

# $\mu$ TC, a programming parallel language for chip multiprocessors

***Thomas Bernard, Chris Jesshope***  
***Computer Systems Architecture***  
***Universiteit van Amsterdam***



# Summary

- Introduction,
- The  $\mu$ TC language,
- Compiler suite and its tools,
- Progress,
- Future work.



# Why another parallel language ?

- Many parallel programming languages:
  - MPI(~90%), not a programming language but API,
  - Open MP(~10%), pragmas and directives
    - N.B. %s from Guang Gao's paper.

# Why another parallel language ?

- Many parallel programming languages:
  - MPI(~90%), not a programming language but API,
  - Open MP(~10%), pragmas and directives
    - N.B. %s from Guang Gao's paper.
- $\mu$ TC is an intermediate language:
  - captures the microthreaded model of concurrency: memory and synchronisation,
  - being used in the AETHER and MicroGrids projects as a target and source language for compilers.

# Background - Microgrids

- This project is the father of  $\mu$ TC.
- It designs microthreaded microprocessors that support the scheduling of small code fragments:
  - from single instructions to complete applications.
- Concurrency in hardware uses many such processors on reconfigurable chips:
  - Constructs in  $\mu$ TC map directly onto instructions in the microthreaded ISA.

# Microthreading model and $\mu$ TC

- Microthreading is captured in the  $\mu$ TC language
  - it is a low level language for its intended target
    - namely microthreaded microgrids.

# Microthreading model and $\mu$ TC

- Microthreading is captured in the language  $\mu$ TC
  - it is a low level language for its intended target - namely microthreaded microgrids.
- The concepts and constructs implemented in hardware with the following characteristics:
  - *blocking threads* - support data-driven execution
  - *families of threads* - heterogeneous or homogeneous
  - *fine-grain synchronisation* - between threads (in a family)
  - *bulk synchronisation* - between families
  - *pre-emption* - of families not individual threads

# $\mu$ TC language



# State-of-the-art

- Extensions to the C language:
  - Cilk, MIT.
  - Split-C, Berkeley University.
  - UPC, Berkeley University.
- Open MP, uses pragmas and directives.

# What is $\mu$ TC ?

- An extension of the C language, Standard C99.
- Its goal is to support the microthreading model: capture explicitly concurrency in the code.
- New keywords and concurrent structures which map to low-level operations (ISA- $\mu$ T instructions).
- Intermediate language, not user language.

# Microthreaded ISA instructions

- The microthreaded ISA is a union of any ISA with *ISA- $\mu$ t* :
  - **Create**: creates a family of threads - yields family id *fid*,
  - **Break**: breaks execution of a family internally
    - stops allocating new threads and kills all active threads,
  - **Kill**: kills a specified family like break but from another family,
  - **Squeeze fid**: squeezes a family, preempts a concurrent section or family so that it can be restarted on other resources
    - stops allocating threads and allows allocated threads to complete any outstanding synchronizations,

# New keywords added to C99

– It supports constructs:

- **Create** families of threads (parameters and code),
- **Thread**, function which specifies a microthread code.
- **Break** from within a family so that any thread can dynamically terminate its family ,
- **Sync** on termination of threads in a family.

– And from an concurrent control thread:

- **Kill** or **Squeeze** (pre-empt) a running family.

– Data dependency between microthreads:

- **Shared** specifies a shared variable.

– Iterator within the family of microthread:

- **Index** defines an iterator within the family.

# Generates a family of threads (1)

```
/* no dependency between microthreads */

int main(void)
{
    int fid;
    int A[100], n=100;

    create(fid; 1; n)
    {
        index int idx;
        A[idx] = n + idx;
    }
    sync(fid);
}
```

# Generates a family of threads (2)

```
/* basic dependency between microthreads */
int main(void)
{
    int fid;
    int *a, s_init=0, n=100;
    create(fid; 1; n)
    {
        index int idx;
        shared int s = s_init;
        s = s + a[idx];
    }
    sync(fid);

    /* s_init receives s written by the last thread when
    sync completes*/
}
```

# Generates a family of threads (3)

```
thread sumint (shared int s, int a[])
{
    index int idx;
    s = s + a[idx];
}

int main(void)
{
    int fid;
    int *a, s_init=0, n=100;

    create(fid; 1; n) sumint (s_init, *a);

    sync(fid);

    /* s_init receives s written by the last thread when
    sync completes*/
}
```

