FP7 – 216693 - MULTICUBE Project

MULTI-OBJECTIVE DESIGN SPACE EXPLORATION OF MULTI-PROCESSOR SOC ARCHITECTURES FOR EMBEDDED MULTIMEDIA APPLICATIONS

Deliverable D3.4.1: Initial report on architectural exploration and optimization of MPSoC architectures

Revision [4]

Delivery due date: M18 (June 2009)
Actual submission date: July 24, 2009
Lead beneficiary: ICT

Dissemination Level of Deliverable		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	
Nature of Deliverable		
R	Report	X
P	Prototype	
D	Demonstrator	
O	Other	

Author(s):	*Zhang Hao (ICT), Fan Dongrui (ICT), Vittorio Zaccaria (POLIMI)*		
Reviewer(s):	*Sara Bocchio (STM-Italy), Gianluca Palermo (POLIMI), Cristina Silvano (POLIMI)*		
WP/Task No:	3.4	**Number of pages:**	29
Identifier:	D3.4.1_V1.4	**Dissemination level:**	PU
Issue Date:	June 2009		

Keywords: Multi-Objective Design Space Exploration, Multi-Processor SoC, Architectures, Design Flow Integration, Interfaces

Abstract: This deliverable (issued at M18) presents the results of the activities carried on from M7 to M18 in Task 3.4 (*Architecture Design and Optimization based on Design Space Exploration Flow*) under the leadership of ICT. More in detail, this deliverable represents the most relevant result of the research activities carried out so far in collaboration with the Institute of Computing Technology (Chinese Academy of Science), one of the two Chinese partners of the MULTICUBE Consortium.

This deliverable gives an overview of the multi-core architecture simulator and its XML interface, and offers the first set of results automatically obtained by the integration of this use case in the overall MULTICUBE design exploration flow. More in detail, the multi-core architecture simulator has been integrated with the Multicube Explorer tool according to the XML interface defined in D1.4.1 (Definition of the Specification of the Design Flow Integration). This deliverable also reports the preliminary results automatically obtained by using the Multicube Explorer tool. These results represent the first concrete steps towards the automatic design space exploration of multi-core architectures to be done in Task 3.4. The results of the more comprehensive design space exploration of the multi-core use case will be presented in the final report D3.4.2 to be issued at the end of the project.

The Project Coordinator

Approved by the Project Coordinator: C Silvano	**Date: July 24th, 2009**

Table of contents

I. Executive Summary

This deliverables represents the results of the exploration and optimization of the multi-core architecture use case and its application, matrix multiplication.

The role of this deliverable is to provide a description of the status of the multi-core architecture simulator and its XML interface, and to give the first set of results about the integration of this use case in the overall Multicube design exploration flow. The multi-core architecture simulator has been integrated with open source Multicube explorer tool, based on the XML interface that has been developed according to the specification defined in D1.4.1, "Definition of the Specification of the Design Flow Integration". Some preliminary results of the design space exploration have also been automatically obtained by using the Multicube Explorer tool. These results represent the first concrete step for the "MultiCore Architecture Design and Optimization based on Design Space Exploration Flow" (Task 3.4).

This deliverable is organized in four sections. Section II describes an updated specification of the architecture model of the multi-core architecture, respect to those described in D.1.3 ("Definition of the specification for the industrial use cases"), issued at M6. The changes deal mainly with design space configurable parameters and metrics. The updates of the specification have also been reported in D4.1.1 ("Initial evaluation plan"). Section III provides a description of the XML interface implemented. Section IV describes the implementation of the use case architecture and simulation environment. Section V describes a preliminary design space exploration results obtained by the open source Multicube Explorer tool and provides a first design space analysis consideration. This activity can also be considered as part of the step-by-step validation phase of WP4.

A more comprehensive design space exploration of the use case will be presented in D3.4.2, "Final report on architectural exploration and optimization of MPSoC architectures".

II. Specification of Multi-Core Architecture Use Case

This section describes an updated specification of the architecture model of the multi-core architecture, respect to those described in D.1.3 "Definition of the specification for the industrial use cases", issued at M6. The changes deal mainly with design space configurable parameters and metrics. The updates of the specification have also been reported in D4.1.1 ("Initial evaluation plan").

Multi-core architectures address the physical challenges of power and wire delay by favoring several simpler cores over a large monolithic processor. Compared with heterogeneous multi-core design, a tiled homogeneous multi-core architecture is easier to design, verify and program.

Tiled multi-core processors further arrange the abundant on-chip resources – including logic, wires, and pins – in a scalable tiled pattern. Tiled multi-core architectures replace global-access centralized structures (such as giant register files, buses, and centralized caches), where wire delay and power efficiency scale poorly, with small distributed structures (such as mesh-based on-chip networks) that facilitate efficient local accesses. The tiled multi-core architecture methodology is equally effective at improving power efficiency because it replaces a large monolithic core by several smaller voltage-scaled cores.

To simplify the design and verification progress, we specified this design 'Transformer' as a mesh based homogenous multi-core processor.

II.1. Science Computation Application (Matrix Multiplication)

Matrix multiplications are common in applications of science computation. For a large $N*N$ square matrix, all the elements can not be hold in the local storage, such as cache, so large matrixes are always divided into small $M*M$ sub-matrixes in multiplication for locality. For example, the source matrixes A/B and result matrix C are all $N*N$ square matrix, and they are divided into a series of $M*M$ sub-matrixes, where $M = N/p$. So a sub-matrix of C can be gotten with the formula below.

$$C_{i,j} = \sum_{k=1}^{P} A_{i,k} * B_{k,j}$$

As shown in Figure 1, $C = A*B$, p is 4, $A/B/C$ are all divided into 16 sub-matrixes, and C_{12} sub-matrix can be gotten with the formula below.

$$C_{12} = \sum_{k=0}^{3} A_{1,k} * B_{k,2}$$

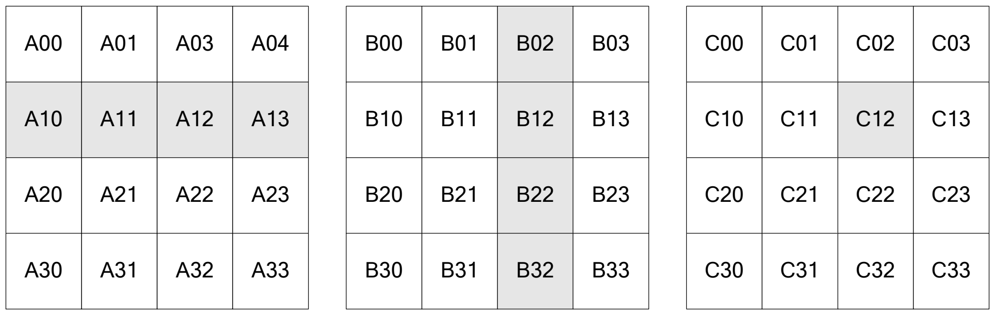


Figure 1 Matrix multiplication

There is no RAW (read after write) dependency between different sub-matrixes of C, and A/B introduce no dependency as they are read-only matrixes. For this reason, different sub-matrixes of C can be calculated in different threads, or even on different cores in a multi-core system. In multi-core system, as each core are free of dependency, they only need synchronization at the end of calculation for collecting the sub-matrixes of C. The workloads for each thread or each core are the same, so it is natural for them to reach the synchronization point. As shown in Figure 2, the synchronization can also be done in parallel. Threads with number that can be divided evenly by 2 collect the result from the threads numbered next to them, and then threads with number that can be divided evenly by 4 collect the result from the threads which number can be divided evenly by 2 but can not be divided evenly by 4. At last, all the results are hold in the storage of thread0. It costs time when establish connections between different cores in a multi-core system. This policy saves time by reducing the connections to thread0, although the data transferred to thread0 are not changed at all.

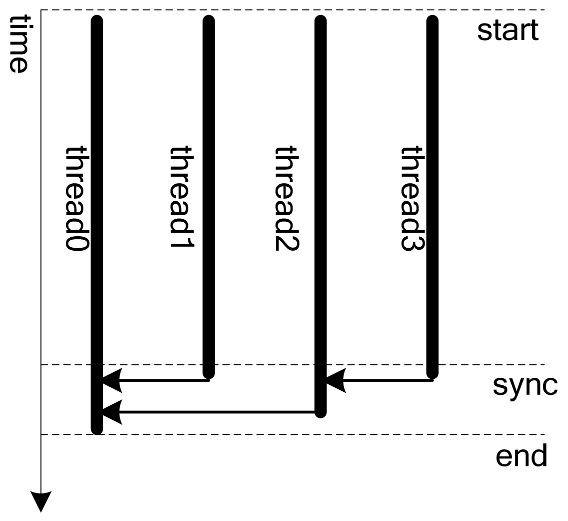


Figure 2 Computing flow

Figure 3 compares optimized synchronization with normal synchronization for 8 threads (cores). This optimized policy reduced the time complexity of synchronization form $O(n)$ to $O(\lg^n)$.

