Baremetal programming on Intel SCC
A Tutorial for MARC Symposium at ONERA 19th – 20th July 2012

Eric Noulard and Johannes Scheller

<surname>.<name>@onera.fr

Friday 20th July, 2012
Outline

1. Baremetal programming
2. Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API
3. Time management
   - Intel SCC mechanisms
   - API
4. MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake
5. Conclusion
6. References
1. Baremetal programming

2. Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API

3. Time management
   - Intel SCC mechanisms
   - API

4. MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake

5. Conclusion

6. References
Baremetal programming

Bare metal programming is a way to program a computer with a minimalistic set of [software] tools, the name comes from a reverse sentence: programming on the bare metal [of the processor].

- Programming directly on the underlying hardware
- Generally by a mix of:
  - Low level programming language (i.e. Assembler)
  - C language

More references here: http://sites.onera.fr/scc/baremetal
Baremetal programming

Bare metal programming is a way to program a computer with a minimalistic set of [software] tools, the name comes from a reverse sentence: programming on the bare metal [of the processor].

- Programming directly on the underlying hardware
- Generally by a mix of:
  - Low level programming language (i.e. Assembler)
  - C language

More references here: http://sites.onera.fr/scc/baremetal
Bare metal programming is a way to program a computer with a minimalistic set of [software] tools, the name comes from a reverse sentence: programming on the bare metal [of the processor].

- Programming directly on the underlying hardware
- Generally by a mix of:
  - Low level programming language (i.e. Assembler)
  - C language

More references here: http://sites.onera.fr/scc/baremetal
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \( \leadsto \) control WCET.
- Control when tasks communicate \( \leadsto \) evaluate WCTT.
- true MIMD (not SPMD) \( \leadsto \) minimize local memory footprint
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\leadsto\) control WCET.
- Control when tasks communicate \(\leadsto\) evaluate WCTT.
- true MIMD (not SPMD) \(\leadsto\) minimize local memory footprint
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\leadsto\) control WCET.
- Control when tasks communicate \(\leadsto\) evaluate WCTT.
- true MIMD (not SPMD) \(\leadsto\) minimize local memory footprint
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\sim\) control WCET.
- Control when tasks communicate \(\sim\) evaluate WCTT.
- true MIMD (not SPMD) \(\sim\) minimize local memory footprint
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\sim\) control WCET.
- Control when tasks communicate \(\sim\) evaluate WCTT.
- \textit{true} MIMD (not SPMD) \(\sim\) minimize local memory footprint
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\leadsto\) control WCET.
- Control when tasks communicate \(\leadsto\) evaluate WCTT.
- true MIMD (not SPMD) \(\leadsto\) minimize local memory footprint
The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\leadsto\) control WCET.
- Control when tasks communicate \(\leadsto\) evaluate WCTT.
- true MIMD (not SPMD) \(\leadsto\) minimize local memory footprint
Baremetal programming: why?

The main arguments for programming on the bare metal usually are:

- A quest for maximum speed of execution
- Need to tackle with memory space scarcity issues
- Avoid Operating System overhead and/or perturbation
- More generally handle things only the way you want

SCC baremetal solutions

There are currently two solutions: ETI framework and baremichael framework.

Baremetal for real-time on the SCC[2]:

- Deterministic execution on NoC-based many-core \(\leadsto\) control WCET.
- Control when tasks communicate \(\leadsto\) evaluate WCTT.
- true MIMD (not SPMD) \(\leadsto\) minimize local memory footprint
Message passing:

- DRAM used for message data
- MPB used for signalling

First experiments focused on [2]:

- Local clock offsets
- Bandwidth

Further information available at:

- http://communities.intel.com/docs/DOC-5913

- **Message passing:**
  - DRAM used for message data
  - MPB used for signalling
- First experiments focused on [2]:
  - Local clock offsets
  - Bandwidth

Further information available at:
Message passing:
- DRAM used for message data
- MPB used for signalling

First experiments focused on [2]:
- Local clock offsets
- Bandwidth

Further information available at:
- http://communities.intel.com/docs/DOC-5913
ETI Baremetal


- Message passing:
  - DRAM used for message data
  - MPB used for signalling
- First experiments focused on [2]:
  - Local clock offsets
  - Bandwidth

Further information available at:
- http://communities.intel.com/docs/DOC-5913
ETI Baremetal


Message passing:
- DRAM used for message data
- MPB used for signalling

First experiments focused on [2]:
- Local clock offsets
- Bandwidth

Further information available at:
- http://communities.intel.com/docs/DOC-5913
Message passing:
- DRAM used for message data
- MPB used for signalling

First experiments focused on [2]:
- Local clock offsets
- Bandwidth

Further information available at:
Message passing:
- DRAM used for message data
- MPB used for signalling

First experiments focused on [2]:
- Local clock offsets
- Bandwidth

Further information available at:
- http://communities.intel.com/docs/DOC-5913
BareMichael

- Baremetal framework provided by Michael Ziwisky [7]
- Brings the cores from real into protected mode and jumps to own code
- SPMD message passing support (RCCE V2.0) since BareMichael v6
- Further information can be found at:

BareMichael with ONERA additions

BareMichael v6 (and then v7) was extended at ONERA (Johannes Scheller and Eric Noulard) in order to support various low-level message passing mechanisms, timing primitives and an MIMD programming model.

http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/ONERA/tarballs
BareMetal framework provided by Michael Ziwisky [7]
Brings the cores from real into protected mode and jumps to own code
SPMD message passing support (RCCE V2.0) since BareMichael v6
Further information can be found at:

BareMetal with ONERA additions
BareMichael v6 (and then v7) was extended at ONERA (Johannes Scheller and Eric Noulard) in order to support various low-level message passing mechanisms, timing primitives and an MIMD programming model.

http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/ONERA/tarballs
BareMichael

- Baremetal framework provided by Michael Ziwisky [7]
- Brings the cores from real into protected mode and jumps to own code
- SPMD message passing support (RCCE V2.0) since BareMichael v6
- Further information can be found at:

BareMichael with ONERA additions

BareMichael v6 (and then v7) was extended at ONERA (Johannes Scheller and Eric Noulard) in order to support various low-level message passing mechanisms, timing primitives and an MIMD programming model.

http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/ONERA.tarballs
BareMichael

- Baremetal framework provided by Michael Ziwisky [7]
- Brings the cores from real into protected mode and jumps to own code
- SPMD message passing support (RCCE V2.0) since BareMichael v6
- Further information can be found at:

