Expert Implementation Guide for Fault-Tolerant Task Mapping (FTTM) on the Noxim Simulator

1. Introduction to FTTM and MNOC Architecture

This report details the systematic procedure necessary to install, modify, and implement the Fault-Tolerant Task Mapping (FTTM) algorithm and the Modified Network-on-Chip (MNOC) architecture within the SystemC-based Noxim simulator environment. The methodology adheres strictly to the framework outlined in the research titled, "Enhancing Reliability and Energy Efficiency in Many-Core Processors Through Fault-Tolerant Network-on-Chip".[1] The primary objective of the FTTM algorithm is to enhance system performance and reduce communication energy consumption, particularly when persistent hardware failures occur.[1]

The proposed solution fundamentally addresses permanent faults by optimizing task mapping both initially and dynamically during runtime. This process involves the introduction of a spare core technology and the optimization of the physical network topology itself.[1] Verification of this approach is conducted using the cycle-accurate Noxim simulator, demanding precise modification of its SystemC source files to model the novel architecture and algorithmic control mechanisms.[1]

1.1. Key Architectural Innovation: The Modified NoC (MNOC)

The MNOC platform is critical to the performance gains achieved by this research. A traditional 2-D Mesh Network-on-Chip utilizes routers equipped with five Input/Output (I/O) ports: four dedicated to neighboring routers (North, East, South, West) and one connected to the local processing core (PE).[1]

The MNOC architecture modifies this standard by proposing a **6-port router** configuration.

This is accomplished by ensuring that every core is connected to two routers, meaning that each router features two I/O ports connected to the local processing element.[1] This design choice increases the connectivity and redundancy within the network fabric. This dual connection minimizes communication energy and enhances performance by optimizing point-to-point data transmission through the routers, yielding significant improvements over the traditional 2-D NoC, especially when dealing with task mapping and scheduling.[1]

1.2. FTTM Algorithmic Overview

The FTTM approach governs task placement and failure recovery through two defined procedures: Task Mapping (Algorithm 1) and Spare Core Placement (Algorithm 2).[1] The mechanism is driven by a need to bypass faulty elements to maintain processor consistency.[1]

Initial task placement utilizes the **Optimal Distance (OD)** metric for selecting and optimizing the mapping region, followed by allocation based on minimizing the **Total Communication Energy (T_{CE})**.[1] When faults are encountered, the system first employs error correction codes (specifically, Hamming code) to detect and correct transient or intermittent faults. If the fault persists after this correction mechanism, it is definitively designated as a permanent core failure, initiating the necessary task migration process.[1]

2. Noxim Simulation Environment Setup and Prerequisites

The Noxim simulator, developed at the University of Catania, is implemented in SystemC, requiring a specific set of software dependencies and installation procedures within a Linux operating environment.[2]

2.1. Required Software Dependencies

A stable Linux distribution, such as Ubuntu or Fedora, is necessary, as automated setup scripts are available for these platforms.[4]

1. **C++ Toolchain and Libraries:** A modern C++ compiler (such as GCC) is required for building the simulator.[5] Additionally, Noxim uses YAML configuration files, necessitating the presence of the corresponding header files for successful compilation.[7]
2. **SystemC Library:** SystemC is the foundational component for Noxim.[2] SystemC must be installed prior to compiling the simulator. This process typically involves downloading the SystemC source (e.g., version 2.3.1) and performing a manual build.[8]
   - The installation procedure involves defining the installation environment variable (export SYSTEMC_HOME=/path/to/systemc), configuring the build process (./configure --prefix=$SYSTEMC_HOME), and executing the compilation and installation commands (make; sudo make install).[8] Successful compilation is predicated on ensuring all required C++ standard libraries and kernel headers are correctly linked, as SystemC installation can often encounter compilation issues related to missing or outdated includes.[5]

2.2. Acquiring and Installing the Noxim Simulator

The Noxim source code is most readily obtained using Git.[4]