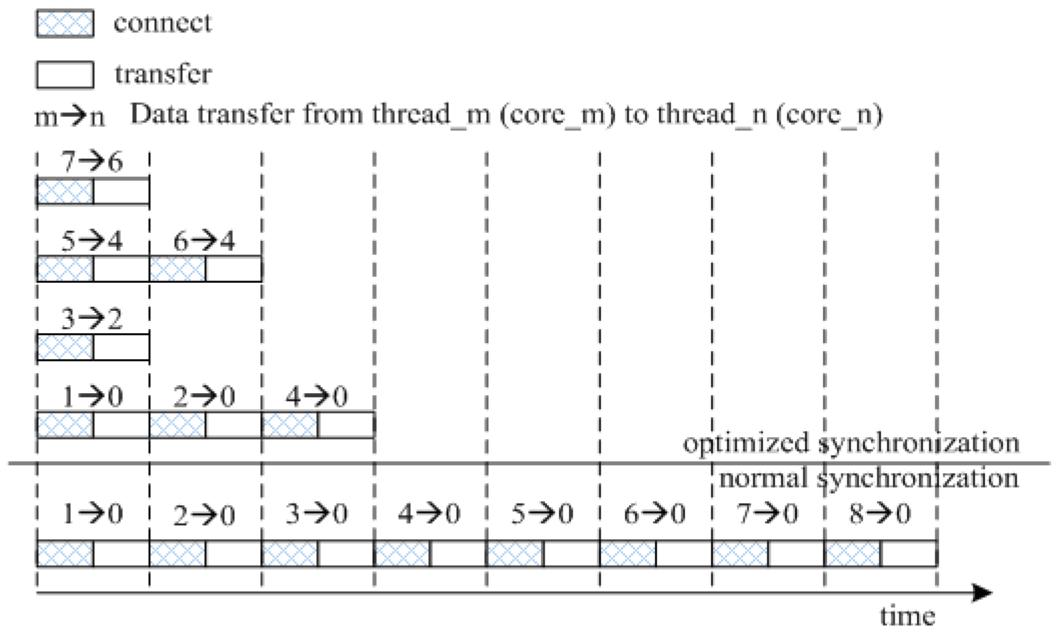

connect
transfer
m→n Data transfer from thread_m (core_m) to thread_n (core_n)
7→6
5→4
6→4
3→2
1→0
2→0
4→0
optimized synchronization
normal synchronization
1→0
2→0
3→0
4→0
5→0
6→0
7→0
8→0
time

Figure 3 Optimized synchronization

From the view of a given level of the memory hierarchy, matrix multiplication is a computationally intensive science computation. For the two source square matrixes of order N, N^3 multiplications and N^3 adds are needed, and these $2N^3$ calculations can also be treated as N^3 MAC (Multiply and Accumulate) operations. Loading these two source matrixes needs to load $2N^2$ elements, then the ratio of computation and memory access is $N/2$. So we can see that matrix multiplication is rich of locality and computationally intensive.

II.2. Processor Architecture

The Transformer Processor is a tiled multi-core architecture. A tiled multi-core architecture is a multiple-instruction, multiple-data (MIMD) machine consisting of a 2D grid of homogeneous, general-purpose compute elements, called cores or tiles. Instead of using buses or rings to connect the many on-chip cores, the Transformer Architecture couples its processors using 2D mesh networks, which provide the transport medium for off-chip memory access and other communication activity.

By using mesh networks, the Transformer Architecture can support anywhere from a few to many processors without modifications to the communication fabric. In fact, the amount of in-core (tile) communications infrastructure remains constant as the number of cores grows. Although the in-core resources do not grow as tiles are added, the bandwidth connecting the cores grows with the number of cores.

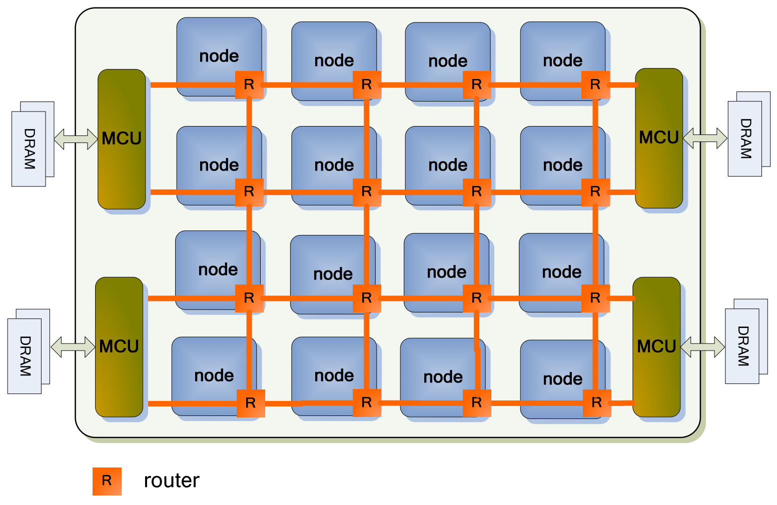


Figure 4 Architecture of transformer

As shown in Figure 4, the Transformer Processor Architecture consists of a 4*4 2D grid of identical compute elements, called nodes. Each node is a powerful computing system that can independently run an entire application.

As Figure 29 shows, the perimeters of the mesh networks in a Transformer Processor connect to memory controllers, which in turn connect to the respective off-chip DRAMs through the chip's pins. Each node combines a processor and its associated cache hierarchy with a router, which implements the interconnection network. Specifically, each node implements a two-issue out-of-order pipelined processor architecture with an independent program counter and a two-level cache hierarchy.

II.2.1. Node Architecture

There is separated level1 instruction/data cache and unified level2 cache in each node. The node is connected to the associated router through coprocessor2.

II.2.2. Instruction Set

As there is need for 64-bit computing in multimedia applications, 64-bit MIPS-III ISA is chosen for the node.

Some MAC (Multiply-Accumulate) instructions which need three source operands are also supported by node.

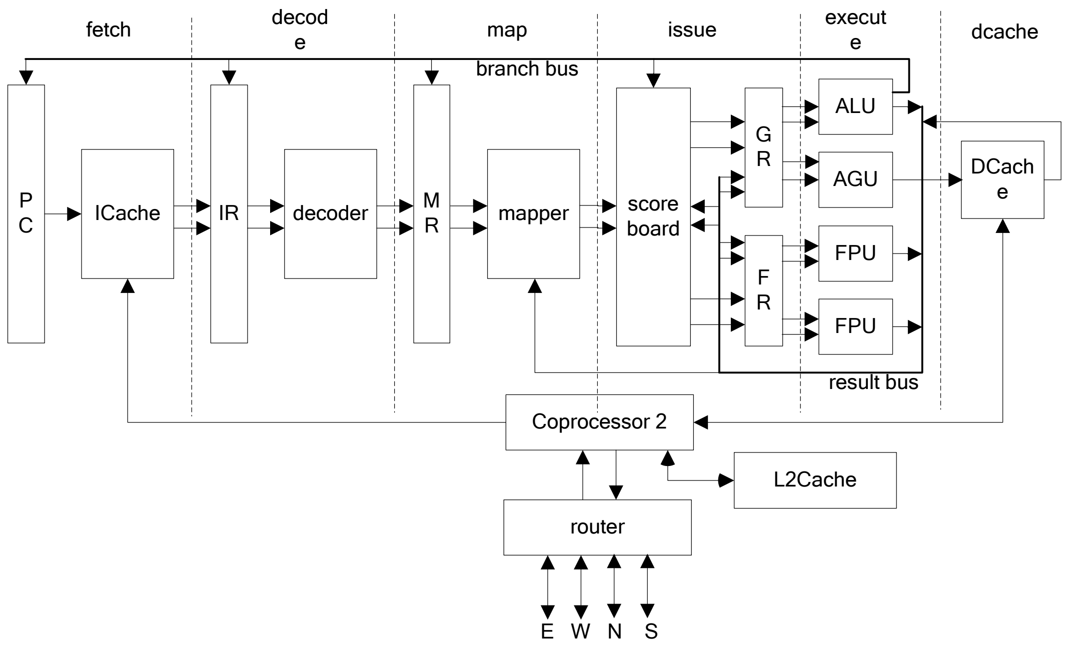


Figure 5 Architecture of node

II.2.3. Pipeline

Each node implements a two-issue out-of-order pipelined processor architecture, as shown in

Figure 5. The pipeline is divided into 6 stages: fetch, decode, map, issue, execute and write back. For memory operations, there is one more stage 'data cache' between 'execute' and 'write back'. Short names used in

Figure 5 are described in Table 1.