BareMichael with ONERA additions

BareMichael v6 (and then v7) was extended at ONERA (Johannes Scheller and Eric Noulard) in order to support various low-level message passing mechanisms, timing primitives and an MIMD programming model.

http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/ONERAtarballs
BareMichael

- Baremetal framework provided by Michael Ziwisky [7]
- Brings the cores from real into protected mode and jumps to own code
- SPMD message passing support (RCCE V2.0) since BareMichael v6
- Further information can be found at:

BareMichael with ONERA additions

BareMichael v6 (and then v7) was extended at ONERA (Johannes Scheller and Eric Noulard) in order to support various low-level message passing mechanisms, timing primitives and an MIMD programming model.

http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/ONERA.tarballs
BareMichael

- Baremetal framework provided by Michael Ziwisky [7]
- Brings the cores from real into protected mode and jumps to own code
- SPMD message passing support (RCCE V2.0) since BareMichael v6
- Further information can be found at:

BareMichael with ONERA additions

BareMichael v6 (and then v7) was extended at ONERA (Johannes Scheller and Eric Noulard) in order to support various low-level message passing mechanisms, timing primitives and an MIMD programming model.

http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/ONERAtarballs
BareMichael ONERA additions

libperf

Primitives for easy use of hardware counters like cache misses
BareMichael ONERA additions

libperf
Primitives for easy use of hardware counters like cache misses

libmp
MIMD message passing library
BareMichael ONERA additions

**libperf**
Primitives for easy use of hardware counters like cache misses

**libmp**
MIMD message passing library

**libsnc**
Primitive for synchronizing the SCC cores and building a global clock with local access.
BareMichael ONERA additions

**libperf**
Primitives for easy use of hardware counters like cache misses

**libmp**
MIMD message passing library

**libsync**
Primitive for synchronizing the SCC cores and building a global clock with local access.

**libtime**
Primitive to access local (TSC) and global (GTSC) clocks and time calibration.
BareMichael ONERA additions

**libperf**
Primitives for easy use of hardware counters like cache misses

**libmp**
MIMD message passing library

**libsync**
Primitive for synchronizing the SCC cores and building a global clock with local access.

**libtime**
Primitive to access local (TSC) and global (GTSC) clocks and time calibration.
**BareMichael ONERA additions**

- **libperf**
  Primitives for easy use of hardware counters like cache misses

- **libmp**
  MIMD message passing library

- **libsync**
  Primitive for synchronizing the SCC cores and building a global clock with local access.

- **libtime**
  Primitive to access local (TSC) and global (GTSC) clocks and time calibration.
The BareMichael source tree layout is the following:

```
baremichael
    boot
    compile
    include
    miketerm
    system
    test
    lib
        libxc
        libmp
        libperf
        libsync
        libtime
```

**BareMichael distribution**

Contains a baremichael/README file which explains the role of each directory and files.

**ONERA additions**

ONERA additions adds baremichael/README.cmake which explains the CMake [1] machinery.
1. Baremetal programming

2. Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API

3. Time management
   - Intel SCC mechanisms
   - API

4. MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake

5. Conclusion

6. References
Experimental many-core processor to evaluate hardware assisted message passing with an on-die mesh network

Chip made out of 24 tiles arranged in 6x4 matrix

2 P54C cores per tile

Extensive performance adjustment capabilities
2 cores per tile

- Tile internal interconnection by bus
- Mesh interface unit (MIU) provides bus network interface
- Message passing buffer (MPB) accessible through MIU
- Translation from physical (i.e., virtual) to system address
Tile structure

- 2 cores per tile
- Tile internal interconnection by bus
  - Mesh interface unit (MIU) provides bus network interface
  - Message passing buffer (MPB) accessible through MIU
  - Translation from physical (i.e., virtual) to system address
2 cores per tile

Tile internal interconnection by bus

Mesh interface unit (MIU) provides bus network interface

Message passing buffer (MPB) accessible through MIU

Translation from physical(i.e. virtual) to system address
2 cores per tile
Tile internal interconnection by bus
Mesh interface unit (MIU) provides bus network interface
Message passing buffer (MPB) accessible through MIU
Translation from physical (i.e., virtual) to system address
2 cores per tile

Tile internal interconnection by bus

Mesh interface unit (MIU) provides bus network interface

Message passing buffer (MPB) accessible through MIU

Translation from physical (i.e. virtual) to system address
The address translation is described in SCC EAS [3], it makes it possible to:

- access MPBs
- access CRBs
- access DRAM
- access FPGA
The address translation is described in SCC EAS [3], it makes it possible to:

- access MPBs
- access CRBs
- access DRAM
- access FPGA
The address translation is described in SCC EAS [3], it makes it possible to:

- access MPBs
- access CRBs
- access DRAM
- access FPGA
Address translation

The address translation is described in SCC EAS [3], it makes it possible to:

- access MPBs
- access CRBs
- access DRAM
- access FPGA
The address translation is described in SCC EAS [3], it makes it possible to:

- access MPBs
- access CRBs
- access DRAM
- access FPGA
Memory types

32 Bit logical address → address translation → page table → physical address & access mode

L1

L2

WCB

MIU

LUT's

34 Bit system address

Three memory types [3, 6]
Memory types

32 Bit logical address

address translation

memory mapping

page table

physical address & access mode

Cached

34 Bit system address

L1

L2

WCB

MIU

LUT’s

Cached memory -> L1 -> L2 -> private DRAM/destination

Cached read & write

on cache miss

on cache miss

Three memory types [3, 6]
Memory types

1. **Cached memory** -> L1 -> L2 -> private DRAM/destination

2. **Uncached memory (UC)**  
   -> destination in 1, 2, or 4 byte packages

---

Three memory types [3, 6]
Memory types

1. Cached memory -> L1 -> L2 -> private DRAM/destination
2. Uncached memory (UC) -> destination in 1, 2, or 4 byte packages
3. MPBT memory -> L1 -> destination + extra instruction to invalidate

Three memory types [3, 6]
Memory types

1. Cached memory -> L1 -> L2 -> private DRAM/destination
2. Uncached memory (UC) -> destination in 1, 2, or 4 byte packages
3. MPBT memory -> L1 -> destination + extra instruction to invalidate

Three memory types [3, 6]
Baremetal programming

Message Passing
- Intel SCC mechanisms
- Message Passing Design and Choices
- API