1. **Source Acquisition:** The latest source can be cloned from the repository: git clone https://github.com/davidepatti/noxim.git.[4]
2. **Automated Setup:** For Ubuntu users, an automated script simplifies the dependency installation and compilation steps: bash <(wget -qO- --no-check-certificate https://raw.githubusercontent.com/davidepatti/noxim/master/other/setup/ubuntu.sh).[4]
3. **Manual Compilation:** If manual compilation is preferred, the user navigates to the source directory, potentially modifies the Makefile to enable debugging (DEBUG := -g -DDEBUG), and compiles using make clean; make.[10] Verification is achieved by running the executable against a default configuration file (./noxim -config../config_examples/default_config.yaml) to ensure readiness for modification.[10]

3. Architectural Implementation: Modified NoC (MNOC) Topology in SystemC

Replicating the research environment necessitates substantial modifications to the core SystemC files that define the Noxim architecture. This is essential to implement the unique

6-port MNOC router design.

3.1. Identifying Source Code Impact and Router Structure

The implementation of a custom topology, such as the MNOC, typically requires changes across several fundamental SystemC files that manage global parameters, data structures, and the network's connection logic.[11] These impacted files include the declarations and definitions for global settings (GlobalParams.h), data structures (DataStructs.h), and the router logic (router.cpp/h), as well as the network assembly logic (NoC.cpp/h).[11]

The primary structural change involves redefining the number of router ports from the standard 5 to 6.[11] The new router must logically accommodate two local ports ($LOCAL\_1$ and $LOCAL\_2$), in addition to the four cardinal ports.

Implementing the 6-port structure requires a significant update to the **arbitration unit** within the router.cpp source code. Standard Noxim arbitration is designed for 5 ports. The MNOC's increased port count introduces higher potential contention, especially since two ports connect to the same local core. Therefore, the router's arbitration scheme must be adapted to effectively manage resource access among the six inputs, potentially employing a modified round-robin or prioritized scheduling mechanism to handle the two local connections without becoming a communication bottleneck. This optimized arbitration logic is crucial for realizing the low-latency, energy-efficient communication benefits claimed by the MNOC.[1]

3.2. Topology Wiring and Modified Routing

The network construction logic, located primarily within the build_topology() function in NoC.cpp, must be altered to correctly instantiate and wire the MNOC architecture. For every router at coordinate (i, j), the topology generation function must establish **two distinct output channels** to the local processing element, thereby structurally embedding the MNOC redundancy.[1]

The simulation relies on a **Modified XY Routing** algorithm.[1] In standard deterministic XY routing, packets strictly traverse the X-dimension before the Y-dimension.[12] The modification required here serves two purposes: first, ensuring that the routing decisions correctly handle the presence of the two local ports on the 6-port router, and second, enabling fault-aware detouring. Since the FTTM architecture involves a "Manager Core" that handles task migration upon fault detection, the routing algorithm must be responsive to core failures, enabling packets to bypass permanently failed nodes when instructed.[1] This adaptive capability is necessary to maintain communication paths around faulty regions, overcoming the inherent limitation of deterministic XY routing's path diversity.[1]

4. Data Modeling and Input Generation using TGFF

To rigorously validate the FTTM algorithm, the simulation must process realistic application workloads—specifically, the task graphs (TGs) used in the research evaluation. These graphs are generated using the Task Graphs for Free (TGFF) tool.[1]

4.1. Installation and Parametric Generation

TGFF is a pseudorandom task graph generator that creates Directed Acyclic Graphs (DAGs) consisting of vertices (tasks, $V$) and weighted edges (communication links, $E$).[1]

