

μ TC – an intermediate language for programming chip multiprocessors

Chris Jesshope

Institute for Informatics, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, Netherlands, Jesshope@science.uva.nl

Abstract. μ TC is a language that has been designed for programming chip multiprocessors. Indeed, to be more specific, it has been developed to program chip multiprocessors based on arrays of microthreaded microprocessors as these processors directly implement the concepts introduced in the language. However, it is more general than that and is being used in other projects as an interface defining dynamic concurrency. Ideally, a program written in μ TC is a dynamic, concurrent control structure over small sequences of code, which in the limit could be a few instructions each. μ TC is being used as an intermediate language to capture concurrency from data-parallel languages such as single-assignment C, parallelising compilers for sequential languages such as C and concurrent composition languages, such as Snet. μ TC's advantage over other approaches is that it allows an abstract representation of maximal concurrency in a schedule-independent form. Both Snet and μ TC are being used in a European project called AETHER, in order to support all aspects of self-adaptive computation.

Keywords: Self-adaptive computing, concurrent languages, data-driven computation, programming chip multiprocessors.

1. Introduction

This paper describes language work originating in the *MicroGrid* project at the University of Amsterdam, which is designing chip multiprocessors based on the microthreaded model of concurrency [1]. It is also being adapted as a virtual system's architecture (SVM) for highly concurrent, self-adaptive systems in the European *AETHER* project. In the former, it represents a transparent view over the underlying hardware support for concurrency and in the latter it presents a pragmatic attempt to define the functionality of a virtual machine for system-level interfaces between self-adaptive network entities (SANEs). SANEs are the concurrent components that are dynamically manipulated to achieve the project's goals of self-adaptive computing.

The language defined in this paper provides the functional definition and concurrency of the virtual machine describing SANE components. Most of the scheduling and resource aspects of SVM are outside of the scope of this paper. However, as μ TC directly captures the hardware implementation of microthreaded microprocessors it has an abstract view of resource issues in this context. More details including simulations chip-multiprocessors based on this model can be found in [2].

μ TC is a rather profound but simple extension to the C language, allowing it to capture thread-based concurrency. μ TC is capable of expressing static, heterogeneous concurrency and dynamic, homogeneous concurrency. It is similar in some aspects to OpenMP [3], however there are significant differences. One of the most significant differences is that the language assumes a synchronizing memory. This captures dependencies between threads allowing sequence to be transformed into concurrency, where any dependencies are managed transparently in a data-driven manner. The other major difference is that C is extended with executable constructs rather than being annotated with pragmas as in OpenMP. For example, families of threads are created as named entities within the language and can be referenced, for example in a control component for a given SANE. This difference is fundamental and provides the mechanism for dynamically manipulating SANEs (as families of threads) as required in self-adaptive systems. For example, an identified family of threads can be terminated, either with prejudice or in a controlled and orderly manner that allows pending synchronizations to complete, so that a concurrent program can be moved to new resources or have its behaviour modified.

μ TC is based on the concept of microthreading [1], which includes synchronising memory and efficient, low-level scheduling. Our prior results [2] on microthreading show that very efficient implementations of these concepts are possible. In general, there is a range of scheduling options for μ TC. On a microthreaded microprocessor, μ TC programs are scheduled dynamically and the constructs introduced simply reflect instructions in the ISA. On conventional processors, some kind of static schedule will need to be generated by the μ TC compiler or its run-time system, to remove the requirement for synchronising memory. For example, a sequential implementation of μ TC exists, which executes threads in creation order with no interleaving. The philosophy adopted, captures maximal application concurrency and reflects the asynchrony and locality of communication that is found in chip multi-processors. The assumption is that the transformation from concurrent to sequential is trivial in principle, although difficult to define in the case of non-determinism in timing. However, the real goal is to execute μ TC directly, which is indeed possible. Ideally, a program written in μ TC is a dynamic concurrent structure over small sequences of code, which, in the limit, could be just a few instructions each.