In order to achieve reasonable performance and implement the node in a simple way, we use a junior out-of-order pipeline based on scoreboard. The job of 'map' stage is to inspect the dependencies between the output instructions of 'decode' stage and the instructions already in the scoreboard, or even between the output instructions of 'decode' stage. Then the mapper sends the instructions without unknown dependency.

The scoreboard unit in each node is responsible for accepting decoded instructions from the 'map' stage, and issuing them to the functional units (address generators, ALUs and FPUs) satisfying dependencies among the instructions. To achieve this goal the main element of the scoreboard is the instruction queue which holds decoded instructions and issues them once the resources they require are free and their dependencies have cleared.

Table 1 Details of node

Short name	**Description**
PC	Program Counter
ICache	Instruction cache
IR	Instruction register
MR	Mapper register
GR	General purpose register file
FR	Floating point register file
ALU	Fixed point function unit
AGU	Address generating function unit
FPU	Floating point function unit
DCache	Data Cache
L2Cache	Level 2 cache
E/W/N/S	East/West/North/South

Explicit branch predicting strategy is used for now. In general, backward branches are predicted as taken, and forward branches are predicted as not taken. When the head entry of instruction queue in the scoreboard contains a written back branch instruction, this instruction is checked for the accuracy of prediction. If a wrong prediction has been made, the instructions after this branch instruction in the pipeline will all be flushed, and proper instructions are fetched from the right direction.

Details of the pipeline can be seen from Table 2.

Table 2 Parameters of Pipeline

Parameter	**Default Values**
Instruction cache	8KB, 2way, 32B/line, 4bank/way, bandwidth:8B/cycle, single port
Data cache	8KB, 2way, 32B/line, 4bank/way, bandwidth:16B/cycle, single port
Decoder	2instruction/cycle
Instruction buffer	6-instruction
Mapper	2instruction/cycle
Scoreboard	16-entry instruction queue, 2-issue, 2-commit
Register	GPR: general purpose register file, 32entry, 32bit FR: vector register file, 32entry, 64bit
Function Unit	1ALU: integer/branch 1AGU: address generating unit 2FPU: 64bit floating point function unit

II.2.4. Register

As there is no register renaming in node pipeline, the entry number of physical register file is same to the number of the architectural registers. Each node has a 32-entry 64-bit GR (general purpose register file) and a 32-entry 64bit FR (floating point register file).

For the GR, a normal ALU instruction needs two operands and a normal AGU instruction needs two operands ,one as base address to calculate the memory address and the other as the value to be stored for store operation. So GR should provide 4 read ports for the situation that two instructions were issued to ALU and AGU separately in one cycle. GR can only accept data from instruction queue in scoreboard when it commit instructions, so it should provide 2 write ports for the situation that the head two instructions in instruction queue can be committed in one cycle and they both have results to be written to GR.

The GR has 4 read ports and 2 write ports, as shown in Figure 6.

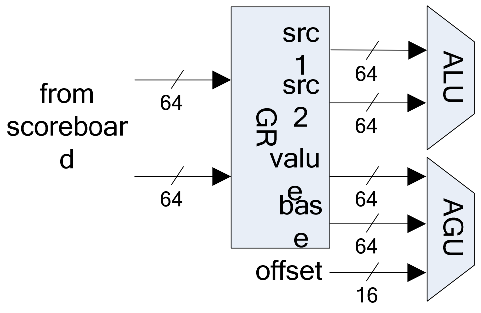


Figure 6 GR organization

For the FR, a normal floating point instruction needs two to three (MAC) operands. With a 2-issue pipeline, FR should provide 6 read ports for the situation that two MAC instructions were issued to the two FPUs separately in one cycle. FR can only accept data from instruction queue in scoreboard when it commit instructions, so it should provide 2 write ports for the situation that the head two instructions in instruction queue can be committed in one cycle and they both have results to be written to FR.

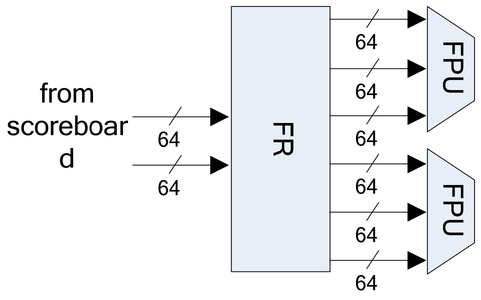


Figure 7 FR organization

II.2.5. Interface

Interface works as an important module which controls the data and commands flowing in and out the node. As the extended MIPS-III ISA meets all the requirements of science computation applications, we use the COP2 interface for the nodes to exchange data between level2 caches and interface registers are implemented as COP2 registers. MIPS-III ISA reserves 32 data register and 32 control register for COP2, and associated instructions reserved by MIPS-III ISA for COP2 are listed in Table 3.

Table 3 COP2 instruction

Instruction	Description
LWC2	Load 32bit value from memory to COP2 data register
SWC2	Store 32bit value from COP2 data register to memory
MTC2	Move 32bit value from GR to COP2 data register
MFC2	Move 32bit value from COP2 data register to GR
CTC2	Move 32bit value from GR to COP2 control register
CFC2	Move 32bit value from COP2 control register to GR

The interface is controlled by modifying these interface registers with instructions reserved for COP2.

II.2.6. On-chip Network

The Transformer Architecture's interconnect provides communication via direct user-accessible communication networks. The direct user accessible communication networks allow for scalar operands, streams of data, and messages to be passed between nodes.

Taking advantage of the huge amount of bandwidth afforded by the on-chip integration of multiple mesh networks requires new programming APIs and a tuned software runtime system. We build a library which provides primitives for streaming and messaging, much like a lightweight form of the familiar MPI.

II.2.6.a. Mesh Network

Mesh network connects five directions: north, south, east, west, and to the node. Each link consists of two 32-bit-wide unidirectional links; thus, traffic on a link can flow in both directions at once.

Each node uses a fully connected crossbar, which allows all-to-all five-way communication. Figure 8 shows a single router in detail.

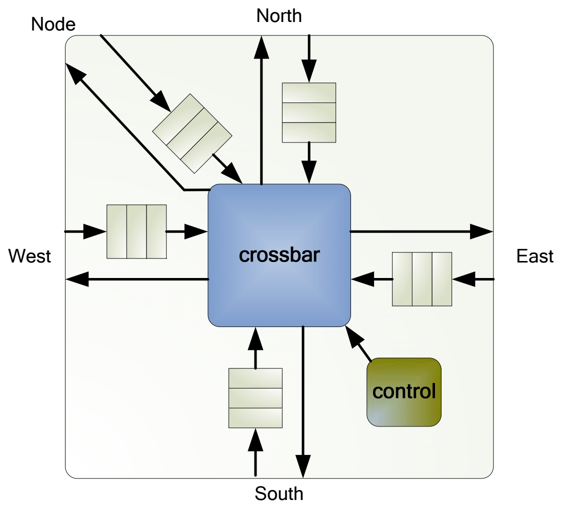


Figure 8 Architecture of router

The network provides a packetized, fire-and-forget message interface. Each packet is treated as a message and can be divided into a group of 32bit-long flits, containing a head flit, a series of body flits, and a tail flit. The message can even contain only one single flit. The head flit contains information denoting the x and y destination location for the message along with the message's length, up to 32 words per message.

The network is x-y dimension-ordered and wormhole-routed. The latency of each hop through the network is one cycle when message is going straight, and one extra cycle for route calculation when a packet must make a turn at a switch. Because the networks are wormhole-routed, they use minimal in-network buffering. In fact, the network's only buffering comes from small, three-entry FIFO buffers that serve only to cover the link-level flow control cost. The network preserves ordering of messages between any two nodes, but do not guarantee ordering between sets of nodes. A message is considered to be atomic and is guaranteed not to be interrupted at the receiving node. The dynamic networks are flow controlled and guarantee reliable delivery.

II.2.6.b. Message

With message passing, it is up to the software instead of hardware to maintain data consistency. This relaxes the hardware designer but exposes the details of data consistency to the software designer. To message passing strategy, result data from a producer node should be sent to the consumer node via on-chip network. It is the duty of on-chip network to maintain sequence of the message, we choose router based mesh as on-chip network, and select x-y dimension order routing because it is free of deadlock.

We previously described that each link consists of two 32-bit-wide unidirectional links. Actually some assistant lines are also needed to indicate the validation and type of the flit passing on the link, as shown in Figure 9.

