Microthreading and μTC for Massive On-Chip Concurrency

C.R. Jesshope, M.A. Hicks, M. Lankamp, R.C. Poss, T. Bernard, A. Pimintel, C. Grelck
M.W. van Tol, S. Polstra, L. Zhang, K.S. Bousias

Computer Systems Architecture Group,Informatics Institute, University of Amsterdam, NETHERLANDS

The Computer Systems Architecture research group is tackling significant problems in the scalability of massive on-chip concurrency. The efficacy of the massively-parallel Microgrid architecture, μTC language and associated toolchain are demonstrated here, showing important progress in this vibrant research area. – *SIREN 5th November 2009*

1) Project Motivation and Goals

Trends in microprocessor design – feature size continues to increase in line with Moore's Law, allowing increasing gate density on chips. However, locality, power density and reliability are becoming limiting factors.**Processors will become distributed and multicore!**
The **Goals** are similar in nature to all distributed systems:

- Scalable and distributed architecture
- Program multiple cores from (existing) sequential code
- Dynamic resource and concurrency management
- Support for legacy code

Our **Solution** is a scale invariant programming model from the ISA level upwards.

2a) What is a Microthread?

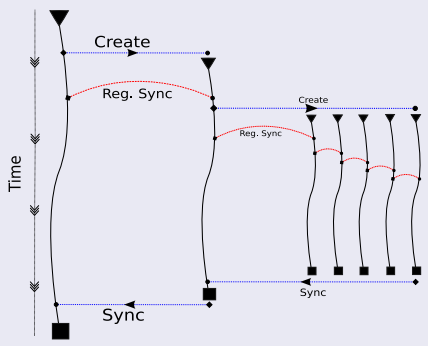
- A sequence of instructions defined to form a *lightweight* thread
- Has it's own *microcontext of registers*
- Created in *named, parameterised* families of many threads
- Smallest unit of execution in the Microthreading model

2b) The Instruction Set, Architecture and Programming Model

Microthreaded Instruction set implements blocking threads based on parameterised families. These can capture all kinds of loops, function calls and asynchronous task creation. Only the following instructions need to be added to an existing ISA:

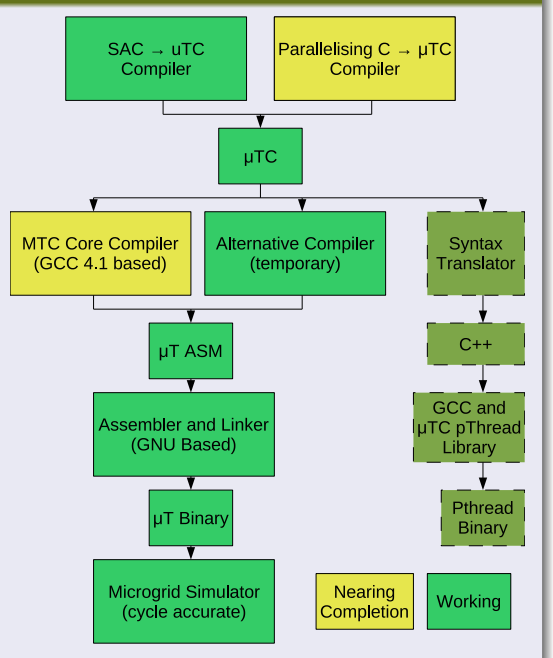
- **create** – create a named, parameterised family of blocking threads
- **sync** – associated barrier synchronisation on named family
- **break** – stop the execution of all other threads within current family
- **kill** – stop the execution of all threads in a named family

Synchronised communication between adjacent threads in a family is achieved through scalar *i-structures* in a large, distributed register file.

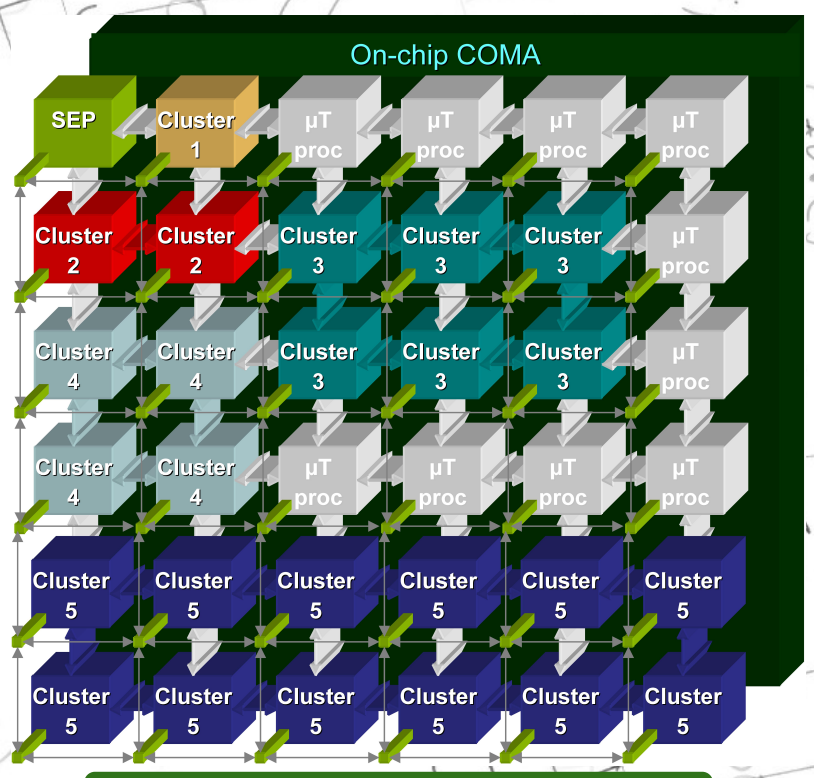


- **Concurrent Composition** – programs are constructed concurrently
- **Black line segments represent threads**
- **Branches** in threads represent the **creation** of subordinate thread families (parent continues)
- **Synchronisation** over a subordinate family
- **Dependencies** between threads in the same family (red lines in diagram)