Time management
- Intel SCC mechanisms
- API

MIMD programming
- Simple non-SPMD application example
- Install Baremichael using CMake

Conclusion

References
- Pull mechanism
  - Sender puts data into own MPB and signals the receiver
  - Receiver then pulls the data from the sender's MPB and acknowledges receipt

Locally accessible distributed shared space
Pull mechanism

Sender puts data into own MPB and signals the receiver

Receiver then pulls the data from the sender’s MPB and acknowledges receipt
Pull mechanism

Sender puts data into own MPB and signals the receiver

Receiver then pulls the data from the senders MPB and acknowledges receipt
Push mechanism

- Sender pushes data into receivers MPB and signals the receiver
- Receiver gets the data from own MPB and acknowledges receipt
Locally accessible distributed shared space

- Push mechanism
- Sender **pushes** data into receivers MPB and signals the receiver
- Receiver gets the data from own MPB and acknowledges receipt
Push mechanism

- Sender pushes data into receivers MPB and signals the receiver
- Receiver gets the data from own MPB and acknowledges receipt

Locally accessible distributed shared space
Copy Cores

Core 0

MPB L2 L1

Core 47

MPB L2 L1

Locally accessible distributed shared space

Copy core

MPB L2 L1
Copy Cores

Locally accessible distributed shared space

Copy core

Core 0

Core 47

MPB  L2  L1

MPB  L2  L1

0

47
Copy Cores

Locally accessible distributed shared space

Eric Noulard and Johannes Scheller (ONERA)
Baremetal programming on Intel SCC
Friday 20th July, 2012 18 / 48
Copy Cores

Core 0

Core 47

Locally accessible distributed shared space

Copy core
Copy Cores

Core 0

MPB  L2  L1

Core 47

MPB  L2  L1

Locally accessible distributed shared space

Copy core

MPB  L2  L1

Eric Noulard and Johannes Scheller (ONERA)

Baremetal programming on Intel SCC

Friday 20th July, 2012 18 / 48
Partitioning for PUSH

- 8192$B$ total MPB space available
- 32$B$ one fool line for writes < 32$B$
- 64$B$ for send flags
- 64$B$ for receive flags
- 32$B$ setup space
- 160$B$ remaining per core
**Partitioning for PUSH**

- **Fool line**
- **Send flags**
- **Receive flags**
- **SETUP space**
- **MPB 0**
- **MPB 47**

- 8192\(B\) total MPB space available
- 32\(B\) one fool line for writes < 32\(B\)
- 64\(B\) for send flags
- 64\(B\) for receive flags
- 32\(B\) setup space
- 160\(B\) remaining per core
Partitioning for PUSH

- **8192B** total MPB space available
- **32B** one fool line for writes < **32B**
- **64B** for send flags
- **64B** for receive flags
- **32B** setup space
- **160B** remaining per core
Partitioning for PUSH

- **8192B** total MPB space available
- **32B** one fool line for writes < 32B
- **64B** for send flags
- **64B** for receive flags
- **32B** setup space
- **160B** remaining per core
Partitioning for PUSH

- 8192B total MPB space available
- 32B one fool line for writes < 32B
- 64B for send flags
- 64B for receive flags
- 32B setup space
- 160B remaining per core
Partitioning for PUSH

- 8192B total MPB space available
- 32B one fool line for writes < 32B
- 64B for send flags
- 64B for receive flags
- 32B setup space
- 160B remaining per core
Partitioning for PULL

- 8192B total MPB space available
- 32B one fool line for writes < 32B
- 64B for send flags
- 64B for receive flags
- 32B setup space
- 8000B single send space
8192B total MPB space available
32B one fool line for writes < 32B
- 64B for send flags
- 64B for receive flags
- 32B setup space
- 8000B single send space
Partitioning for PULL

- **8192B** total MPB space available
- **32B** one fool line for writes < 32B
- **64B** for send flags
  - **64B** for receive flags
- **32B** setup space
- **8000B** single send space
Partitioning for PULL

- **Single fool line**
  - Fool line
  - Send flags
  - Receive flags
  - SETUP space

- **Single send byte flags (64 Byte)**

- **8192B total MPB space available**
- **32B one fool line for writes < 32B**
- **64B for send flags**
- **64B for receive flags**
  - 32B setup space
  - 8000B single send space
Partitioning for PULL

- 8192\(B\) total MPB space available
- 32\(B\) one fool line for writes < 32\(B\)
- 64\(B\) for send flags
- 64\(B\) for receive flags
- 32\(B\) setup space
- 8000\(B\) single send space
Partitioning for PULL

- 8192\(^B\) total MPB space available
- 32\(^B\) one fool line for writes < 32\(^B\)
- 64\(^B\) for send flags
- 64\(^B\) for receive flags
- 32\(^B\) setup space
- 8000\(^B\) single send space
1. Baremetal programming

2. Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API

3. Time management
   - Intel SCC mechanisms
   - API

4. MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake

5. Conclusion

6. References
Receive and send functions

- MPBinit(void)
- int MPBsend(void* buffer, int* Target, int numDest, int numBytes, int type)
- int MPBreceive(void* buffer, int Source, int numBytes, int type)
Receive and send functions

- **int MPBinit(void)**
  - this function is necessary even for non real-time applications to initialize the MPB partitioning
  - stores MPB location in local data structure
  - provides test&set register location for cores
  - cleans the MPBs

- **int MPBsend(void* buffer, int* Target, int numDest, int numBytes, int type)**
- **int MPBreceive(void* buffer, int Source, int numBytes, int type)**
Receive and send functions

- **MPBinit(void)**
- **int MPBsend(void* buffer, int* Target, int numDest, int numBytes, int type)**
  - sends data given in buffer from local core to Target core
  - type of send defined by type argument (MPB_BLOCKING, MPB_NONBLOCKING, MPB_ASYNC)
  - ex: send “coucou” to core 7
- **int MPBreceive(void* buffer, int Source, int numBytes, int type)**

```c
int CoreTarget = 7
unsigned char buffer[] = {'c','o','u','c','o','u'};
check = MPBsend((void*)(buffer[0]), &CoreTarget, 1, sizeof(buffer), MPB_NONBLOCKING);
```
Receive and send functions