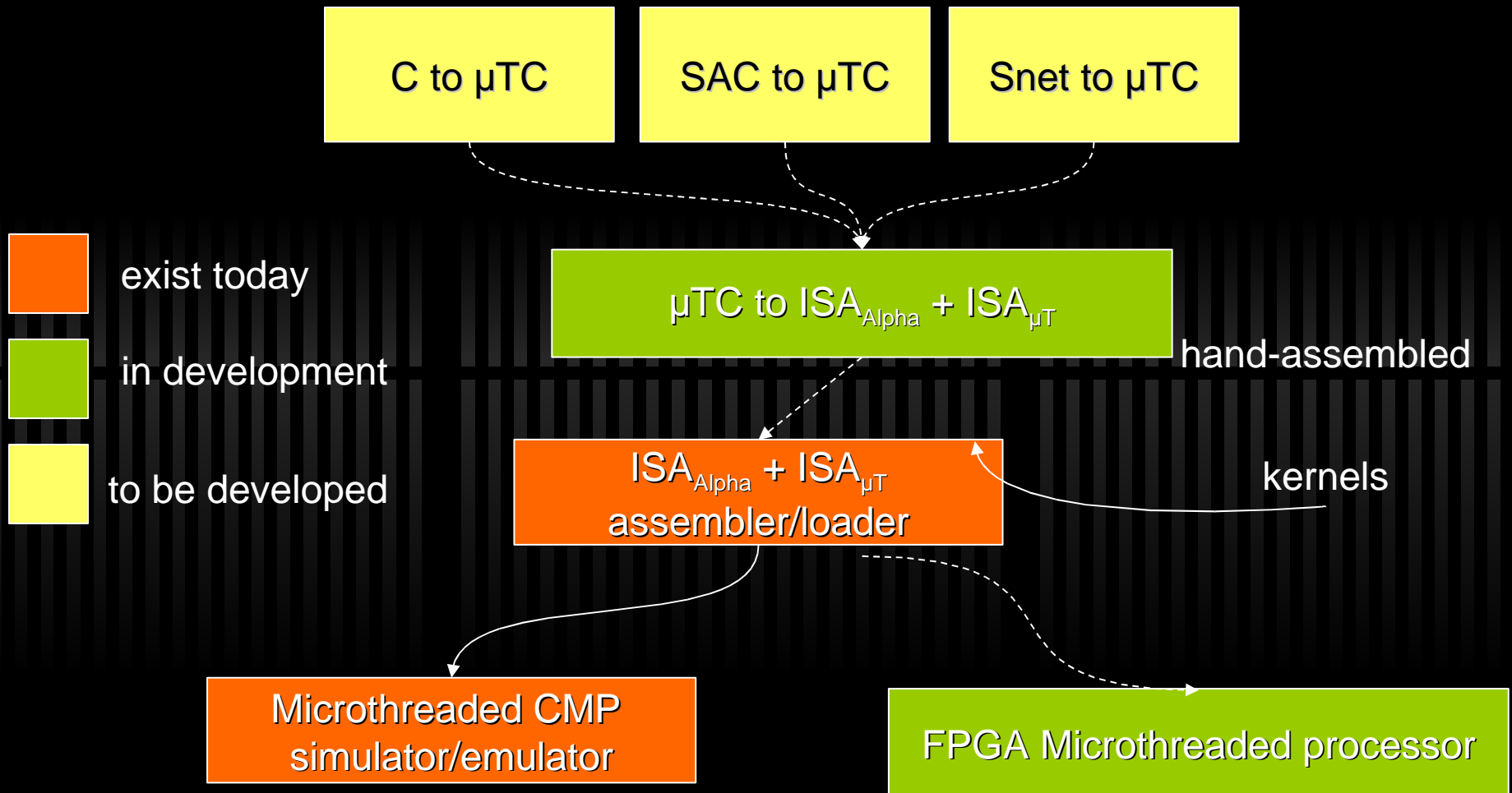
# Relation between C and $\mu$ TC

```
int main(void){
  for(i=0; i<n; i++){
    int s = 0;
    for(j=0; j<n; j++){
      s = s + a[i][j]*x[j]
    }
    y[i] = s;
  }
}
```