*) The μTC Toolchain

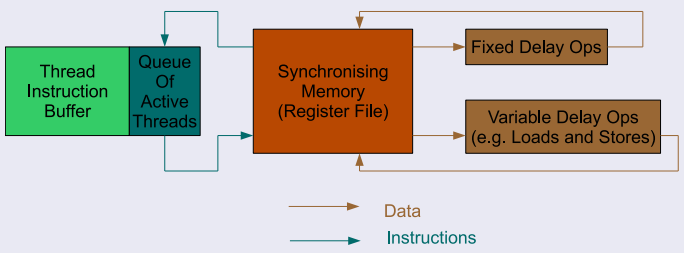


- **Full cycle-accurate simulation of the Microgrid with COMA**
- Massive Progress in toolchain integration over last 12 months



A Microgrid of Microthreaded Cores (3b)

3a) Individual Microthreaded Core Pipeline



1. Threads are created dynamically with a context in synchronising memory
2. Instructions issued from the head of the active queue read synchronising memory
3. If data is available then instruction is executed, otherwise thread suspends on empty register
4. When data is written, suspended threads are rescheduled to execute

The **Synchronising Memory** is a large register file that is used to control program execution. The processor **tolerates large latencies** by cheap switching between lightweight thread contexts (from the active queue).

3b) The Microgrid Architecture (see central figure)

A Microgrid consists of **configured rings** of Microthreaded cores (see 3a) and uses a diffuse **COMA** memory system (allowing flexible partitioning of on-chip memory). An SEP core allocates rings of cores to threads which can then delegate work.
Dependency rings are circuit switched, Delegation grid is packet switched
A create instruction distributes a family of threads to a ring of processors.

4) The Microthreaded C (μTC) System Level Language

- Captures ISA Extensions of μT Architecture in C language
- Intended as a compiler target language for C, C++ etc.
- **thread, family, place (resources)** are captured as new C types
- **shared** type modifier shares a variable between threads, as an i-structure

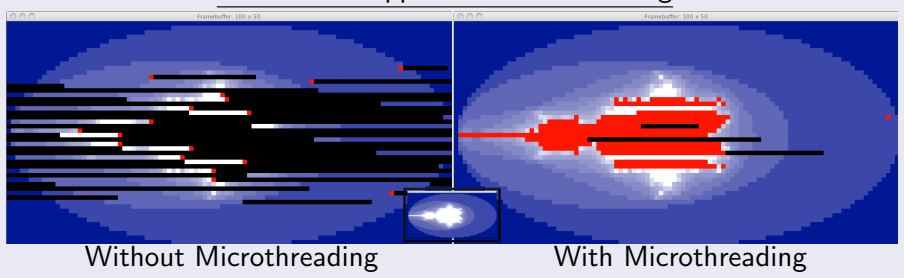
```
/* C Sum of Squares */
float a[100], b[100], sum=0;

for(int i=0; i<100;i++) {

    sum+=a[i]*a[i] + b[i]*b[i];

}
```

```
/* µTC Sum of Squares */
float a[100], b[100], sum=0;
thread void sqsum(shared float s) {
    index i;
    s+= a[i]*a[i] + b[i]*b[i];
}
family fid;
place pid;
shared float sum; /* will contain result */
create (fid, pid, 0, 99, sqsum, sum);
sync(fid);
```

Example of a loop in standard C transformed into μTC, with dependency captured

5) Experimental Results

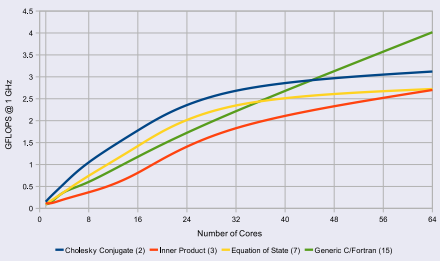
Mandlebrot Approximation Rendering



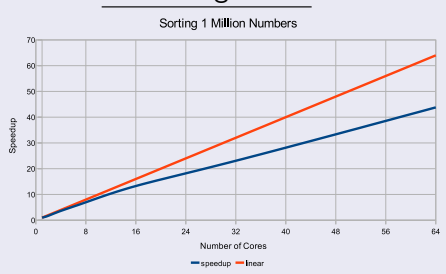
Without Microthreading (1 thread per core) | With Microthreading (256 threads per core)

⇒Rendering 15,000 points on 32 cores, shown **after 35,000 cycles**.
⇒Microthreaded version is nearly **4**× **faster**, due to latency handling

Livermore Kernels



NAS Integer Sort



⇒Even for large numbers of cores, near theoretical-maximum-performance is achieved!

References:

K. Bousias, L. Guang, C.R. Jesshope, M. Lankamp (2008) Implementation and Evaluation of a Microthreaded Architecture, Journal of Systems Architecture