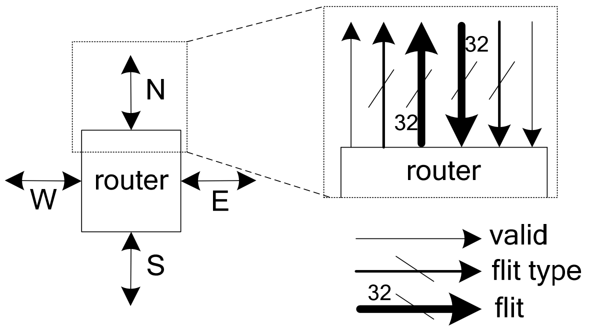


Figure 9 Link in details

Table 4 lists the flit types, and the construct of head flit can be seen in Figure 10. The 32bit head flit contains 6 fields, in which (*dst_x*, *dst_y*) indicates the destination node, (*src_x*, *src_y*) indicates the source node, *msg_type* indicates the type of this message and *msg_len* indicates the length of this message.

Table 4 Flit types

Flit Type	Description
Single	The message has only one flit.
Head	The head flit in a series of flits which compose a message.
Tail	The head flit in a series of flits which compose a message.
Body	The flits between Head flit and Tail flit in a series of flits which compose a message.

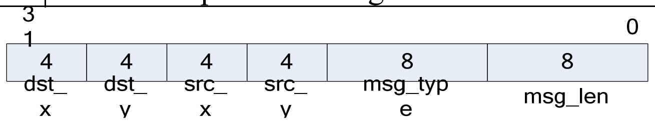


Figure 10 Head flit architecture

Message types are shown in Table 5. We use producer buffered message passing strategy in Transformer. There is a *request_mask* register in each node, and the length of this register is equal to the number of nodes. When a node acts as a consumer, it send *acquire* message to the producer. After the producer receives this request, the corresponding bit in producer's *request_mask* register is set to '1'. When the result data produced by a producer node are ready, the producer looks up its *request_mask* register. If the value '1' is found in the corresponding bit, a response message is sent to the consumer node by the producer, and the corresponding bit in producer's *request_mask* is reset to '0'. If the producer comes to the synchronizing point earlier than the consumer, it stores the message in its own storage and goes on running. When the *acquire* message comes form the consumer, a user level interrupt is triggered and the message is then sent by the user level interrupt handler. The user level handler will be discussed later.

Table 5 Message types

Type	Description
Acquire	The consumer node send acquire message to the producer.
Response	The producer node send response message to the consumer.
Note	The message that need no response

II.2.7. Memory System

II.2.7.a. Segmented Memory

As the number of nodes grows, it is not possible for the system to provide a separate physical memory and associated MCU (Memory Control Unit). So a natural implementation is to share a MCU between several nodes.

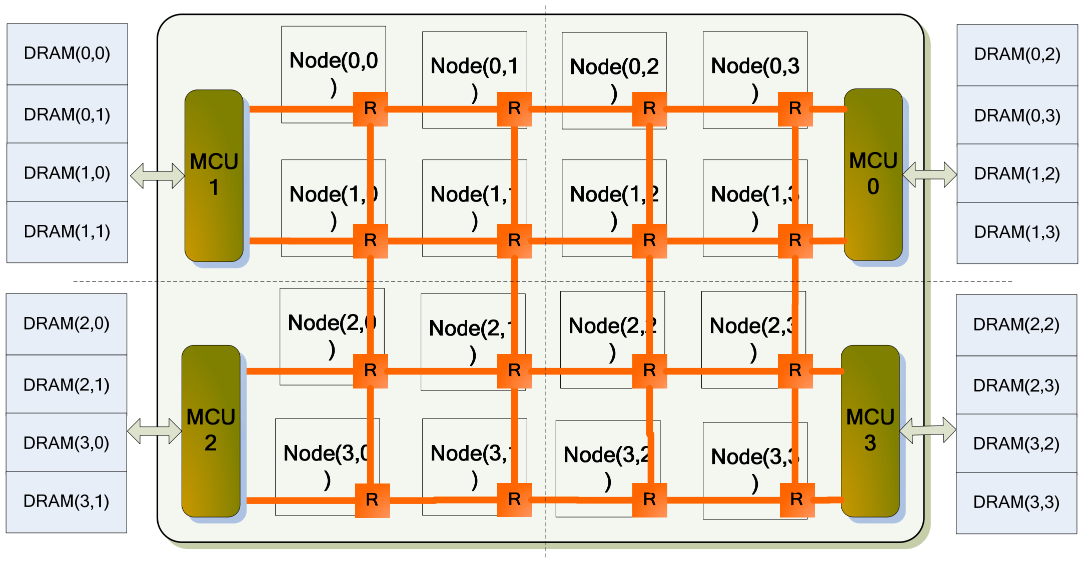


Figure 11 Segmented physical memory

As shown in

Figure 11, 16 nodes share 4 MCUs and each MCU serves 4 nodes close to it by dealing with memory access requests come from these nodes. As nodes do not share data in memory, each node occupies a segment of physical memory. For a 2GB memory controlled by a MCU, it is separated into 4 512MB segments. Each segment is the whole memory a corresponding node can access.16 nodes are divided into 4

When a node accesses its associated memory segment, the ‘physical’ address provided by the node can not be used by the MCU directly. This ‘physical’ address should be prefixed with the segment number, and then the MCU use this prefixed address for real memory access.

II.2.7.b. Multi-level Storage

Each node has 8KB level1 instruction cache, 8KB level1 data cache and 64KB unified level2 cache. Level1 data cache and level2 cache both obey write-back policy. As there is no need to maintain cache coherence between different nodes, the relationship between level1 cache and level2 cache is neither strictly inclusive nor strictly exclusive.

When a message is transferred between two nodes through UNet, there is no cache coherence problem for the receiver node as the received value is directly used as an immediate number.

MIPS-III ISA provides enough dedicated instructions for cache operations.

II.2.8. Programming on Transformer

II.2.8.a. Programming Model

The Transformer platform is much like an on-chip cluster with user controlled dynamic on-chip network. The programming model is similar to MPI (Message Passing Interface), but the implementation of APIs is based on user controlled NoC, which make the message passing process more efficient.

Every node executes the same program and each chooses a different part in this program according to its position in the mesh network, and each part of this program can be implemented as a function. In such mode, it is convenient for the programmer to development a parallel program and to map it onto Transformer platform.

II.2.8.b. Software Tool

For now, programs should be written in C programming language with the APIs supported by Transformer platform. As the APIs are implemented by embedding MIPS assemble language in C, these APIs can be directly invoked in C source code. A standard C compiler like GCC can be used here, which can be easily gotten. The main APIs provided by Transformer platform are shown in Table 6.

Table 6 Main APIs

API	**Description**
int tf_get_position(&x, &y);	Get position information of current core in the mesh network. The row position is stored in *x* and the column position is stored in *y*.
void tf_send(char* buf, int size, int dst_x, int dst_y);	Used by producer to send data to consumer. *Buf* stands for the address of the first byte in a series of data to be sent, *size* stands the size of this dataset, and (*dst_x*, *dst_y*) indicates position of the destination node. It appears in pair with tf_receivce() function.
void tf_receive(char* buf, int size, int dst_x, int dst_y);	Used by consumer to receive data from producer. *Buf* stands for the address of the first byte where the received data will be stored, *size* stands the size of this dataset, and (*dst_x*, *dst_y*) indicates position of the source node. It appears in pair with tf_send() function.

For *tf_send()* and *tf_receive()*, when *size* is less than or equal to 32 words, this pair of APIs will be translated into a message transfer process. But when *size* is more than 32 words, this pair of APIs will be translated into a large dataset transfer process.

After the program has been loaded, each node executes *tf_get_position()* API to find the accurate entry point where it should begin. The entry points for all the nodes are decided by the programmer.

II.3. Design Space Associated

This specification is the beginning of Transformer's design and it had better be convenient for further tuning. This is because the raw design may be proved to be inefficient after being tested with standard multimedia applications.

One goal of the design of Transformer is to make a reconfigurable platform, allowing further tuning as the evaluation works goes on. According to the system metrics, the designers can tune the hardware arguments for higher performance.

In the simulator, all the tunable arguments are collected in a configuration file in XML style. When the simulator is started, it read the arguments from this configuration file. The configuration parameters are listed in Table 7; this table represents an update with respect to the corresponding table reported in D1.3, "Definition of the specification of industrial use cases". The changes of the configurable parameters are part of the specification update reported in D4.1.1, "Initial evaluation plan".