- MPBinit(void)
- int MPBsend(void* buffer, int* Target, int numDest, int numBytes, int type)
- int MPBreceive(void* buffer, int Source, int numBytes, int type)
  - receives data from Source core and stores it in buffer
  - type of receive defined by type argument (MPB_BLOCKING, MPB_NONBLOCKING, MPB_ASYNC)
  - ex: receive from core 0

```c
int CoreTarget = 0
unsigned char buffer[15];
check = MPBreceive((void*)(&buffer[0]), CoreTarget, sizeof(buffer), MPB_NONBLOCKING);
```
Receive and send functions

- MPBinit(void)
- int MPBsend(void* buffer, int* Target, int numDest, int numBytes, int type)
- int MPBreceive(void* buffer, int Source, int numBytes, int type)

PUSH and PULL are specified at compile time.
Baremetal programming

Message Passing
- Intel SCC mechanisms
- Message Passing Design and Choices
- API

Time management
- Intel SCC mechanisms
- API

MIMD programming
- Simple non-SPMD application example
- Install Baremichael using CMake

Conclusion

References
Timestamp counters

Two [hardware] time-related counters available from each core [4]:

1. Local timestamp counter (a.k.a. TSC):
   - locally accessible via RDTSC instruction
   - reference to local processor clock which depends on the frequency chosen for the tile (in our case 533 MHz)

2. Global timestamp counter (a.k.a. GTSC):
   - accessible via the mesh network
   - with base address: 0xf9000000
   - lower 32 bits at offset 0x08224
   - upper 32 bits at offset 0x08228
   - reference to FPGA clock (125 MHz)
1 Baremetal programming

2 Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API

3 Time management
   - Intel SCC mechanisms
   - API

4 MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake

5 Conclusion

6 References
Time management functions

- int sync_cores(unsigned long long offset, long calibration)
- unsigned long long calc_offset()
- unsigned long long LocalTimestamp()
- unsigned long long GlobalTimestamp()
Time management functions

- `int sync_cores(unsigned long long offset, long calibration)`
  - calculates the cores’ offset to the global clock
  - mastercore defines start time
  - provides both 1 ms sleep calibration and local offset factor

- `unsigned long long calc_offset()`
- `unsigned long long LocalTimestamp()`
- `unsigned long long GlobalTimestamp()`
Time management functions

- `int sync_cores(unsigned long long offset, long calibration)`
- `unsigned long long calc_offset()`
  - calculates the cores’ offset to the global clock
- `unsigned long long LocalTimestamp()`
- `unsigned long long GlobalTimestamp()`
Time management functions

- `int sync_cores(unsigned long long offset, long calibration)`
- `unsigned long long calc_offset()`
- `unsigned long long LocalTimestamp()` - gets the value of the rdtsc
- `unsigned long long GlobalTimestamp()`
Time management functions

- int sync_cores(unsigned long long offset, long calibration)
- unsigned long long calc_offset()
- unsigned long long LocalTimestamp()
- unsigned long long GlobalTimestamp()
  - gets the current value of the global clock
1. Baremetal programming

2. Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API

3. Time management
   - Intel SCC mechanisms
   - API

4. MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake

5. Conclusion

6. References
NON-SPMD pingpong example
done using one sender and one receiver in MPB_BLOCKING mode
different packetsizes from 24B to 32kB
both PUSH and PULL evaluated

PINGPONG
All code can be found in the Onera Tarball in the directory:
baremichael/test/tutorial/pingpong
```c
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBscce.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
... 
};
```

Definition of simulation parameters

- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core(else through define)
- define send data size via array
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBsc.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
... 
};

Definition of simulation parameters

- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core(else through define)
- define send data size via array
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBscn.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
  ...
};

Definition of simulation parameters
  * Max. size 32768 B
  * 5 Measurement repetitions
    - static definition of the target/source core (else through define)
    - define send data size via array
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBsc.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
... }

Definition of simulation parameters
- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core (else through define)
- define send data size via array
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBsc.c"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
... 
};

Definition of simulation parameters
- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core(else through define)
- define send data size via array
#include <stddef.h>
#include <stdio.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBscch.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
...}

Definition of simulation parameters
- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core (else through define)
- define send data size via array
```c
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBsc.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

#define sendArray = {
... 
};
```

**Definition of simulation parameters**
- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core (else through define)
- define send data size via array
```c
#include <stddef.h>
#include <stdio.h>
#include <scc.h>
#include <apic.h>
#include <interrupt.h>
#include <clock.h>
#include "hwcounter.h"
#include "counters.h"
#include "MPBscc.h"
#include "sync.h"

#define PACKAGESIZE 32768
#define SCC_CLOCK_PER_USEC 533
#define ITERATIONS 5

#ifndef TARGET
#define TARGET 13
#endif

int sendArray[] = {
    ... 
};
```

Definition of simulation parameters
- Max. size 32768 B
- 5 Measurement repetitions
- static definition of the target/source core(else through define)
- define send data size via array
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((& offset),(& calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, sendBytes, MPB_BLOCKING);
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, sendBytes, MPB_BLOCKING);
        }
    }
    ...
}
```

- Initialize MPB
- Synchronize the cores
- Send & receive the data
```c
int main(void) {
...
me = get_my_coreid();
MPBinit();
sync_cores((& offset),(& calibration));
...
for (j = 0; j < ITERATIONS; j++) {
    for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
        ...
        check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, sendBytes, MPB_BLOCKING);
        check = MPBreceive((void*)(&buffer[0]), CoreTarget, sendBytes, MPB_BLOCKING);
    }
}
...
```

- Initialize MPB
- Synchronize the cores
- Send & receive the data
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, sendBytes, MPB_BLOCKING);
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, sendBytes, MPB_BLOCKING);
        }
    }
}
```

- Initialize MPB
- Synchronize the cores
- Send & receive the data
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset), (&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, sendBytes, MPB_BLOCKING);
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, sendBytes, MPB_BLOCKING);
        }
    }
    ...
}
```

- Initialize MPB
- Synchronize the cores
- Send & receive the data
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, sendBytes, MPB_BLOCKING);
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, sendBytes, MPB_BLOCKING);
        }
    }
    ...
}
```

- Initialize MPB
- Synchronize the cores
- Send & receive the data
```c
int main(void) {
    ... 
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ... 
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, sendBytes, MPB_BLOCKING);
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, sendBytes, MPB_BLOCKING);
        }
    } 
    ... 
    }
```
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBreceive((void *)&buffer[0], CoreTarget, recbytes, MPB_BLOCKING);
            check = MPBsend((void *)&buffer[0], &CoreTarget, 1, recbytes, MPB_BLOCKING);
        }
    }
}
```

- Initialize MPB
- Synchronize the cores
- Receive & send the data
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBreceive((void *)&buffer[0], CoreTarget, recbytes, MPB_BLOCKING);
            check = MPBsend((void *)&buffer[0], &CoreTarget, 1, recbytes, MPB_BLOCKING);
        }
    }
}
```