2. Motivation and background

Reference [4] provides a compelling argument for the elimination of non-determinacy in programming concurrent systems and claims that we are on the threshold of a potential disaster as multi-threaded code is migrated to chip multi-processors with non-deterministic scheduling. The same argument is made in [5], where similar issues are raised about programming state-of-the-art multiprocessor systems. These include the following real or perceived problems:

- the user has to parallelise existing serial code;
- explicitly threaded programs using a thread library are not portable;
- writing efficient multi-threaded programs requires intimate knowledge of the machine's architecture and micro-architecture.

Here the following solutions are adopted to these problems. Users do not normally parallelise applications but generate μ TC from deterministic code, such as plain old C or Single-assignment C (SAC) [6] (a functional, data-parallel language). Moreover, concurrency in μ TC is achieved in an abstract way that does not require reference to a thread-library, thus only the μ TC compiler will need to have knowledge of the architecture or micro-architecture and any run-time support for a given target.

The main tools in the AETHER project will be compilers for conventional languages and for the configuration language Snet [7] all of which will target μ TC, as well as various implementations of μ TC to specific targets. Our own interest is the compilation of μ TC to microthreaded binaries and their implementation in reconfigurable processor arrays to provide dynamic management of resources, e.g. see [8]. For this, gcc will be modified to compile μ TC to schedule-invariant, microthreaded binary code. As the tool chain above is being developed, μ TC will be used as a user programming language. The first implementation of μ TC is a translation to C, using a rather trivial schedule that executes all threads in index sequence. Subsequent work will focus on the automatic parallelisation of C programs targeted to μ TC, which together with the μ TC compiler, will allow the execution on microthreaded binaries from legacy, sequential C code on our Microgrid simulator.

The remainder of this paper introduces the language and provides numerous examples to illustrate the semantics of the constructs and how they would be used in a number of different application scenarios.

3. Additions to C

Only a small number of constructs are added to C, along with the semantics of the synchronising memory, which is described in detail below. The constructs map onto low-level operations that provide the concurrency controls in a microthreaded ISA, see [1] and allow concurrent programs to be dynamically instanced and preempted, either gracefully or with a prejudice. Family identifiers provide the control over the concurrent sections. No other language to our knowledge provides such support for families of threads and this enables many of the dynamic aspects of SANEs.

μ TC adds the following keywords to standard C. They can be used anywhere in a C program, subject to restrictions described in each keyword's description. They are:

create	Control construct used to create a family of microthreads;
thread	Type specifier to indicate the functions that define the microthreads;
shared	Type qualifier of variables shared between microthreads;
index	Type qualifier of the index variable of a family of microthreads;
sync	Construct that waits for the termination of a specified family;
break	Construct that terminates a family from one of its threads;
kill	Construct that terminates a specified family externally;
squeeze	Construct that preempts the execution of a specified family so that it may be restarted without loss of state.

Before these constructs are defined in detail, a brief definition of the memory model used in implementing them must be given. It is assumed that there are two kinds of memory (analogous to registers and main memory in the sequential machine model).

They are a synchronising memory and a non-synchronising memory. The latter is shared main memory with no assumptions about access time. All inter-thread communication is performed in synchronising memory, which is assumed to be fast, on-chip and close to the processor. There are also restrictions on inter-thread communication that reflect the asynchrony and locality of on-chip communications. Synchronising memory is allocated dynamically to a thread on its creation and is released when that thread completes (or is forced to complete). It implements dataflow synchronisation and threads block on reading it. Thus if a thread attempts to read an undefined location in synchronising memory, it will not proceed beyond the statement that attempted to read the undefined variable.

Non-synchronising memory has relaxed consistency and is bulk synchronous with respect to a family of microthreads. This means that during the execution of a family of threads, threads may read from or write to a structure in synchronising memory but the state of the writes is not consistently defined until the whole family completes (or is forced to complete). This means that two concurrent threads in the same family cannot usefully share data via non-synchronising memory.