Table 7 Configurable parameters

Parameter	Description
Mesh_order	Size of mesh
Cache_block_size	Size of cache block
ICache_ways	The number of instruction cache ways
Icache_entries	The number of entries in one way of instruction cache
DCache_ways	The number of data cache ways
DCache_entries	The number of entries in one way of data cache
L2Cache_ways	The number of level2 cache ways
L2Cache_entries	The number of entries in one way of level2 cache
L2Cache_access_latency	The latency of level2 cache
Memory_size	Size of system memory
Memory_access_latency	The latency of system memory
Router_buffer_entries	The number of router's input buffer entries

The configurable parameters are classified in two categories as follows:

- Design space of the mini-core
    - Icache_ways/dcache_ways/L2cache_ways indicates how many ways there are in the icache/dcache/L2cache. L2cache_ways should not be less than icache_ways+dcache_ways. The simulator supports the choices for 2, 4, 8, 16 for icache and dcache, and 32 is another choice for L2cache.
    - Icache_entries/dcache_entries/L2cache_entries indicates how many entries there are in each way of the icache/dcache/L2cache. The simulator supports the choices for 128, 256, 512.
    - Cache_block_size indicates how many bytes there are in each block of the icache/dcache/L2cache. It can be 32 or 64 bytes.

- Design space of the on-chip network
    - Mesh_order indicates the scale of the mesh style on-chip network. The simulator supports the choices for 2, 4, 8.
    - Memory_size indicates the physical memory space that can be used by one single mini-core. The simulator supports the choices for 32, 64, 128, 256.
    - Memory_access_latency indicates the latency of off-chip memory access.

II.4. Design Evaluation Metrics

When the simulator finished running, it write the system metrics into an output file, which can be used by the design space exploration tools. The system metrics reflects the performance of the platform under a particular configuration. So these metrics are important in tuning the configuration parameters.

The system metrics are listed in Table 8; this table represents an update with respect to the corresponding table reported in D1.3. The changes of the system metrics are part of the specification update reported in D4.1.1, "Initial evaluation plan".

Table 8 System metrics

Parameter	Description
Cycle	Number of cycles
Instructions	Number of instructions
Peak_power_dissipation	The peak power consumed by the architecture during the execution of the use case.
Power	The average power consumed by the architecture during the execution of the use case.
Area	The overall system area occupied by the architecture

System metrics are the following:

- Cycles indicates how many cycles the simulator has run.
- Instructions indicates how many instructions the whole multi-core processor has executed.
- Peak_power_dissipation indicates the peak power the multi-core processor exhausted.
- Power_dissipation indicates the average power dissipation of the multi-core processor.
- Area just means the area of the multi-core processor when it is translated into a chip.

III. XML interface description and implementation

The structure of the simulator can be seen in Figure 12. The simulator and XML input file both obey the rules described in design space definition file. XML input file is used to configure the simulator when it starts to run, and XML output file is generated by the simulator when it finish running. User can use "./simulator applications" to execute the simulator, and log files are stored in the same directory as the executable simulator file.

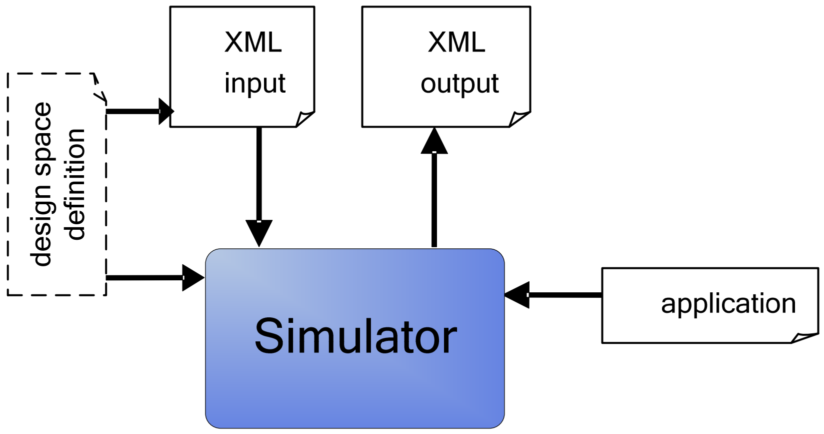


Figure 12 Multi-core use case interface

The simulator uses xml style files to describe its behaviors and architecture. As the multi-core structure is configurable, the rules of how to configure the simulator is describe in the design space definition file, as shown in Figure 13. The rules of how to evaluate the simulator are also included in this design space definition file.

```
<?xml version="1.0" encoding="UTF-8"?>
<design_space xmlns="http://www.multicube.eu/" version="1.3">
   <simulator>
      <simulator_executable path="/home/ST/bin/ict"/>
   </simulator>
   <parameters>
      <parameter name="mesh_order" description="Size of mesh"  min="2" max="8"
type="exp2" />
      <parameter name="cache_block_size" description="size of instruction cache block"
type="exp2" min="32" max="64"/>
      <!--unit="byte"-->
      <parameter name="icache_ways" description="number of instruction cache ways"
type="exp2" min="2" max="16"/>
      <parameter name="icache_entries" description="number of entries in one way of
instruction cache" type="exp2" min="128" max="512"/>
      <parameter name="dcache_ways" description="number of data cache ways" type="exp2"
min="2" max="16"/>
      <parameter name="dcache_entries" description="number of entries in one way of data
cache" type="exp2" min="128" max="512"/>
      <parameter name="L2cache_ways" description="number of level2 cache ways"
type="exp2" min="4" max="32"/>
      <parameter name="L2cache_entries" description="number of entries in one way of l2
cache" type="exp2" min="128" max="512"/>
      <parameter name="L2cache_access_latency" description="latency of level2 cache"
type="integer" min="3" max="10"/>
      <parameter name="memory_size" description="Size of system memory" type="exp2"
min="8" max="32"/>
      <!--unit="mega byte"-->
      <parameter name="memory_access_latency" description="latency of system memory"
type="integer" min="30" max="100"/>
      <parameter name="router_buffer_entries" description="number of router buffer entries of
one port" type="integer" min="2" max="8"/>
   </parameters>
   <rules>
      <rule>
         <less-equal>
            <expr operator="+">
               <parameter name="icache_ways"/>
               <parameter name="dcache_ways"/>
            </expr>
            <parameter name="L2cache_ways"/>
         </less-equal>
      </rule>

   </rules>
   <system_metrics>
      <system_metric name="cycles" type="integer" unit="cycles"/>
      <system_metric name="instructions" type="integer" unit="instructions"/>
      <system_metric name="peak_power_dissipation" type="float" unit="W"/>
      <system_metric name="power_dissipation" type="float" unit="W"/>
      <system_metric name="area" type="float" unit="mm2"/>
   </system_metrics>
</design_space>
```

Figure 13 Design space definition XML file

The simulator is written in C++, each kind of function unit of the multi-core platform is designed as a single class, and function units belongs to the same kind are implemented as objects of the corresponding class. These function units in the multi-core architecture are connected at initialization time of the simulator. At the same time, the simulator is configured with the input xml configuration file (in Figure 14 an example of the XML file is shown) which obeys the rules described in design space definition file.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator_input_interface xmlns="http://www.multicube.eu/" version="1.3">
<parameter name="mesh_order"             value="8"/>
<parameter name="memory_size"            value="64"/>
<parameter name="memory_access_latency"   value="30"/>
<parameter name="cache_block_size"        value="32"/>
      <parameter name="dcache_ways"            value="4"/>
<parameter name="dcache_entries"         value="128"/>
<parameter name="icache_ways"            value="4"/>
<parameter name="icache_entries"         value="128"/>
<parameter name="L2cache_ways"           value="8"/>
<parameter name="L2cache_entries"        value="256"/>
<parameter name="L2cache_access_latency"  value="3"/>
<parameter name="router_buffer_entries"      value="3"/>
</simulator_input_interface>
```

Figure 14 Example of input XML file

The cores are organized as Mesh in an extendable way. Data is transferred between each two cores as message, and programmer can handle this with MPI-Like programming interface.
With the debug interface, the user can run the simulator cycle by cycle, or by a number of cycle, or to a number of cycles. The user can also view the status of each concerned function unit and position of memory with the debug commands supported by the simulator. As the simulator is running, users can interrupt it with Ctrl-C command, view the status of the simulator and then start running from the interrupted place.
The concerned system metrics include cycles, executed instructions, cache performance, power and area. The information is collected as simulator running, and outputted in the style as defined by the system metric definition described in output xml file(Figure 15), when quit from the simulation.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator_output_interface xmlns="http://www.multicube.eu/" version="1.3">
   <system_metric name="cycles" value="22947"/>
   <system_metric name="instructions" value="1381568"/>
   <system_metric name="peak_power_dissipation" value="1288.24"/>
   <system_metric name="power_dissipation" value="511.732"/>
   <system_metric name="area" value="102.4"/>
</simulator_output_interface>
```

Figure 15 Example of output XML file

IV. Design and implementation of simulator environment

The simulator is implemented in C++ at a cycle-accurate level. An XML configuration file is used as input, and the system metrics are generated as output, which can be referenced by design space exploration tools. The input and output xml files are both defined under the rules described in the design space definition file. The simulator can be easily configured with different input xml files, and the metrics of each kind of configuration can be viewed as output xml file after running the simulator.