- Initialize MPB
- Synchronize the cores
- Receive & send the data
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, recbytes, MPB_BLOCKING);
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, recbytes, MPB_BLOCKING);
        }
    }
}
```

- **Initialize MPB**
- **Synchronize the cores**
- **Receive & send the data**
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBreceive((void *)&buffer[0], CoreTarget, recbytes, MPB_BLOCKING);
            check = MPBsend((void *)&buffer[0], &CoreTarget, 1, recbytes, MPB_BLOCKING);
        }
    }
}
```

- Initialize MPB
- Synchronize the cores
- Receive & send the data
```c
int main(void) {
    ... 
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBreceive((void*)(&buffer[0]), CoreTarget, recbytes, MPB_BLOCKING);
            check = MPBsend((void*)(&buffer[0]), &CoreTarget, 1, recbytes, MPB_BLOCKING);
        }
    }
}
```

- Initialize MPB
- Synchronize the cores
- Receive & send the data
```c
int main(void) {
    ...
    me = get_my_coreid();
    MPBinit();
    sync_cores((&offset),(&calibration));
    ...
    for (j = 0; j < ITERATIONS; j++) {
        for (i = 0; i < sizeof(sendArray) / sizeof(sendArray[0]); i++) {
            ...
            check = MPBreceive((void *)(&buffer[0]), CoreTarget, recbytes, MPB_BLOCKING);
            check = MPBsend((void *)(&buffer[0]), &CoreTarget, 1, recbytes, MPB_BLOCKING);
        }
    }
}
```

- Initialize MPB
- Synchronize the cores
- Receive & send the data
Hi -- I'm core 2 with pingpong_send.c!

Sending packets to: 14

Pingpong

02]: Packagesize: 37768 Minimum time: 518568 (clockcycles) 1160 (microseconds) Maximum time: 1134200 (clockcycles) 2127 (microseconds)
02]: Packagesize: 31264 Minimum time: 599605 (clockcycles) 1124 (microseconds) Maximum time: 600553 (clockcycles) 1126 (microseconds)
02]: Packagesize: 24000 Minimum time: 461484 (clockcycles) 865 (microseconds) Maximum time: 461971 (clockcycles) 866 (microseconds)
02]: Packagesize: 19968 Minimum time: 385645 (clockcycles) 723 (microseconds) Maximum time: 386244 (clockcycles) 724 (microseconds)
02]: Packagesize: 16384 Minimum time: 314101 (clockcycles) 589 (microseconds) Maximum time: 314617 (clockcycles) 590 (microseconds)
02]: Packagesize: 12288 Minimum time: 210469 (clockcycles) 394 (microseconds) Maximum time: 211043 (clockcycles) 395 (microseconds)
02]: Packagesize: 8004 Minimum time: 105258 (clockcycles) 197 (microseconds) Maximum time: 105310 (clockcycles) 197 (microseconds)
02]: Packagesize: 6004 Minimum time: 69411 (clockcycles) 128 (microseconds) Maximum time: 69696 (clockcycles) 128 (microseconds)
02]: Packagesize: 4004 Minimum time: 45640 (clockcycles) 87 (microseconds) Maximum time: 46955 (clockcycles) 88 (microseconds)
02]: Packagesize: 2004 Minimum time: 25286 (clockcycles) 49 (microseconds) Maximum time: 26543 (clockcycles) 49 (microseconds)
02]: Packagesize: 1004 Minimum time: 15193 (clockcycles) 30 (microseconds) Maximum time: 16479 (clockcycles) 30 (microseconds)
02]: Packagesize: 504 Minimum time: 10717 (clockcycles) 20 (microseconds) Maximum time: 10825 (clockcycles) 20 (microseconds)
02]: Packagesize: 254 Minimum time: 8123 (clockcycles) 15 (microseconds) Maximum time: 8216 (clockcycles) 15 (microseconds)
02]: Packagesize: 124 Minimum time: 7347 (clockcycles) 13 (microseconds) Maximum time: 7701 (clockcycles) 14 (microseconds)
02]: Packagesize: 62 Minimum time: 7154 (clockcycles) 13 (microseconds) Maximum time: 7289 (clockcycles) 13 (microseconds)
02]: Packagesize: 32 Minimum time: 6330 (clockcycles) 11 (microseconds) Maximum time: 6531 (clockcycles) 12 (microseconds)
02]: Packagesize: 24 Minimum time: 5845 (clockcycles) 10 (microseconds) Maximum time: 5942 (clockcycles) 11 (microseconds)
02]: SUCCESS
02]: Thanks for using libMP
02]: GOODBYE
PING and PONG

- NON-SPMD sender and receiver example
- Single sender which will send for 12 s every PERIOD using different size packets
- Receiver continously receives
- Allows to evaluate the different modes of communication (MPB_BLOCKING, MPB_NONBLOCKING, MPB_ASYNC)

All code can be found in the Onera Tarball in the directory: baremichael/test/tutorial/pingANDpong
[00]: ******************Simulation DONE******************
[00]: Simulation latency: 1 (microseconds)
[00]: Total frames: 11718 should be: 12000
[00]: Partial frames: 0
[00]: Thanks for using libMP
[00]: GOODBYE

[01]: ******************Simulation DONE******************
[01]: Simulation latency: 1 (microseconds)
[01]: Total frames: 11180 should be: 12000
[01]: Partial frames: 0
[01]: Thanks for using libMP
[01]: GOODBYE

[12]: ******************Simulation DONE******************
[12]: Simulation latency: 1 (microseconds)
[12]: Total frames: 11717
[12]: Partial frames: 0
[12]: Damaged frames: 0
[12]: Wrong addressed frames: 0
[12]: Wrong source frames: 0
[12]: Thanks for using libMP
[12]: GOODBYE

[13]: ******************Simulation DONE******************
[13]: Simulation latency: 1 (microseconds)
[13]: Total frames: 11179
[13]: Partial frames: 0
[13]: Damaged frames: 0
[13]: Wrong addressed frames: 0
[13]: Wrong source frames: 0
[13]: Thanks for using libMP
[13]: GOODBYE
1. Baremetal programming

2. Message Passing
   - Intel SCC mechanisms
   - Message Passing Design and Choices
   - API

3. Time management
   - Intel SCC mechanisms
   - API

4. MIMD programming
   - Simple non-SPMD application example
   - Install Baremichael using CMake

5. Conclusion

6. References
Using CMake: why?

BareMichael makefile

The unmodified BareMichael use a Makefile located in baremichael/compile. It has been crafted to produce SPMD binaries.

