  speedup, microthreaded code can use the number of processors in a cluster, create a single thread per processor that implements a distributed reduction on the result of subordinate families that are executed locally performing  $P$  partial reductions, one per processor. When this code is executed, near linear speedup is achieved on up to 128 processors for 64K iterations. This is illustrated in Figure 7. It should be noted that this code is also schedule independent and the only dynamic information required is the number of processors in the default cluster.

### C. Core area estimation

The cost of implementing a microgrid, like most other multi-cores, scales with the number of cores. A microthread core also scales in area with the number of threads it supports, which is a design-time parameter that fixes the maximum latency the processor can tolerate. The variable sized support structures for implementing the model are the register file, the thread table and family table. All these memories have a fixed number of ports and scale proportionally.

Figure 8 shows the relative area of the various components of a single-issue microthread core, which shares an FPU and multiplier between two cores. The components are a register file (1024 entries), a family table (16 entries), a thread table (256 entries) a 4KByte D-cache and 1KByte I-cache. For SRAM components, the area was estimated using CACTI 4.0 [18] and for the functional units, the area was estimated by referring to [19]. It can be seen that the major core area contributors for a conventional processor (the L1 caches) are insignificant in a microthread core and that the support components that dominate the area are the register file, thread and family tables, which scale to give increased latency tolerance. The area of a core of this configuration in 65nm technology is approximately  $3.5mm^2$  and can be reduced to about  $2mm^2$  by reducing the latency tolerated from a few multiples of 256, to a few multiples of 64. Assuming half the core size was given over to an on-chip COMA L2 memory, then compared to the Niagara 2 chip, which is  $342mm^2$  in a 65nm process [2], approximately 50 of the largest microthread

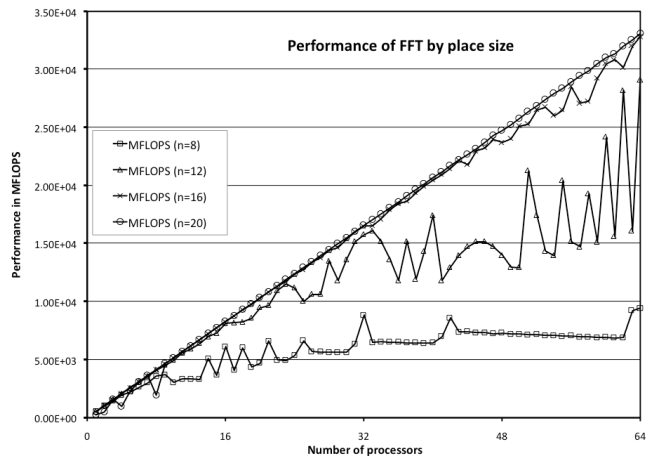


Fig. 9. Performance of FFT in MFLOPS against cluster size for FFTs of length  $2^n$ .

cores could be integrated into the same die size and support some 12K threads.

### D. FFT Performance

At this stage in our work, while waiting for the  $\mu TC$  compiler to be complete, it is difficult to provide performance comparisons for large numbers of benchmarks because each simulation currently requires assembly code to be produced by hand. However, in order for performance comparisons to be made against other published results, we have undertaken extensive simulation of the performance of a very common algorithm, namely FFT. Figure 9 shows these results. Again the same binary code was executed on a range of clusters from 1 to 64 cores. The results are presented in MFLOPS assuming a 1.5GHz core and one FPU per core. All operations are assumed to be pipelined with a latency of 3 cycles for add and mult and 8 cycles for div. Results are given for FFTs of length  $2^n$ , where  $n$  ranges from 8 to 20 by 4.

Results show a maximum performance of 0.5 GFLOPs per processor, with linear scaling out to 64 processors for transform lengths of 64K and above. The smallest transform of 256 points gives a maximum performance of between 5 and 10 GFLOPs and saturates at using 12 processors, or about 10 threads per processor.

## VI. RELATED WORK

The UltraSPARC T1 and T2 processors target Internet, database and network servers and are also referred to as Niagara [16] and Niagara II [2]. The latter has 8 cores supports only 64 threads in hardware. Each core has its own L1 cache, FPU and two ALUs and all cores share a 4MB L2 cache. The major difference between this and a microgrid is twofold, one is parametric and one fundamental. As described above, a DRISC core can support 100s of threads and 10s of thousands of threads. In programming, Niagara utilises speculative threading, whereas DRISC manages user or compiler generated concurrency explicitly, with conservative (dataflow) execution rather than speculative execution.

Although the Niagara may appear to be the closest work to that described here, WaveScalar from the University of Washington [7] with its dataflow execution model, is conceptually closer. Both models capture a programs dependencies and the difference is in the execution model. WaveScalar uses the dataflow firing rule, whereas DRISC uses RISC instruction execution with blocking on register reads. Both models contextualise their synchronising memory, e.g. via waves in the WaveScalar or in DRISC as concurrent thread contexts in an extended register file. In WaveScalar, all synchronisations are producer-consumer but in DRISC, there is a bulk synchronisation on the termination of a family that extends synchronisation to the models shared memory. Both models define locality statically and create and distribute concurrency dynamically, which are necessary features for efficient compilation and execution. Both models also tolerate high levels of latency and can exploit the latency in a distributed cache-coherent shared memory. This property can greatly reduce the pressure on the processor-memory interface. The conceptual similarities were strengthened recently. In prior work on WaveScalar, the sequential model of programming was managed by providing an ordering on loads and stores but this proved to be too sequential and the model has now been extended with the concept of threads to manage memory concurrency [7].