1. **TGFF Acquisition and Installation:** The TGFF source code (e.g., version 3.6) should be downloaded and compiled using a standard make process: tar -xvzf tgff-3.6.tgz; cd tgff-3.6; make -j4.[14] The resulting executable must be added to the system PATH.[14]
2. **Benchmark Replication:** The research evaluates FTTM using both multimedia benchmarks (MPEG-4, VOPD, MWD, 263 dec/enc, Mp3dec) and synthetic benchmarks (G1, G2, G3, G4).[1] These applications range in size from 12 vertices (MPEG-4) up to 128 vertices (G4).[1]
3. **Reproducibility:** Although the exact TGFF input scripts are not provided, the TGFF tool allows parametric control (seed, tg_cnt, task_cnt, task_degree) to generate task sets whose structural properties (vertex and edge counts) align with the documented benchmarks.[15] Consistency in setting the seed is crucial for replicating the specific problem instances used for comparison.[15]

4.2. Conversion to Noxim Traffic Trace

The output generated by TGFF must be translated into an input format consumable by Noxim, typically a traffic trace or matrix. This conversion extracts two primary components: the communication weights (e_{ij}), which define the volume of data (flits) to be transmitted between tasks, and the task dependency constraints, which dictate the sequence and timing of packet injection into the NoC based on the initial FTTM mapping.[1] This ensures the simulation accurately models application-specific data flow rather than generic traffic patterns.

5. FTTM Algorithmic Implementation: Mapping and Optimization

The FTTM algorithm provides dynamic runtime control over task placement and reallocation. This control mechanism must be instantiated within the Noxim simulation environment through a dedicated SystemC module that acts as the centralized manager.

5.1. The Manager Core Module

The research identifies a "Manager core" responsible for overseeing the status of all cores (Free, Busy, Failed, Spare) and performing task migration in the event of a fault.[1] In Noxim, this translates into a dedicated SystemC module that executes the FTTM scheduling logic, replacing Noxim's default or simplistic traffic handling mechanisms. The Manager Core maintains the current mapping state and computes necessary metrics for optimization.

5.2. Implementation of Optimal Distance (OD) and Task Mapping (Algorithm 1)

Optimal Distance (OD) Calculation:
The Manager Core first determines the optimal mapping region by calculating the OD, which represents the shortest path between any two cores in the network.1 The calculation for a network of size $X \times Y$ is given by Equation 5:

$$\text{OD} = \frac{X+Y}{3}\left(1-\frac{1}{XY}\right)$$

Selecting the region corresponding to the minimum OD ensures that tasks are mapped to the most tightly interconnected and centrally available region, inherently minimizing communication latency prior to any traffic analysis.[1]

Task Mapping Execution:
The implemented logic for Algorithm 1 proceeds as follows:

1. Initialize all tasks (vertices V) as unmapped.
2. Select the optimal region based on the minimum OD calculation.
3. Identify the initial vertex (V_{i}) that exhibits the maximum communication with other vertices.
4. Map V_{i} to the core (C_{xy}) located at the center of the optimal region, specifically choosing the center core that communicates with the maximum number of free neighboring cores.[1]
5. Sequentially map remaining tasks by selecting the unmapped vertex with the next highest communication connectivity to already mapped vertices, placing it on a free core that has a direct connection to the central mapped core.[1]

Communication Energy Metrics:
The decision-making process is continually informed by minimizing communication energy (T_{CE}).1 The following equations must be implemented and utilized within the mapping cost function:

- The communication distance between two mapped vertices $V_i=(x_1, y_1)$ and $V_j=(x_2, y_2)$ is calculated using the Manhattan distance 1:

  $$|l_{V_iV_j}|=|x_1-x_2|+|y_1-y_2|$$
- The communication energy for a specific link (i, j) is the product of the communication rate (e_{ij}) and the communication distance 1:

  $$\text{CE}(V_i,V_j) = e_{ij} \times |l_{V_iV_j}|$$
- The total communication energy, which serves as the primary optimization objective during mapping, is the sum of energies for all links 1:

  $$T_{CE}=\sum_{\forall(i,j)}(e_{ij})\times C_{ij}$$

6. Fault Management and Dynamic Spare Core Allocation

The FTTM algorithm's defining feature is its ability to adapt to permanent core failures through dynamic spare core allocation.