3.1 create

```
create(fid; start; limit; step; block) <named thread>|  
    <compound statement>;
```

The `create` construct defines a concurrent section as a family of microthreads over an index variable. Threads can be defined either by a compound statement or a named thread, which is similar to a function (see Section 3.2). `create` returns a unique family identifier, *fid*, to identify and control the family created and may be used anywhere within a C program, including from within another thread. `create` has the following components:

- *fid*: a variable from the creating context that receives the family identifier, that uniquely identifies the created family and can then be used to synchronise or terminate the family.
- *start*: an expression defining the start of the index sequence for the family of microthreads; it is evaluated when the create is executed (default: 0).
- *limit*: an expression defining the limit of the index sequence for the family of microthreads; it is evaluated when the create is executed (default: unlimited).
- *step*: an expression defining the step value between indices; it is evaluated when the create is executed (default: 1).
- *block*: an expression defining the maximum number of index values allocated per processor in a single allocation round; the expression is evaluated when the create is executed (default: maximum possible).

The triple (*start*, *limit*, *step*) defines an index sequence over the threads created and a unique value from this sequence is available to each thread. Any of these expressions may be omitted and an appropriate default value will be assumed. A blank *limit* statement causes an infinite number of threads to be created and in this case, thread creation will have to be terminated by a `break`, `squeeze` or `kill`. A blank

block expression means that an implementation will allocate as many threads as there are resources available to do so.

The `create` construct creates threads in block-index order and dynamically allocates synchronising memory to each thread it creates (the memory is released when the thread completes). Variables in synchronising memory are initialised empty (i.e. they block a thread that attempts to read them). The exception is the thread index, which is initialised to the index value for the thread as defined by the triple above.

An arbitrary number of processors, p say, which can be defined at create time, may be used to execute the created threads and the implementation will distribute the threads over those processors in block-index order. Block-index order is where the first *block* index values are allocated to the first processor and so on, to each processor involved, so that $p \cdot \textit{block}$ indices are created in a round of allocation over the p processors. This allows infinitely many threads to be defined and managed and is similar to k-bounded loops used in dataflow, i.e. it provides an artificial dependency limiting the use of resources and providing management over resource deadlock.

Two examples that create exactly the same family of threads are given in Table 1, together with the equivalent sequential code for reference.

Table 1. Creating families of threads with compound statements and named threads.

<i>Compound statement</i>	<i>Named thread</i>	<i>Sequential equivalent</i>
<pre>int a[10]; int fid, s=0; ... create(fid; 0; 9){ index int i; shared int s; s = s + a[i]; } sync(fid); ...s...</pre>	<pre>thread sint(shared int sum, int array[]){ index int idx; sum = sum + array[idx]; } int a[10]; int fid, s = 0; ... create(fid; 0; n-1) sint(s, a[]); sync(fid); ... s ...</pre>	<pre>int a[10]; int fid, s=0; ... for(i=0; i<10; i++){ s = s + a[i]; } ...s...</pre>

Each thread in this homogeneous family contains its own copy of the shared variable s/\textit{sum} , which defines a dependency chain through the family of threads, with each thread reading its neighbour's shared variable. For more detail on this see also the definition of `shared` in section 3.7.

A heterogeneous `create` makes use of a list of named threads. In the heterogeneous case, the compiler must know the index range statically. This form can be used to represent ILP in basic blocks or to manage MIMD concurrency at the application level. Again there can be `shared` variables that define dependency chains between the threads and these are declared in the argument list of the threads, must be common to all threads and bound to variables in the creating thread. An example is given in Table 2 below, along with the equivalent sequential code.

Table 2. Creating a heterogeneous family of threads.

<i>Thread list</i>	<i>Sequential equivalent</i>
<pre>thread mt1(shared real sr){ sr=b*b-sr;} thread mt2(shared real sr){ r1=(sr-b)/2*a; r2=-(sr+b)/2*a; } ... real a, b, c, sr, r1, r2; int fid; create(fid;1;3) mt1(sr), sqrt(sr), mt2(sr); sr=4*a*c sync(fid) /*r1, r2 now valid*/</pre>	<pre>... real a, b, c, sr, r1, r2; sr=sqrt(b*b-4*a*c); r1=(sr-b)/2*a; r2=-(sr+b)/2*a;</pre>

In this example, a single shared variable, `sr`, defines a dependency chain between the threads. Note that the built-in thread `sqrt` gets its parameter and passes its result via this shared variable. When created, each user-defined thread can proceed with some computation. Thread `mt1` can compute b^2 and `mt1` can compute $2a$, while the main thread computes $4ac$. Then the computation is constrained by the shared variable `sr`, which is passed from main to `mt1`, `mt1` to `sqrt` and `sqrt` to `mt2`. The result is written in two global variables in non-synchronising memory, which are defined only when the threads have been synchronised. The code in Table 2 is an example of explicitly programmed ILP.