CMake CMakeLists.txt

ONERA additions uses CMake in order to try to make it simpler to build a MIMD user application

- CMake supports out-of-source build out-of-the-box so we can live aside from original Makefile.
- CMake is a portable build system generator (should be possible to work on Windows as well)
- CMake has relatively simple macro language which makes it possible to wrap and automatized the different building steps
The CMake workflow (pictured)

- **CMakeLists.txt**
- **Project file(s), Makefiles, ...**
- **Object files**
- **Installed files**
- **Binary package**
- **Installed package**

- **Source files**
- **Generated Sources files**
- **Source package**

- **CMake time**
- **Build time**
- **Install time**
- **CPack time**
- **Package Install time**

Eric Noulard and Johannes Scheller (ONERA)
The CMake workflow (pictured)

- **CMakeLists.txt**
- **Project file(s), Makefiles, ...**
- **Source files**
- **Generated Sources files**
- **Object files**
- **Installed files**
- **Binary package**
- **Installed package**

- **Source package**
- **CMake time**
- **Build time**
- **Install time**
- **CPack time**
- **Package Install time**
The CMake workflow (pictured)
The CMake workflow (pictured)
The CMake workflow (pictured)
The CMake workflow (pictured)
Whatever the compilation solution there are several steps to produce a baremetal binary for the SCC:

1. create the reset vector and protected mode binary objects
2. compile the system (boot, system containing assembler and C files) to produce a binary object library
3. compile the users libraries (xc, mp, perf, . . .)
4. create startup binary object for each binary image
5. create one binary object for each user application linked with previously built libraries and other binary objects
6. create memory map for all MIMD applications
7. create object files ready to be loaded on the SCC (sccMerge)
8. send to MARC system (using ssh/scp connection)
SCC BareMichael CMake file set

The CMake build system for BareMichael contains:

- The main baremichael/CMakeLists.txt file which configure the project (discover compilers and SCCKit tools) and the definition of user applications

- one CMakeLists.txt for building each library (i.e. one such file per libXXX directory)

- compile/BMBMFTools.cmake which contains CMake macros definitions which wraps the previously described building steps

- compile/Toolchain-BMBMF-scc.cmake the SCC cross-compiling toolchain file for CMake (see http://www.cmake.org/Wiki/CMake_Cross_Compiling)
Configure your BareMichael source tree

1. unarchive the distribution file
2. create the build directory (may be a separate directory inside the source tree)
3. setup environment variables (may be done in your shell login file)
4. run cmake using the provided cross-compile toolchain

```
$ tar zxvf Baremichael-7.1.onera-Source.tar.gz
$ cd Baremichael-7.1.onera-Source; mkdir b; cd b
$ export SCC_INTELGCC_PATH=<<...>/compilers/i386-unknown-linux-gnu
$ export SCCKIT_PATH_HINT=<<...>/sccKit/release/development/bin
$ cmake -DCMAKE_TOOLCHAIN_FILE=../compile/Toolchain-BMBMF-scc.cmake ..
-- BMBMF:: Using SCC_INTELGCC_PATH=<<...>/compilers/i386-unknown-linux-gnu
-- The C compiler identification is GNU 3.4.5
-- Check for working C compiler: <<...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- Check for working C compiler: <<...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc -- works
[...]
-- The ASM compiler identification is GNU
-- Found assembler: <<...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- FindSCCKit: using PATH HINT: <<...>/sccKit/release/development/bin
-- Found SCCKIT: <<...>/sccKit/release/development/bin/sccMerge
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: <<...>/Baremichael-7.1.onera-Source/b
```
Configure your BareMichael source tree

1. **unarchive the distribution file**

   ```
   $ tar zxvf Baremichael-7.1.onera-Source.tar.gz
   $ cd Baremichael-7.1.onera-Source; mkdir b; cd b
   $ export SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
   $ export SCCKIT_PATH_HINT=<...>/sccKit/release/development/bin
   $ cmake -DCMAKE_TOOLCHAIN_FILE=../compile/Toolchain-BMBMF-scc.cmake ..
   -- BMBMF:: Using SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
   -- The C compiler identification is GNU 3.4.5
   -- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
   -- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc -- works
   -- The ASM compiler identification is GNU
   -- Found assembler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
   -- FindSCCKit: using PATH HINT: <...>/sccKit/release/development/bin
   -- Found SCCKIT: <...>/sccKit/release/development/bin/sccMerge
   [...]  
   -- Configuring done
   -- Generating done
   -- Build files have been written to: <...>/Baremichael-7.1.onera-Source/b
   ```

2. create the build directory (may be a separate directory inside the source tree)

3. setup environment variables (may be done in your shell login file)

4. run cmake using the provided cross-compile toolchain
Configure your BareMichael source tree

1. unarchive the distribution file
2. create the build directory (may be a separate directory inside the source tree)
3. setup environment variables (may be done in your shell login file)
4. run cmake using the provided cross-compile toolchain

```
$ tar zxvf Baremichael-7.1.onera-Source.tar.gz
$ cd Baremichael-7.1.onera-Source; mkdir b; cd b
$ export SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
$ export SCCKIT_PATH_HINT=<...>/sccKit/release/development/bin
$ cmake -DCMAKE_TOOLCHAIN_FILE=../compile/Toolchain-BMBMF-scc.cmake ..
```

```
-- BMBMF:: Using SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
-- The C compiler identification is GNU 3.4.5
-- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- Found assembler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- Configuring done
-- Generating done
-- Build files have been written to: <...>/Baremichael-7.1.onera-Source/b
```
Configure your BareMichael source tree