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

# Compiler suite for $\mu$ TC

# The *microthreaded* tool chain



# C-to- $\mu$ TC Compiler

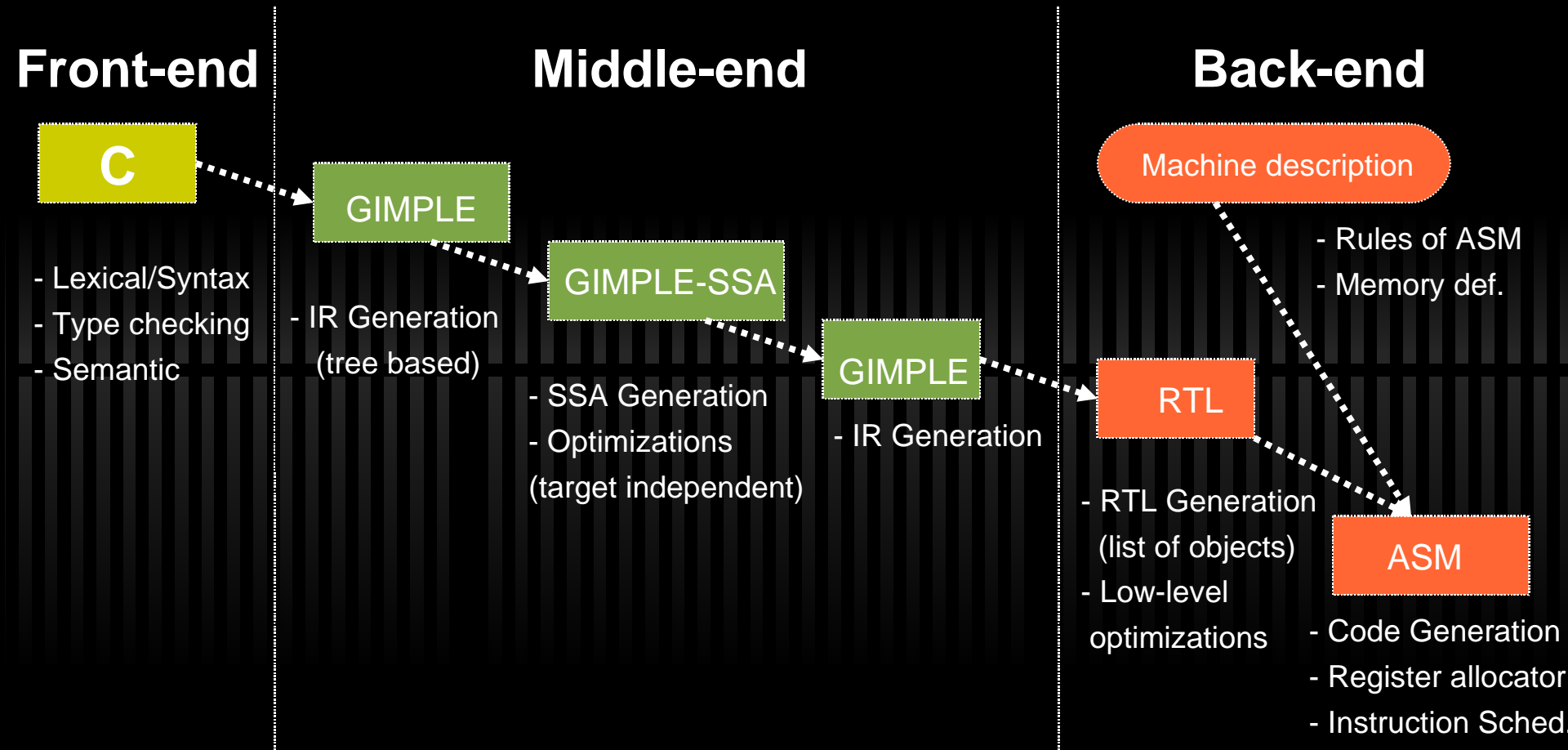
- CoSy system, Ace - Associated Computers Expert
  - Website: [www.ace.nl](http://www.ace.nl).
- Source-to-source compiler.
- Extract concurrency from a C source file.
- High-level code analysis on tree representation of the program.
- $\mu$ TC contains no scheduling information, need only discover parallelism.