3.2 thread

```
thread <name> (<argument list>){...}
```

The `thread` construct defines a C function as a thread in μ TC. It can be used with `create` to generate instances of the function as dynamic threads and to match an argument list in the definition with a set of parameters from the creating environment. There are a number of differences between a function and a thread. Firstly, there is no return type (or it is assumed to be void), as threads do not return values other than by shared variables or writes to non-synchronising memory. A `break`, see Section 3.4, can also return a value to the creating thread that via the `sync` construct.

Threads cannot contain calls to functions but can create further subordinate threads, which are concurrent function calls, where the thread is triggered by writing values to its arguments and the creating environment waits on results using `sync`. Results are either defined by the local variable used to initialise a shared-variable dependency chain or can be written to non-synchronising memory, which is shared (both are used in the example in Table 2). There is no reason why C programs cannot be completely translated to threaded programs using threads instead of functions.

3.3 sync

```
sync(fid; return);
```

The `sync` construct is used to detect the termination of a concurrent section defined by a family of threads with identifier *fid*. It also returns a value to *return* in the definition above, which defaults to *maxint* if the family terminates normally. The construct blocks until the family specified by *fid* has completed and then completes its execution by setting the return value. `sync` returns a value that is set by a `break` construct, if one was executed, otherwise the return value defaults to *maxint*. A `create` and corresponding `sync` define a concurrent section, which includes both the creating thread as well as the family of created threads. Global memory written in a concurrent section cannot be reliably read by other threads in the same concurrent section, nor by another family, until the family writing global memory has been synchronised using the `sync` construct. Only one `sync` may be issued on a given family of threads and a `sync` in two concurrent threads on the same family may have unpredictable results.

3.4 break

```
break(result);
```

The `break` construct terminates a family of threads from within one of its threads. It stops any remaining thread creation and releases all synchronising memory, losing any synchronising state that the family may have had. It also allows the breaking thread to return a value to the creating environment by its parameter *result*.

An important issue in the implementation of `break/sync` is the guarantee that an outstanding synchronisation on a location in synchronising memory will not interfere with any subsequent use of that location, i.e. if it is subsequently allocated to another family of threads. For example, assume that a load from non-synchronising memory had been issued in a thread and a `break` released the target location in synchronising memory before the load was satisfied. The implementation of `break/sync` must ensure that any subsequent response from memory for that family will no longer update synchronising memory.

3.5 squeeze

```
squeeze(fid; return);
```

The `squeeze` construct is similar in operation to `sync` but is executed concurrently with the creating environment and is used to bring a family of threads identified by *fid* to a well-defined termination state. It stops any remaining thread creation and waits for any outstanding synchronisations to complete before returning the index value of the first thread not created to its *return* parameter. Like `sync` it blocks until the family of threads has terminated. The *return* is the concurrent program's equivalent of a program counter in a sequential program when pre-empting the program and

enables the family to be restarted (perhaps on different resources) without loss of data. Note that termination of a thread requires the termination of any synchronised, subordinate threads within it and these subordinate families are not automatically squeezed. If a deep squeeze is required, it must be programmed, as it requires the building of a data structure of index values for all subordinate squeezed families. In practice this construct could be executed in any thread that has access to a family's *fid* and it is required to dynamically migrate SANE components.

Only one squeeze may be issued on a given family of threads and squeezing in two concurrent threads on the same family may have unpredictable results. An example of the use of squeeze is given below:

```
int fid1, fid2, resume;
...
create(fid1;1;1){ /*job wrapper*/
    shared int fid2;
    create(fid2)job(); /*job to be squeezed*/
}
sync(fid1)
...
squeeze(fid2, resume)
```

In this example the job wrapper, family *fid1*, creates an infinite family of threads defined by a thread named *job* and returns the family identifier, *fid2*, back to the main thread, leaving the family detached, as *fid2* is never synchronised. This example shows how to obtain and use *fid2* to asynchronously terminate the family. Note that *fid2* becomes defined on the sync on family *fid1*. This code skeleton is an example of a SANE component having a control part and a functional part running side by side.