6.1. Modeling Fault Injection and Persistence

The simulation must accurately model the fault condition. This involves programmatic injection of permanent faults (stuck-at faults) into cores, preferentially targeting those cores hosting high-communication vertices.[1]

The system must emulate the fault detection process: when a core reports an error, it is initially subjected to correction attempts (emulating Hamming code usage). A persistent error signal over a measured time window signifies that the transient/intermittent fault detection failed, confirming a permanent core failure. Upon registering a permanent fault at a core C_{fld}, the Manager Core triggers the spare core placement mechanism.[1]

6.2. Implementing Spare Core Placement (Algorithm 2)

Algorithm 2 executes the spare core recovery mechanism [1]:

1. **Identification:** All available free cores (C_{f}) are immediately designated as potential spare cores (P_{spare}).[1]
2. **Energy Evaluation:** For each task V_{i} that was mapped to the failed core, the Manager Core iterates through every available P_{spare}. For each candidate, the Manager Core calculates the projected T_{CE} for migrating V_{i} to that location. This calculation considers the increased latency and energy associated with the new communication path to all neighboring tasks.[1]
3. **Selection Criterion:** The selection strategy mandates choosing the spare core C_{xy} that results in the **minimum total communication energy** among all free core candidates.[1] This ensures that the spare core placement minimizes the energy overhead introduced by the migration, prioritizing proximity to the task's high-communication neighbors.[1]
4. **Migration and State Update:** Once selected, the task is migrated to C_{xy}, and the platform status is updated, marking the newly assigned core as Busy and the original core as Failed.[1]

6.3. Modeling Task Migration Cost

In faulty scenarios (One-Faulty Core, 1FC, or Two-Faulty Cores, 2FC), the simulation must account for the computational overhead incurred during recovery. This involves explicitly modeling the time required to transfer the tasks' internal state and data across the NoC from the failed core to the newly assigned spare core. This "migration time" is integrated into the simulation cycle count and is necessary to accurately capture the total processor execution time results presented in the paper.[1]

7. Simulation Execution, Parameter Tuning, and Verification

Accurate replication of the experimental results requires precise tuning of the Noxim configuration, defined through the config.yaml file.

7.1. Essential Simulation Parameters

The research utilized a cycle-accurate setup with specific resource definitions.[1]

Parameter	**Value**	**Context**
Simulator Type	Noxim, SystemC cycle-accurate	Baseline tool for evaluation [1]
Network Topology	5×5 to 10×10 Mesh/MNOC	Various network sizes used for testing [1]
Flit Length	16 flits	Basic unit of data transfer size [1]
Total Simulation Cycles	20,000 cycles	Overall simulation runtime [1]

Warm-up Cycles	10,000 cycles	Time to achieve network stability before metric collection [1]
Routing Algorithm	Modified XY Routing	Custom implementation required for fault handling [1]

7.2. Algorithmic Performance Comparison (Simulation Results)

The FTTM algorithm on the MNOC platform demonstrated significant quantitative advantages in terms of latency and communication energy compared to established fault-tolerant mapping algorithms, specifically FASA , FTDTMS , and FTNOC .[1] The following comparison, derived from simulation results, highlights the superior efficiency of the FTTM/MNOC approach.

Table I: Mapping Time and Latency Comparison (Simulation) [1]

Metric	****FASA ****	****FTDTMS ****	****FTNOC ****	**FTTM (2-D NoC)**	**FTTM (MNOC)**
Total Latency (clock cycles)	92467	91926	89984	86641	84592
Longest Path Latency (clock cycles)	30864	29986	28768	27192	26853
Mapping Time (ms)	124578	112624	108675	100262	99184

The FTTM algorithm implemented on the MNOC platform achieved the lowest total latency (84592 clock cycles) and the shortest mapping time (99184 ms) among all tested algorithms and architectures.[1] This reduction in mapping time is achieved because the algorithm, optimized by the OD metric, efficiently explores and selects optimal mapping regions, avoiding the lengthy search space typical of NP-hard mapping problems.