The other work that is conceptually related to microthreading is the Data-Driven Multithreading Chip Multiprocessor (DDM-CMP) [17] proposed by the University of Cyprus. This is a coarse-grain dataflow model based on managing blocking threads from the cache interface of a regular processor. Each core in this design has its own L1 cache incorporating a Thread Synchronisation Unit (TSU), this implements synchronisation on the blocking threads before executing those threads as a conventional code segment. A system network connects the cores and a TSU network connects TSUs. The TSU stores a dataflow graph in its Graph Memory and a number of thread queues, which maintain the status of threads and code blocks. The scheduling of complete threads is done at runtime using the dataflow firing rule and a thread is never issued before all of its operands are ready, when it can execute to completion because of the blocking semantics.

## VII. CONCLUSION AND FURTHER WORK

This paper has introduced the microthread-programming model, which manages families of concurrent non-blocking threads. An implementation of that model has been undertaken as a software emulator, which is cycle accurate and this has been used to show speedup, intolerance to latency and hence size of L1 cache, and efficiency of the pipeline in terms of both stalls and conservative instruction execution. Speedup on independent families of threads as well as reductions is near linear to 128 cores. Moreover, microthread code is more efficient than the corresponding sequential code for the same processor. To date the results are limited by the lack of a core compiler for TC, however this compiler is already generating code and soon these results will be extended with more widely used benchmarks. These emulated results will

eventually be used in a high-level simulation of the chip architecture from which decisions will be made in terms of this flexible design, e.g. the number of processing cores, FPUs and L2 cache blocks required in different clusters. Note that this will be a heterogeneous design using high-level homogeneous components, where the processor is the transistor and the cluster the processor!

## ACKNOWLEDGMENT

The authors acknowledge support from the European Community in funding the research undertaken in the ÆTHER project. Support was also provided through the Netherland Research Organisation, NWO, through their funding of the Microgrids project.

## REFERENCES

- [1] Panesar G, Towner D, Duller, A, Gray, A and Robbins, W (2006) Deterministic parallel processing, *International Journal of Parallel Programming*, 34, 4, pp 323-341, ISSN:0885-7458.
- [2] McGhan, H (2006) Niagara 2 opens the floodgates, *Microprocessor Report*, Nov. 2006, pp 1-9.
- [3] Semiconductor Industry Association, (2006) *International technology roadmap for semiconductors 2006 update*, Technical Report, 2006.
- [4] Borkar, S (2005) Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation, *IEEE Micro*, 25, no.6, pp. 10-16.
- [5] Herlihy, M P and Moss J E B (1993) Transactional memory: architectural support for lock-free data structures, in *Proc. 20th International Symposium on Computer Architecture (ISCA 1993)*, pp. 289-300.
- [6] Papadopoulos, G M (1991) *Implementation of a General-Purpose Dataflow Multiprocessor*, MIT Press (Research Monographs in Parallel and Distributed Computing), ISBN-10: 0262660695.
- [7] Swanson S, Schwerin A, Mercaldi M, Petersen A, Putnam A, Michelson K, Oskin M and Eggers S J (2007) The WaveScalar architecture, *ACM Transactions on Computer Systems (TOCS)*, 25, issue 2, no. 4.
- [8] Bolychevsky, A, Jesshope C R and Muchnick, V B (1996) Dynamic scheduling in RISC architectures, *IEE Trans. E, Computers and Digital Techniques*, 143, pp 309-317.
- [9] Jesshope C R (2007) A model for the design and programming of multicores, to be published in: *Advances in Parallel Computing*, IOS Press, Amsterdam.
- [10] K.S.R. Corporation(1992) *KSRI technical summary*. Technical report.
- [11] Hagersten E, Landin A and Haridi S (1992) DDM - a cache-only memory architecture, *IEEE Computer*, 25(9), pp. 445-4.
- [12] Zhang L, Jesshope C R (2007) On-chip COMA Cache-coherence protocol for Microgrids of Microthreaded Cores, *Proc EuroPar 2007 Workshops*. Volume 4854 LNCS, Springer, pp. 38-48.
- [13] Jesshope C (2006)  $\mu$ TC - an intermediate language for programming chip multiprocessors, *Proc. Asia Pacific Computer Systems Architecture Conference - ACSAC06*, ISBN 3-540-4005, LNCS 4186, pp147-160.
- [14] Gao G R, Sarkar V (2000) Location consistency-a new memory model and cache consistency protocol, *IEEE Transactions on Computers*, 49(8), pp. 798813.
- [15] Bernard T A M et. al. (2007) Strategies for Compiling  $\mu$ TC to Novel Chip Multiprocessors, *International Symposium on Systems, Architectures, Modeling and Simulation - SAMOS 2007*, S. Vassiliadis et al. (Eds.), LNCS 4599, pp. 127-138.
- [16] Poonacha Kongetira, Kathirgamar Aingaran, Kunle Olukotun (2005) Niagara: A 32-Way Multithreaded Sparc Processor, *IEEE Micro*, 25(2), pp. 21-29.
- [17] Trancoso P, Evripidou P, Stavrou K, and Kyriacou C (2006) A case for chip multiprocessors based on the data-driven multithreading model. *Int. J. Parallel Program*, 34, 3 (Jun. 2006), pp. 213-235.
- [18] Tarjan D, Thoziyoor S, and Jouppi N P (2006) *Cacti 4.0*, Western Research Laboratory, Compaq, Tech. Rep., 2006.
- [19] Gupta S, Keckler S W, and Burger D, (2000) *Technology independent area and delay estimates for microprocessor building blocks*, Computer Architecture and Technology Laboratory, the University of Texas at Austin, Tech. Rep.