3.6 kill

```
Kill(fid);
```

The `kill` construct is similar in operation to `squeeze` and is also executed from a concurrent control thread but it is used to bring a concurrent section defined by *fid* to a forced termination, by stopping any thread creation and forcibly terminating any executing threads, i.e. all pending synchronisations are lost! The other difference is that `kill` does operate recursively, i.e. it kills not only the family of threads that is identified but also any subordinate families that have been created.

3.7 index/shared

```
index int i; shared real s;
```

The `index` and `shared` keywords are type modifiers used in μ TC; `index` defines the thread sequence number and is set automatically by `create` and

`shared` defines any variables in synchronising memory that are shared between threads in a family.

4. Memory model

4.1 Synchronising memory

The most important aspect of μ TC code is the concept of synchronising memory. Each thread has a context of local, scalar variables dynamically allocated to it in synchronising memory, which are initialised to the *empty* state and which are garbage collected when the thread completes. These variables provide synchronisation with data from non-synchronising memory and also with other threads if the variables are declared as `shared`. Reading an empty variable in synchronising memory will block the thread reading it until the value has been set (it gets suspended and can no longer proceed until the data is available). Synchronising memory is dynamic and data created by threads must either be `shared` or written to non-synchronising memory before the thread terminates or it will be lost.

To communicate between threads in the same family, synchronising memory must be used and must be declared as `shared`. Sharing is deliberately restricted to reflect the locality of communication found in silicon systems. Each thread has one *neighbour* that can read its shared local values, which is defined as the next thread in index sequence in the `create` for that family. To initialise this chain of neighbours, the first thread reads a variable of the same name from the creating environment (not declared as `shared`) or in the case of a named thread; a binding is made to a variable in the creating environment. Following the termination of the family, a read to the variable in the creating environment will yield the value written to the shared variable from the last thread created. Dependency chains through a family of threads are therefore initialised and closed using variables from the creating thread.

The following example illustrates a potential problem with `shared` variables:

```
int *a, n, s = 0;
...
create(fid; 0; n-1){
    index int i; shared int s;
    s = s + 1;
    s = s * 2;
    a[i] = s;
}
```

Here, the shared variable `s` is written twice in each thread. The value obtained by a read from a neighbour is therefore non-deterministic. Dataflow synchronisation ensures `s` can not read until it is written but when written it can be read before or after the second write. The solution used in the μ TC compiler is to enforce single assignment semantics for shared variables, introducing further local variables as required. A family will then give exactly the same results as if each thread were

executed sequentially. The compiler must ensure that the first read of *s* is from the prior thread and that only the last write will synchronise with the following thread. All other uses of *s* must be local. The μ TC compiler would therefore generate the equivalent of the following code in this example:

```
int *a, n, s = 0;
...
create(fid 0; n-1){
    index int i; shared int s; int t;
    t = s + 1;
    s = t * 2;
    a[i] = s;
}
```

An example using non-local shared variables is given below. It implements a recurrence relation with dependencies from the neighbour and neighbour's neighbour. It computes Fibonacci numbers:

```
int i, fid, temp1, temp2, Fibonacci[10];
temp1=fibonacci[0]=0;
temp2=fibonacci[1]=1;
create(fid; 2; 9){
    index int i; shared int temp1, temp2;
    fibonacci[i] = temp1 + temp2;
    temp1 = temp2;
    temp2 = fibonacci[i];
}
sync(fid);
```

More generally, a shared variable may pass data to an arbitrary thread in a family using deterministic choice within the thread index (this is data-routing).