1. unarchive the distribution file
2. create the build directory (may be a separate directory inside the source tree)
3. setup environment variables (may be done in your shell login file)
4. run cmake using the provided cross-compile toolchain

```
$ tar zxvf Baremichael-7.1.onera-Source.tar.gz
$ cd Baremichael-7.1.onera-Source; mkdir b; cd b
$ export SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
$ export SCCKIT_PATH_HINT=<...>/sccKit/release/development/bin
$ cmake -DCMAKE_TOOLCHAIN_FILE=../compile/Toolchain-BMBMF-scc.cmake ..
```

```
-- BMBMF:: Using SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
-- The C compiler identification is GNU 3.4.5
-- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc -- works
[...]
-- The ASM compiler identification is GNU
-- Found assembler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- FindSCCKit: using PATH HINT: <...>/sccKit/release/development/bin
-- Found SCCKIT: <...>/sccKit/release/development/bin/sccMerge
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: <...>/Baremichael-7.1.onera-Source/b
```
Configure your BareMichael source tree

1. unarchive the distribution file
2. create the build directory (may be a separate directory inside the source tree)
3. setup environment variables (may be done in your shell login file)
4. run cmake, using the provided cross-compile toolchain

```bash
$ tar zxvf Baremichael-7.1.onera-Source.tar.gz
$ cd Baremichael-7.1.onera-Source; mkdir b; cd b
$ export SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
$ export SCCKIT_PATH_HINT=<...>/sccKit/release/development/bin
$ cmake -DCMAKE_TOOLCHAIN_FILE=../compile/Toolchain-BMBMF-scc.cmake ..
```

-- BMBMF:: Using SCC_INTELGCC_PATH=<...>/compilers/i386-unknown-linux-gnu
-- The C compiler identification is GNU 3.4.5
-- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- Check for working C compiler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc -- works
[...]
-- The ASM compiler identification is GNU
-- Found assembler: <...>/i386-unknown-linux-gnu/bin/i386-unknown-linux-gnu-gcc
-- FindSCCKit: using PATH HINT: <...>/sccKit/release/development/bin
-- Found SCCKIT: <...>/sccKit/release/development/bin/sccMerge
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: <...>/Baremichael-7.1.onera-Source/b
Some CMake macro for BMBMF

The file compile/BMBMFTools.cmake contains CMake helper macros:

- **BMBMF_CreateResetVectorBinary**: creates the binary object for reset vector
- **BMBMF_CreateGetProtectedBinary**: creates the binary object containing code to put the processor in protected mode
- **BMBMF_CreateStartupObject**: creates the binary object for starting up a processor
- **BMBMF_CreateOneApplicationImage**: creates one baremetal application binary
- **BMBMF_ParseMappingFile**: creates a set of baremetal applications binaries from a map file
- **BMBMF_MapApplicationsOntoSCC**: create map and ready to load object for SCC
Some CMake macro for BMBMF

The file compile/BMBMFTools.cmake contains CMake helper macros:

- BMBMF_CreateResetVectorBinary: creates the binary object for reset vector
- BMBMF_CreateGetProtectedBinary: creates the binary object containing code to put the processor in protected mode
- BMBMF_CreateStartupObject: creates the binary object for starting up a processor
- BMBMF_CreateOneApplicationImage: creates one baremetal application binary
- BMBMF_ParseMappingFile: creates a set of baremetal applications binaries from a map file
- BMBMF_MapApplicationsOntoSCC: create map and ready to load object for SCC
Some CMake macro for BMBMF

The file compile/BMBMFTools.cmake contains CMake helper macros:

- **BMBMF_CreateResetVectorBinary**: creates the binary object for reset vector
- **BMBMF_CreateGetProtectedBinary**: creates the binary object containing code to put the processor in protected mode
- **BMBMF_CreateStartupObject**: creates the binary object for starting up a processor
- **BMBMF_CreateOneApplicationImage**: creates one baremetal application binary
- **BMBMF_ParseMappingFile**: creates a set of baremetal applications binaries from a map file
- **BMBMF_MapApplicationsOntoSCC**: create map and ready to load object for SCC
Some CMake macro for BMBMF

The file compile/BMBMFTools.cmake contains CMake helper macros:

- **BMBMF_CreateResetVectorBinary**: creates the binary object for reset vector
- **BMBMF_CreateGetProtectedBinary**: creates the binary object containing code to put the processor in protected mode
- **BMBMF_CreateStartupObject**: creates the binary object for starting up a processor
- **BMBMF_CreateOneApplicationImage**: creates one baremetal application binary
- **BMBMF_ParseMappingFile**: creates a set of baremetal applications binaries from a map file
- **BMBMF_MapApplicationsOntoSCC**: create map and ready to load object for SCC
The file `compile/BMBMFTools.cmake` contains CMake helper macros:

- **BMBMF_CreateResetVectorBinary**: creates the binary object for reset vector
- **BMBMF_CreateGetProtectedBinary**: creates the binary object containing code to put the processor in protected mode
- **BMBMF_CreateStartupObject**: creates the binary object for starting up a processor
- **BMBMF_CreateOneApplicationImage**: creates one baremetal application binary
- **BMBMF_ParseMappingFile**: creates a set of baremetal applications binaries from a map file
- **BMBMF_MapApplicationsOntoSCC**: create map and ready to load object for SCC
Some CMake macro for BMBMF

The file compile/BMBMFTools.cmake contains CMake helper macros:

- **BMBMF_CreateResetVectorBinary**: creates the binary object for reset vector
- **BMBMF_CreateGetProtectedBinary**: creates the binary object containing code to put the processor in protected mode
- **BMBMF_CreateStartupObject**: creates the binary object for starting up a processor
- **BMBMF_CreateOneApplicationImage**: creates one baremetal application binary
- **BMBMF_ParseMappingFile**: creates a set of baremetal applications binaries from a map file
- **BMBMF_MapApplicationsOntoSCC**: create map and ready to load object for SCC
Some CMake macro for BMBMF

The file compile/BMBMFTools.cmake contains CMake helper macros:

- BMBMF_CreateResetVectorBinary: creates the binary object for reset vector
- BMBMF_CreateGetProtectedBinary: creates the binary object containing code to put the processor in protected mode
- BMBMF_CreateStartupObject: creates the binary object for starting up a processor
- BMBMF_CreateOneApplicationImage: creates one baremetal application binary
- BMBMF_ParseMappingFile: creates a set of baremetal applications binaries from a map file
- BMBMF_MapApplicationsOntoSCC: create map and ready to load object for SCC
In order to setup your baremetal application you should:

- define the source to compile
- define on which processor core of the SCC the application will be mapped.