The simulator is implemented in a hierarchical design. As C++ is Object-Oriented language, each kind of unit of the many core design is designed as a single class, and all the units are implemented as objects according to the proper classes. First, base elements of the NoC architecture are formed. For example, each mini-core is made by function units, instruction cache, data cache, secondary cache, decode module and other logic. Other elements such as router and memory control unit are formed in the same way. Secondly, these base elements are connected together as a Mesh. Thus, the simulator can be configured flexibly.

IV.1. Speed and Accuracy

For a simulator, speed and accuracy are two topics which always conflict. We make a tradeoff by implementing several cores as C++ cycle-accurate modules and others' IPC are tuned according to the accurate ones. The NoC network and memory system are simulated in a cycle accurate way. As all the cores are homogeneous and computing tasks are equally mapped to each core, we can trust the whole simulator as cycle accurate.

IV.2. Event Driven Simulation

If events aren't guaranteed to occur at regular intervals, and we don't have a good bound on the time step (it shouldn't be so small as to make the simulation run too long, nor so large as to make the number of events unmanageable), then it's more appropriate to use an event-driven simulation to achieve cycle accurate simulation.

This approach uses a list of events that occur at various cycles, and handles them in order of increasing time. Each event in the list has a timing stamp, which is not less than the current system clock. During each cycle, the whole list is scanned and the event plans to occur at current cycle is triggered, and it is handled and removed from the list. This event may cause other events in current cycle or later, these new events are then added to the list. After scanning the event list, time stamp of the most recent event can be recorded, and the system time can "jump" to the time of this event.

As event driven simulator only care about the *changing* part of the system, it can keep running in high speed.

Here is a generic event-driven algorithm:

1. Initialize system state
2. Initialize event list
3. While (simulation not finished)
    a) Collect statistics from current state

b) Remove first event from list, handle it
c) Add new event to the list
d) Renew the time of system

Events are ordered by increasing time. We don't generate all the events in the list at the beginning. Instead we initialize the simulation with certain events, with their associated times. Later events can be inserted at the appropriate place in the event list by handling these certain events.

IV.3. Architecture of Source Code

Each kind of unit is designed as a single class, which is implemented as two separate parts, declaring in a *.h head file and implementing in a *.cpp source file. For example, instruction/data/secondary caches are all instantiated from *cache* class.

IV.4. Usage of the Simulator.

The simulator is developed with gcc, on Linux, and has no special dependence on the version of gcc or operating system.

A makefile is included in the code, the user can use *make* file to generate the executable file and run it with specified application. Before executing the simulator, the user can configure the simulator by modifying the *simulator_input_interface.xml* , and the metrics of the system can be checked in *simulator_output_interface.xml* after executing the simulator. More detail of how to use the simulator can be seen in *README* file.
For example, with a given input configuration file shown in Figure 16, and with matrix multiply as its application, the simulator generates metrics are shown in Figure 17 after execution.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator_input_interface xmlns="http://www.multicube.eu/"
version="1.3">
        <parameter name="mesh_order"             value="8"/>
        <parameter name="memory_size"            value="32"/>
        <!--units="mega"-->
        <parameter name="memory_access_latency"   value="30"/>
        <parameter name="cache_block_size"       value="32"/>
        <parameter name="dcache_ways"            value="4"/>
        <parameter name="dcache_entries"         value="128"/>
        <parameter name="icache_ways"            value="4"/>
        <parameter name="icache_entries"         value="128"/>
        <parameter name="L2cache_ways"           value="8"/>
        <parameter name="L2cache_entries"        value="256"/>
        <parameter name="L2cache_access_latency"  value="3"/>
        <parameter name="router_buffer_entries"      value="3"/>
</simulator_input_interface>
```

Figure 16 Example of input XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator_output_interface xmlns="http://www.multicube.eu/"
version="1.3">
            <system_metric name="cycles" value="400437"/>
            <system_metric name="instructions" value="22985248"/>
            <system_metric name="peak_power_dissipation" value="858.934"/>
            <system_metric name="power_dissipation" value="495.972"/>
            <system_metric name="area" value="102.4"/>
</simulator_output_interface>
```

Figure 17 Example of output XML file

IV.5. Power and Area Model of the Simulator

In our simulator, Princeton University's Wattch [1] power model is adopted (all the parameters used here are renewed according to 0.13μm process). The foundations for Wattch model infrastructure are parameterized power models of common structures present in modern microprocessors.

In CMOS microprocessors, dynamic power dissipation (P_d) is calculated with formula: $P_d = CV_{dd}^2af$. Here, C is the sum of all load capacitance, V_{dd} is the supply voltage, and f is the clock frequency. The activity factor, a, is a fraction between 0 and 1 indicating how often clock ticks lead to switching activity on average.

The main processor units that we model fall into four categories:

1. Array structures: caches, all register files, all buffers and so on.
2. Fully associative content-addressable memories: instruction window / reorder buffer wakeup logic, load/store order checks, for example.
3. Combinational logic and wires: function units, result buses and so on.
4. Clocking: clock buffer, clock wires, and capacitive loads.

In the result parameter, power_dissipation is the average of all cycles, the peak_power_dissipation is the maximum value of all cycles.

The area reported by the simulator is the sum of all units in the simulator, whose parameters are according to the Design Compiler's synthesis result.

V. Preliminary results from DSE flow

The preliminary design space exploration results presented in this report have been obtained by means of the Multicube Explorer tool. The exploration flow is composed of a randomized design of experiments over the design space of the use case (as specified in section II.3), where the L2Cache_access_latency and memory_access_latency are fixed respectively to 10 and 90 cycles and the memory size to 8 MB.

Given that, the feasible design space is composed of 14580 points. Assuming a simulation time of 1 minute, a complete design space exploration would have been required 243 simulation hours. The total number of random points simulated with the current version of the use case is 5.000 (corresponding to 34% of the complete design space).

V.1. Target application

The target application is the *Matrix Multiplication* as presented in Section II.1.

V.2. Objective functions

Before starting with the design space exploration, we selected a subset of the system metrics to be used as objective functions for the analysis. For this preliminary design space exploration, we selected the following, relevant system metrics:

- **Cycles.** This corresponds to the actual number of cycles of execution of the target application.
- **Power_dissipation.** This corresponds to the average power consumption of the architecture when executing the target application.

V.3. Objective space analysis

Figure 18 shows the scatter plot of the objective functions as estimated by the use case simulator on the configurations selected by the randomized design of experiment.

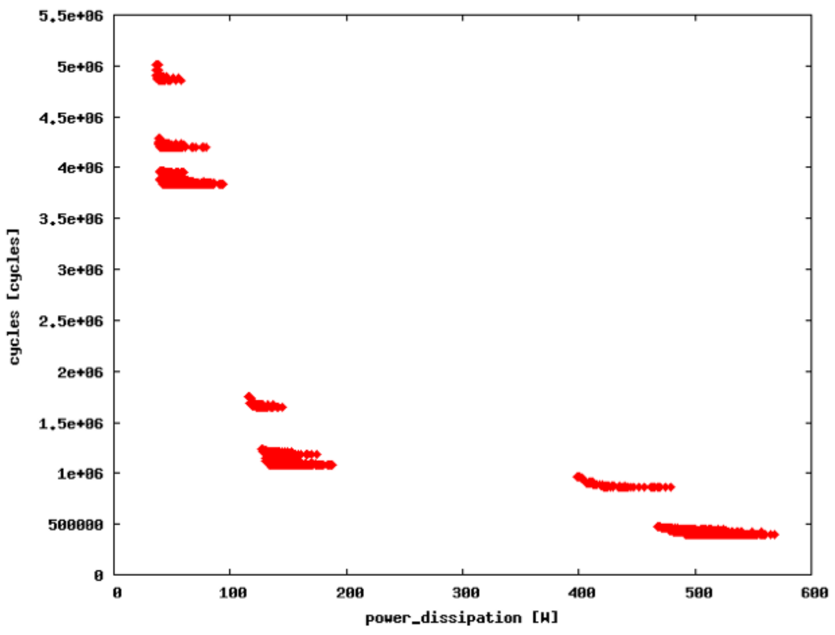


Figure 18 Cycle/power dissipation objective functions

As can be seen, the overall swing in terms of execution cycles ranges from 500K to 5M, while the power consumption varies from less than 50W up to 560W. This significant power consumption is associated with the current technology node selected (130nm) and the highest size of the multi-core chip, which, in our experimental design, is 64 cores.

The configurations are clearly clustered in 3 main groups. As the scatter plot in Figure 19 shows, the three groups are mainly related to the dimension of the mesh (which quadratically impacts the number of cores, e.g., a mesh order 8 corresponds to a configuration with 8 x 8 = 64 cores).