4.2 Non-synchronising memory

Non-synchronising memory has relaxed consistency during the execution of a family of threads and writes to this memory are only well defined only after the family of threads has completed (defined by the `sync` construct). Using non-synchronising memory, a thread may write to any declared variable that is in scope (normal C rules) or has been passed to it as a parameter. It is a requirement for deterministic execution, that each thread in a homogeneous family must write to a unique element of a data structure, which is selected by its index value, e.g. `x[i]`, where *i* is the family's thread index. The range of *i* is defined by `create`. In heterogeneous families uniqueness must be guaranteed by the threads' code and can be to non-indexed variables. In either case, a read after a write to variables updated in a thread family cannot be safely be performed until a `sync` has been executed on the family of threads that performed the write.

When assigning to indexed variables, care must be taken with expressions other than the local index value, as reads and writes to the same element of an indexed

structure can only be guaranteed to be consistent within the same thread or following the `sync`. An example is where different elements of an indexed data structure are required in a thread. Consider the following poorly defined μ TC code fragment:

```
int a[10], fid, n=10;
create(fid; 0; n-2){
    index int i;
    a[i] = a[i] + a[i+1];
}
```

This program does not give deterministic results as `a[i+1]` could be read by a thread either before or after its neighbour had updated `a`. A deterministic program i.e. one that guarantees the result expected from a sequential schedule requires the following transformation:

```
int a[10], shift_a[10], fid, n=10;
create(fid; 0; n-2){
    index int i;
    shift_a[i] = a[i+1];
}
sync(fid);
create(fid;0; n-2){
    index int i;
    a[i] = a[i] + shift_a[i];
}
```

In C, dependency chains may be defined through iterations spaces by indexed data structures, which if translated naively could also give non-determinism. For example:

```
int *sum, *a, fid, n = 10;
sum[0] = a[0];
create(fid; 1; n-1){
    index int i;
    sum[i] = a[i] + sum[i-1];
}
```

Here, `sum` is a global array in non-synchronising memory indexed in each thread (it is not a local shared variable). Although the μ TC compiler could allocate shared variables to implement this dependency chain, it would require the compiler to check that the index expression defined neighbours in the family of threads. Although this is trivial in the example above, it may not always be the case and run-time checks may be unavoidable in some code. To avoid this, shared variables in μ TC must always be declared explicitly and the above example should be written as:

```
int *sum, *a, fid, n = 10, s;
sum[0] = s = a[0];
create(fid; 1; n-1){
    index int i; shared int s;
    sum[i]= s = a[i] + s;
}
```

A more complex example is given below, which uses both thread index and global index expressions. It performs matrix-vector multiplication and is defined as a thread.

```
thread matvec(int *a, *x, *y, n){
    int fido;
    create(fido; 0; n)
    {
        index int i;
        int fidi, s = 0;
        create(fidi; 0; n; 1; 4){
            index int j;
            shared int s;
            s = s + a[i][j]*x[j]
        }
        sync(fidi);
        y[i] = s;
    }
    sync(fido);
}
```

This thread creates n^2 threads, where the n outer threads are independent and the n inner threads contain a dependency chain on s . In the inner family, the code uses both thread-index selection, using j , as well as global index selection, using i .

5. Resource management

The use of the `create`'s *block* parameter provides for management of resources on thread creation. There are two issues here, the placement of code on specific resources and the management of deadlock. The latter is illustrated in the example above. The *block* parameter in the outer `create` says that no more than 4 threads should be allocated to a processor at any time. This allows resources to be allocated to the inner threads. If the *block* parameter had not been used and n was such that the outer loop exceeded the resources available on one processor, then no inner family threads could have been created and no outer thread could have completed, hence deadlock!

In general it is possible to create families of threads that exceed the resources available for their creation and the use of *block* allows those resources to be spread through a chain of creates to avoid or resource deadlock or to minimise inefficiency in virtualising resources in the hardware.

The *block* parameter can also be used to `create` a thread in a particular processor. A modification of the code in Section 3.5 can be made that creates the detached job on a specific processor. For example on p processors the following code would load the detached job onto the j th processor.

```
int fid1, fid2, j;
...
create(fid1;1;p;1;1){ /*loader - p processors*/
    index int i; shared int fid2;
    if (i = j)create(fid2)job();
}
sync(fid1) /*family fid2 loaded on processor j*/
}
```

5. Cost models

μ TC will require a different cost model for each target and that model must be embedded in the compiler for the target. However, no attempt should be made to schedule threads in the μ TC language, as this is counter to its philosophy. If the cost model dictates, compilers for a given target will create schedules for execution, either statically, as in the case of translating μ TC to C, or dynamically as in the case of a microthreaded pipeline. It is important therefore not to carry over a cost model from the world of conventional software threads when writing an application in μ TC.