Those steps may be done in 2 different ways:

1. using a CMake macro call for each application
2. using a global map file
Building one binary user image

1. give a name to your application
2. specify the sources of your application
3. specify the communication mode of this application
4. specify the list of cores this application will be mapped on

```bash
# Build one application image
BMBMF_CreateOneApplicationImage(
    APPNAME "image00",
    STARTUPOBJ ${CMAKE_CURRENT_BINARY_DIR}/startup.o,
    LDSCRIPT ${PROJECT_SOURCE_DIR}/compile/ld.script,
    IMAGE_ADDRESS ${IMG_ADDR},
    RESET_VECTOR_ADDRESS ${RESET_VEC_ADDR},
    GETPROT_ADDRESS ${GETPROT_ADDR},
    C_SOURCES test/tutorial/pingANDpong/ping.c,
    DEPENDS ${GETPROTECTED_BIN_FILENAME} ${RESET_VECTOR_BIN_FILENAME},
    DEFINES "PULL",
    CORES "00"
)
```
Building one binary user image

1. give a name to your application
2. specify the sources of your application
3. specify the communication mode of this application
4. specify the list of cores this application will be mapped on

```c
#define Build one application image
BMBMF_CreateOneApplicationImage(
    APPNAME "image00",
    STARTUPOBJ ${CMAKE_CURRENT_BINARY_DIR}/startup.o,
    LDSCRIPT ${PROJECT_SOURCE_DIR}/compile/ld.script,
    IMAGE_ADDRESS ${IMG_ADDR},
    RESET_VECTOR_ADDRESS ${RESET_VEC_ADDR},
    GETPROT_ADDRESS ${GETPROT_ADDR},
    C_SOURCES test/tutorial/pingANDpong/ping.c,
    DEPENDS ${GETPROTECTED_BIN_FILENAME} ${RESET_VECTOR_BIN_FILENAME},
    DEFINES "PULL"
    CORES "00" )
```
Building one binary user image

1. give a name to your application
2. specify the sources of your application
3. specify the communication mode of this application
4. specify the list of cores this application will be mapped on

```c
# Build one application image
BMBMF_CreateOneApplicationImage(
    APPNAME "image00"
    STARTUPOBJ ${CMAKE_CURRENT_BINARY_DIR}/startup.o
    LDSCRIPT ${PROJECT_SOURCE_DIR}/compile/ld.script
    IMAGE_ADDRESS ${IMG_ADDR}
    RESET_VECTOR_ADDRESS ${RESET_VEC_ADDR}
    GETPROT_ADDRESS ${GETPROT_ADDR}
    C_SOURCES test/tutorial/pingANDpong/ping.c
    DEPENDS ${GETPROTECTED_BIN_FILENAME} ${RESET_VECTOR_BIN_FILENAME}
    DEFINES "PULL"
    CORES "00"
)
```
Building one binary user image

1. give a name to your application
2. specify the sources of your application
3. specify the communication mode of this application
4. specify the list of cores this application will be mapped on

```c
# Build one application image
BMBMF_CreateOneApplicationImage(
    APPNAME "image00",
    STARTUPOBJ ${CMAKE_CURRENT_BINARY_DIR}/startup.o,
    LDSCRIPT ${PROJECT_SOURCE_DIR}/compile/ld.script,
    IMAGE_ADDRESS ${IMG_ADDR},
    RESET_VECTOR_ADDRESS ${RESET_VEC_ADDR},
    GETPROT_ADDRESS ${GETPROT_ADDR},
    C_SOURCES test/tutorial/pingANDpong/ping.c,
    DEPENDS ${GETPROTECTED_BIN_FILENAME} ${RESET_VECTOR_BIN_FILENAME},
    DEFINES "PULL",
    CORES "00" )
```
Building one binary user image

1. give a name to your application
2. specify the sources of your application
3. specify the communication mode of this application
4. specify the list of cores this application will be mapped on

```c
# Build one application image
BMBMF_CreateOneApplicationImage(
    APPNAME "image00",
    STARTUPOBJ ${CMAKE_CURRENT_BINARY_DIR}/startup.o
    LDSCRIPT ${PROJECT_SOURCE_DIR}/compile/ld.script
    IMAGE_ADDRESS ${IMG_ADDR}
    RESET_VECTOR_ADDRESS ${RESET_VEC_ADDR}
    GETPROT_ADDRESS ${GETPROT_ADDR}
    C_SOURCES test/tutorial/pingANDpong/ping.c
    DEPENDS ${GETPROTECTED_BIN_FILENAME} ${RESET_VECTOR_BIN_FILENAME}
    DEFINES "PULL"
    CORES "00"
)
```
Using a global map file

It may be easier to avoid the multiple call to BMBMF_CreateOneApplicationImage and specify all the mapping inside one file and simply call BMBMF_ParseMappingFile(MAP_FILE mappingTutorial.txt)

```
MASTER 0

CORE 00 FUNCTIONS test/tutorial/pingANDpong/ping.c DEFINES PULL TARGET=12
CORE 0C FUNCTIONS test/tutorial/pingANDpong/pong.c DEFINES PULL SOURCE=0
CORE 01 FUNCTIONS test/tutorial/pingANDpong/ping.c DEFINES PUSH TARGET=13
CORE 0D FUNCTIONS test/tutorial/pingANDpong/pong.c DEFINES PUSH SOURCE=1
CORE 02 FUNCTIONS test/tutorial/pingpong/pingpong_send.c DEFINES PULL TARGET=14
CORE 0E FUNCTIONS test/tutorial/pingpong/pingpong_receive.c DEFINES PULL SOURCE=2
CORE 03 FUNCTIONS test/tutorial/pingpong/pingpong_send.c DEFINES PUSH TARGET=15
CORE 0F FUNCTIONS test/tutorial/pingpong/pingpong_receive.c DEFINES PUSH SOURCE=3
```
Conclusion

- Baremichael permits baremetal programming of the SCC
- Onera additions allowed:
  - MIMD mapping of applications
  - to exchange messages between MIMD cores
  - to perform synchronization between independent cores
  - to access the cache miss counters, the TSC and the GTSC

Perspectives

- Baremetal real-time environment for manycore
- Evaluation of worst-case behavior
References I


Bruno d’Ausbourg, Marc Boyer, Eric Noulard, and Claire Pagetti.
Deterministic execution on many-core platforms: application to the scc.
Technical report, Hasso-Planner Institute, December 2011.
In 4th MARC Symposium, Potsdam, Germany.

Intels Labs.

Intels Labs.
What’s New in SCC Kit 1.4.0.


Questions?