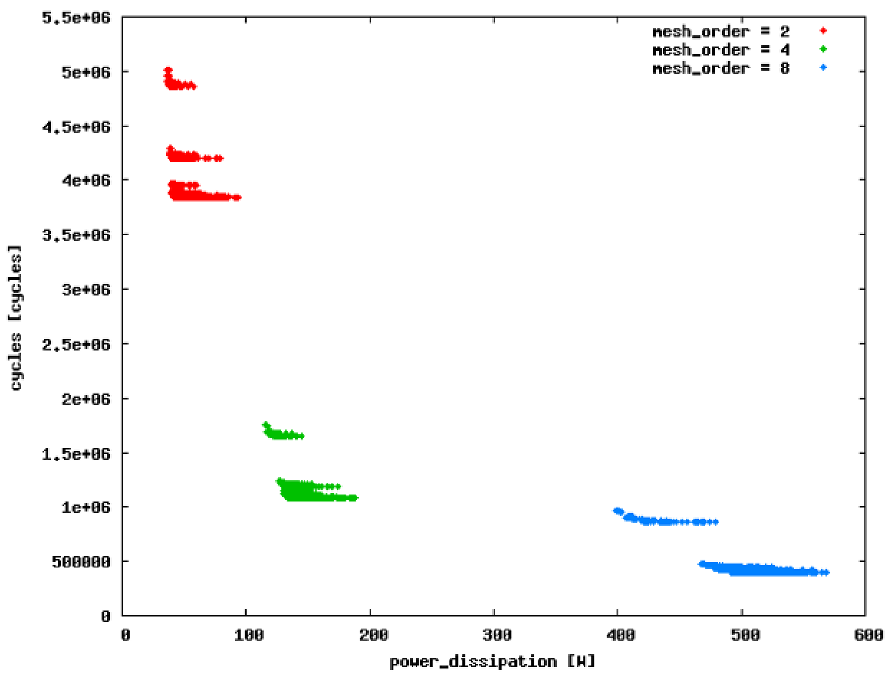


Figure 19 Cycle/power dissipation objective functions with mesh_order highlighted

As can be seen from the scatter plot in Figure 20, increasing the associativity of the data cache (from 2 to 16 ways) enables a reduction of the number of cycles within the same cluster but increases the spread of the power consumption (configurations with up to 16 cores). Besides, when the associativity is low (2 and 4 ways), there is a significant variance in terms of clock cycles.

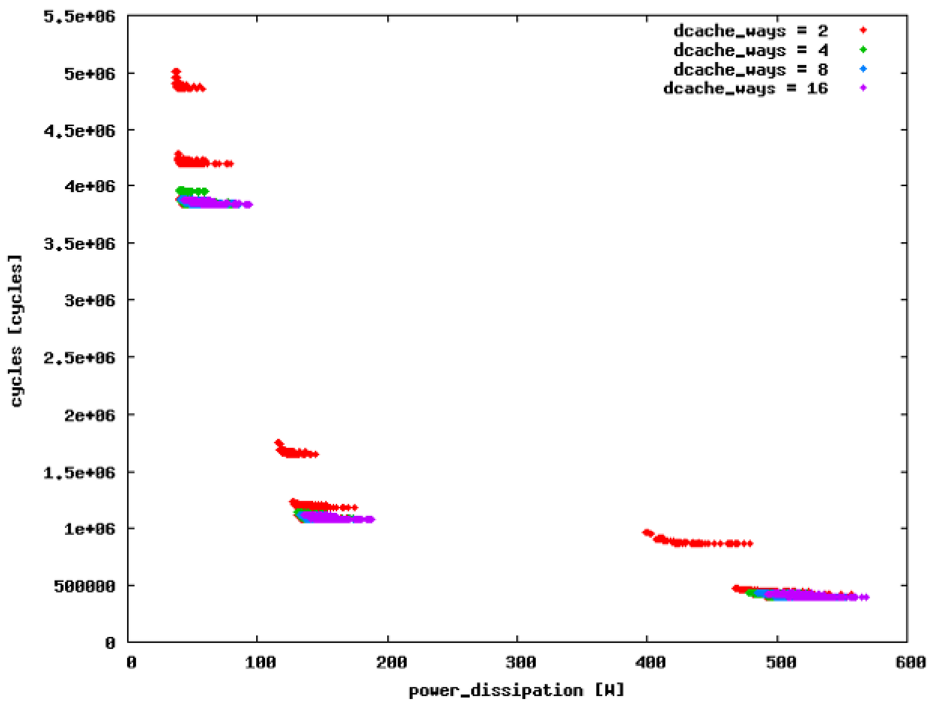


Figure 20 Cycle/power dissipation objective functions with dcache_ways highlighted

A similar behavior can be noted when varying the overall number of entries in the data cache (from 128 to 512), as shown in Figure 21.

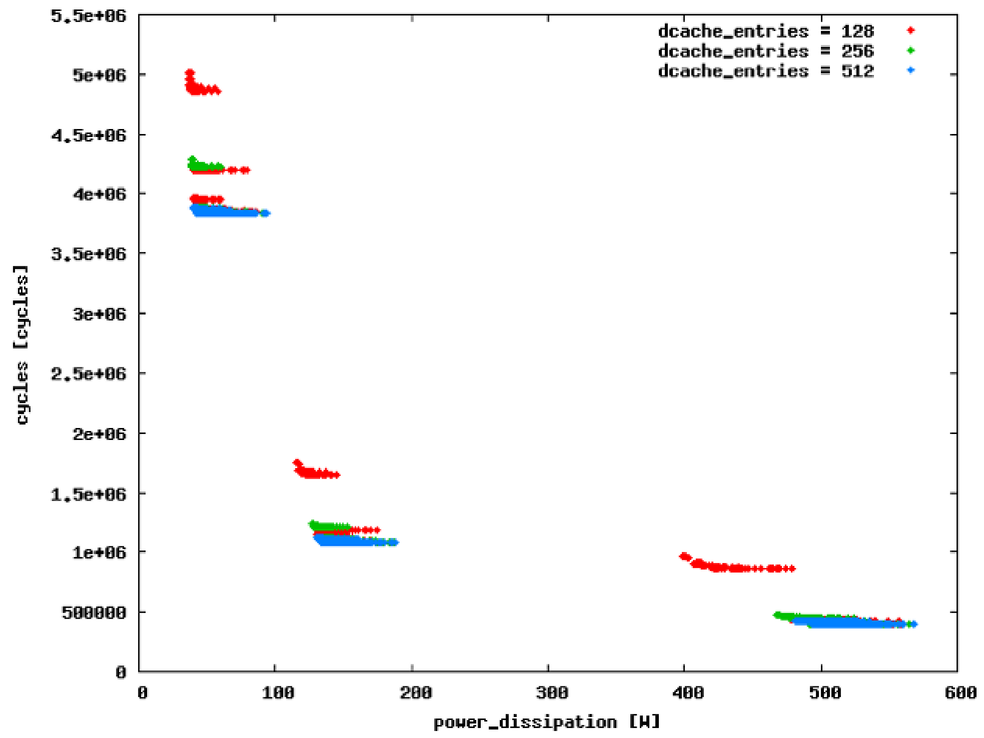


Figure 21 Cycle/power dissipation objective functions with dcache_entries highlighted

The overall impact of the parameters on the average value of the system level objective functions is shown in the **main effects** graphs in Figure 22 and Figure 23. The graphs are generated considering the variation of the objective function from the minimum to the maximum value of a single parameter (- and +) averaged over all the other parameters.