# $\mu$ TC-to-binaries Compiler

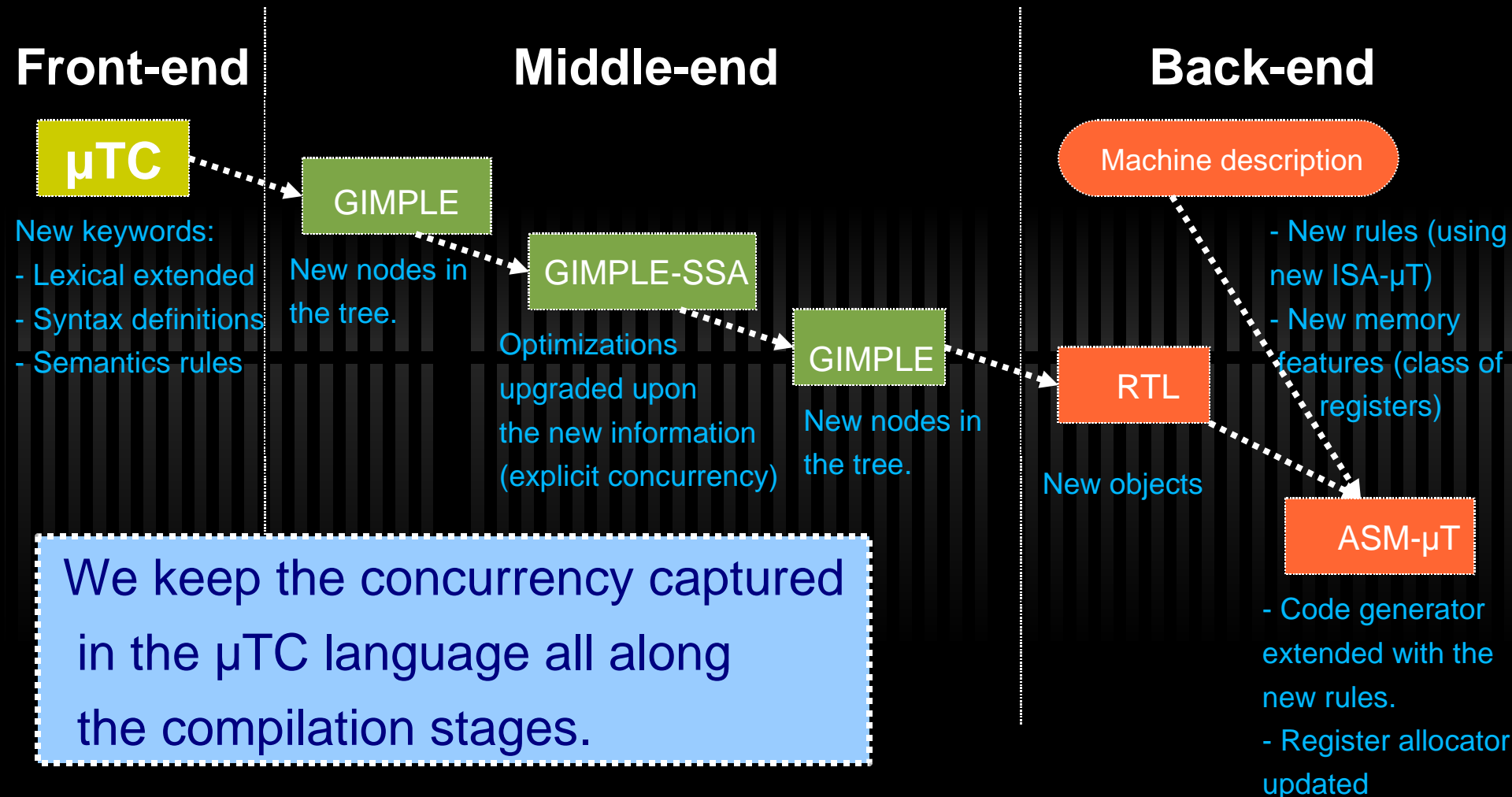
- Our core compiler.
- Translates  $\mu$ TC to binaries.
- Uses GNU GCC 4.1 compiler framework.
  - Website: [gcc.gnu.org](http://gcc.gnu.org)
- New features in our GCC:
  - New language:  $\mu$ TC,
  - And new target: ISA{alpha} + ISA{ $\mu$ T}.

# Progress and experience

# Structure of GCC 4.1



# Design of GCC- $\mu$ T 4.1 (in blue, new)



# What is working ?

- **Front-end:**
  - parsing a full  $\mu$ TC source code,
  - dropping syntax and basic semantic errors/warnings.
- **Middle-end:**
  - generation of the GIMPLE (tree-based) of the  $\mu$ TC program (**shared**, **index**, **break**, **kill**, **squeeze** and **sync**).
- **Back-end:**
  - generation a basic low-level representation (RTL) for some  $\mu$ TC single statements (no registers allocation).

# Experience & issues

- Large amount of code (~500.000 lines),
- Different stages not clearly defined,
- Many targets available (>30 architectures),
- Many optimizations (>150),
- Be patient ...

# Future Work



# Next features in the compiler

## – Front-end:

- Handle all use cases of  $\mu$ TC: semantics checks.

## – Middle-end:

- Generate the GIMPLE for complex statements (**create** and **thread**).

## – Back-end:

- Define all the rules for the machine description files (alpha),
- Extend the RTL with all the notions of concurrency of our model,
- Extend the register allocator,
- Generate some assembly code.

# Future work in compilers

- Optimizations (within GCC) on the code.
- Analysis on the  $\mu$ TC code (**create** statement).
- Link-time optimizations (need investigations).
- C-to- $\mu$ TC compiler (new PhD starting 2007).

# Questions ?