For a microthreaded target no scheduling is necessary as the constructs in μ TC map onto binary instructions and even a family of single-instruction threads can be created and scheduled with little or no overhead. In fact threaded code will often show super-linear speedup on a microthreaded processor [2] as thread index management is implemented in hardware and does not require the increment and test instructions to be generated as a part of the thread code, would be the case if the thread were executed as a loop.

6. Conclusions

A language μ TC has been defined and is currently being implemented using gcc targeting microthreaded chip-multiprocessors. Prior work has used hand-compiled code kernels to produce the results published in [2]. The analysis involved in developing this language has allowed us to extend the microthreading model to capture recursive concurrency and our CMP simulator has been updated to reflect the semantics captured by this. With a μ TC compiler and this updated simulator it will be possible to simulate much more significant benchmarks, as well as supporting work within the AETHER project.

The main difference between this and prior work is that μ TC code is schedule invariant and based on the following assumptions:

- There is a synchronising memory, which is limited in size and which holds local scalar variables. This memory is used to synchronise between executing threads and between a thread and the shared non-synchronising memory.
- The latter is assumed to have arbitrary delay and to be bulk synchronous with respect to a given family of threads.

- Local synchronising memory is shared to provide communication between threads. This sharing however, is restricted to linear chains, which reflect the locality of communication in silicon.

N.b. the model could be generalised to provide planar local sharing, or indeed arbitrary communication between threads. However, the simplest model has been adopted until the necessity of generalising it further it can be shown.

We already have an interpreter for μ TC, which creates a static sequential schedule from it in C, which can be compiled to any target and we are currently working on two compilers, one from μ TC to microthreaded binaries and the other a parallelising compiler from C to μ TC. Collaborators from Hertfordshire University are working on compilers from Snet and SAC to μ TC.

7. Acknowledgements

I would like to acknowledge input in the form of discussions on μ TC from Thomas Bernard, Konstantinos Bousias, Peter Knijnenburg and Mike Lankamp (University of Amsterdam) and Sven-bodo Scholz (University of Hertfordshire). Support for this research is gratefully acknowledged from NWO in funding the MicroGrids project and from the European Union in funding the AETHER project. Without their support, this work would not have been possible.

8. References

- [1] Jesshope C. R. (2005) Microthreading – a model for distributed instruction-level concurrency, to be published, *Parallel processing Letters*, see: [http://staff.science.uva.nl/~jesshope/Papers/ \$\mu\$ -thread.pdf](http://staff.science.uva.nl/~jesshope/Papers/μ-thread.pdf).
- [2] Bousias, K, Hasasneh N M and Jesshope C R (2006) Instruction-level parallelism through microthreading - a scalable Approach to chip multiprocessors, *Computer Journal*, **49** (2), pp 211-233.
- [3] OpenMP (2005) *OpenMP Version 2.5 Specification*, (accessed 16/4/2006), http://www.openmp.org/drupal/mp-documents/draft_spec25.pdf.
- [4] E A Lee (2006) The Problem With Threads, *IEEE Computer*, **36**, (5), May 2006, pp 33-42.
- [5] X Tian, M Girkar , A Bik and H Saito (2005) Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs, *The Computer Journal*, **48**(5), pp588-601.
- [6] Sven-Bodo Scholz (2003) Single Assignment C - Efficient Support for High-Level Array Operations in a Functional Setting, *Journal of Functional Programming*, **13**, (6) pp1005-1059.
- [7] A.Shafarenko (2006) *The principles and construction of SNet*, Internal report, Dept of Computer Science, University of Hertfordshire.
- [8] Bousias, K. and Jesshope, C. R. (2005) The challenges of massive on-chip concurrency. *Tenth Asia-Pacific Computer Systems Architecture Conference*, Singapore, October 24-26. LNCS 3740, pp. 157-170. Springer-Verlag.