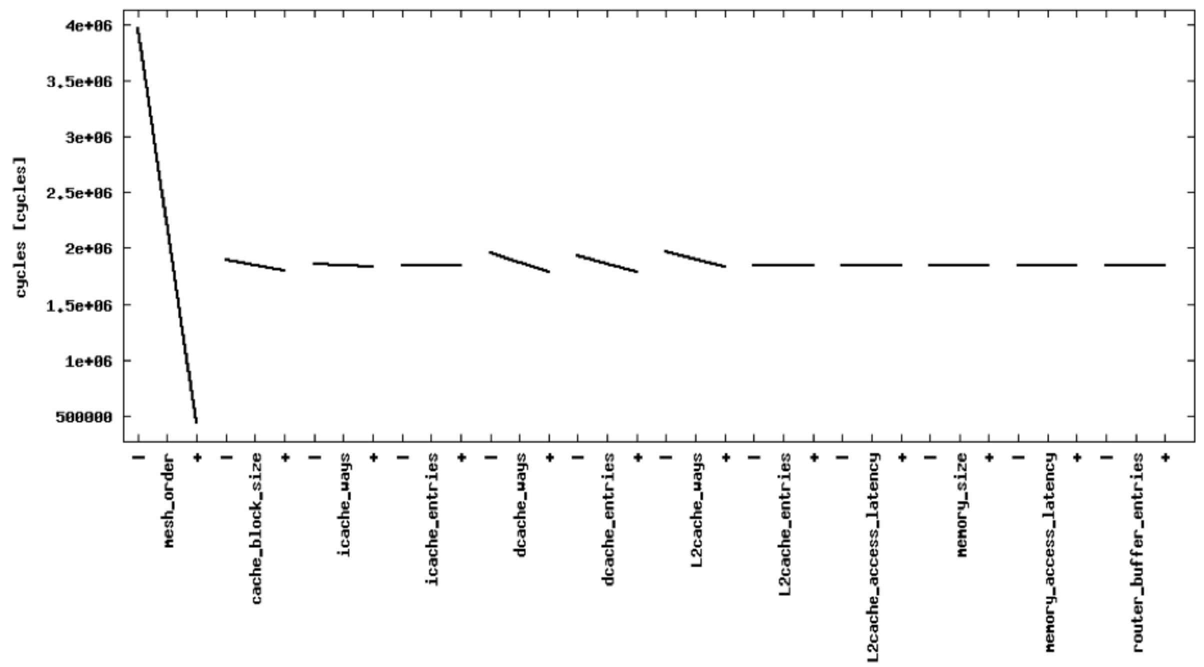


Figure 22 Parameter main effect analysis for the cycle objective function

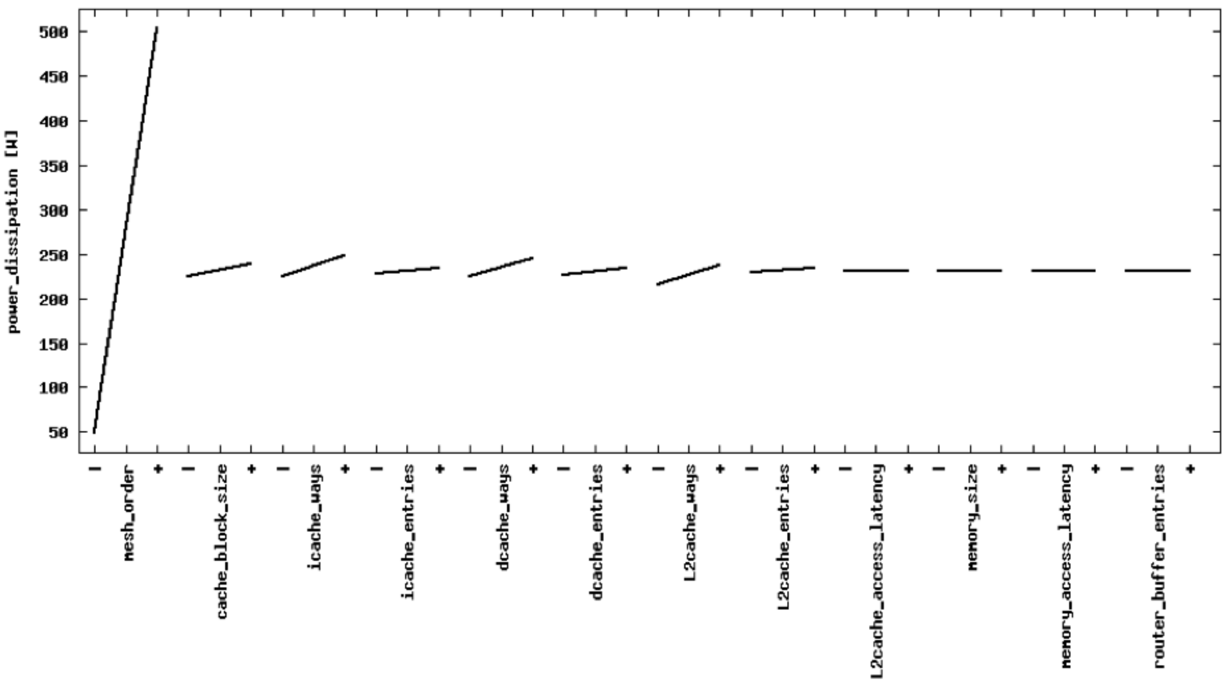
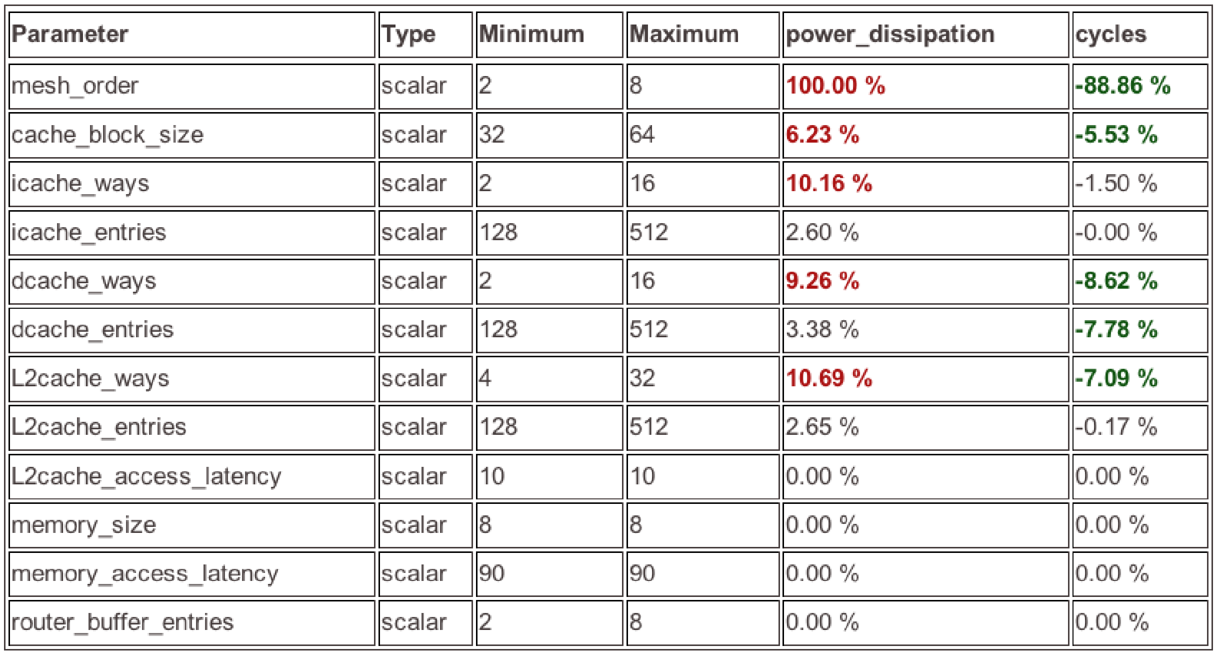


Figure 23 Parameter main effect analysis for the power dissipation objective function

We can definitely identify a strong dependence of the execution cycles and power dissipation with respect to the size of the mesh. The influence of the remaining set of parameters is relatively weaker and should be interpreted based on the variance on the system level objective functions as discussed previously.

Table 9 sums up the graphical significance analysis of the system level objective functions with respect to a positive variation of the system parameters:

Table 9 Summary of the significance analysis of the system level objective functions



Parameter	Type	Minimum	Maximum	power_dissipation	cycles
mesh_order	scalar	2	8	**100.00 %**	**-88.86 %**
cache_block_size	scalar	32	64	**6.23 %**	**-5.53 %**
icache_ways	scalar	2	16	**10.16 %**	-1.50 %
icache_entries	scalar	128	512	2.60 %	-0.00 %
dcache_ways	scalar	2	16	**9.26 %**	**-8.62 %**
dcache_entries	scalar	128	512	3.38 %	**-7.78 %**
L2cache_ways	scalar	4	32	**10.69 %**	**-7.09 %**
L2cache_entries	scalar	128	512	2.65 %	-0.17 %
L2cache_access_latency	scalar	10	10	0.00 %	0.00 %
memory_size	scalar	8	8	0.00 %	0.00 %
memory_access_latency	scalar	90	90	0.00 %	0.00 %
router_buffer_entries	scalar	2	8	0.00 %	0.00 %

We can note that the major impact on power dissipation is due to an increase of the mesh order, the cache block size, and the associativity of both instruction, data and level two cache. A positive variation of the previous parameters impacts positively the performance of the application. Besides, we can note that increasing the number of entries of the data cache has a positive influence too.

VI. References

[1] David Brooks, Vivek Tiwari, Margaret Martonosi, "Wattch: A Framework for Architectural Level Power Analysis and Optimizations" , in Proceedings of the 27th Annual International Symposium on Computer Architecture, pp.83-94, 2000.