The corresponding communication energy consumption comparisons demonstrate similar advantages, particularly in faulty conditions.

Table II: Communication Energy (μJ) Comparison (Simulation) [1]

Algorithm	Zero Fault (0FC)	One Fault (1FC)
FASA	11300	15300
FTDTMS	14200	16200
FTNOC	18300	14300
FTTM (2-D NoC)	12500	14500
FTTM (MNOC)	9500	12500

Under zero-fault conditions (0FC), FTTM on MNOC achieved the lowest communication energy (9500 μJ).[1] Crucially, even with one faulty core (1FC), FTTM on MNOC maintained the lowest energy consumption (12500 μJ), demonstrating the effectiveness of the spare core placement logic (Algorithm 2) in selecting minimal energy replacement paths.[1] For multimedia benchmarks, FTTM on MNOC achieved an average communication energy conservation of 16.275% and performance improvement of 14.215% when compared to the FASA methodology.[1]

7.3. Performance under Faulty Conditions

The performance of FTTM is assessed by the total execution time, which includes task execution time, waiting time, and migration time.[1] Execution was analyzed across 0FC, 1FC, and Two-Faulty Core (2FC) scenarios.

Table VII: Benchmarks Execution Time (s) [1]

Benchm	FASA	FASA	FASA	FTTM (MNOC)	FTTM (MNOC)	FTTM (MNOC)

arks	(0FC)	(1FC)	(2FC)	(0FC)	(1FC)	(2FC)
MPEG4	1.42	12.8	66	1.02	8.32	48.26
VOPD	1.26	10.4	44.2	0.9	6.06	34.8
MWD	1.34	7.46	126.6	1	5.02	106.4
263 Dec	1.42	38.8	968.2	1.02	32.6	869.2
263Enc	1.16	3.82	38.4	0.68	2.92	29.4
Mp3dec	1.18	7.96	124.14	0.712	5.42	102.8

The data confirms that FTTM on MNOC consistently achieves lower execution times across all fault scenarios. For the 2FC scenario, the execution time reduction averages 22% compared to FASA, 28% compared to FTDTMS, 16% compared to FTNOC, and 14.2% compared to FTTM on the 2-D NoC.[1] This pronounced reduction in execution time under failure conditions validates the dynamic efficiency of the FTTM spare core allocation methodology.

7.4. Hardware Verification Metrics (FPGA)

The FTTM methodology was validated using hardware implementation on the Zynq UltraScale MPSoC ZCU104 Evaluation Kit.[1] Hardware metrics confirm the efficiency claims concerning resource utilization (Area) and power consumption.

Table XI: Evaluation of Area and On-Chip Power (FPGA Implementation) [1]

Algorithm	Area (LUTs) 0FC	Area (FFs) 0FC	Power (mW) 0FC	Area (LUTs) 2FC	Area (FFs) 2FC	Power (mW) 2FC
FASA	24572	26486	0.315	29856	34568	0.405

FTDTMS	24964	27868	0.326	31964	33654	0.43
FTNOC	25368	28986	0.334	28564	32984	0.39
FTTM on 2-D NoC	19846	20988	0.285	25486	28642	0.32
FTTM on MNOC	16548	18542	0.272	21468	23546	0.305

The FTTM algorithm on the MNOC platform exhibits superior area efficiency, utilizing the fewest Look-Up Tables (LUTs: 16548) and Flip Flops (FFs: 18542) under zero-fault conditions.[1] This superior efficiency holds true even when two faults are injected and spare cores are activated (LUTs: 21468; FFs: 23546).[1]

The on-chip power consumption is also minimized in the FTTM/MNOC configuration (0.272 mW at 0FC, 0.305 mW at 2FC).[1] Although the MNOC introduces a slight increase in hardware complexity by using dual local connections, the overall area penalty is negated by the efficiency of the FTTM algorithm in optimizing resource use and communication paths. This confirms that the combined FTTM/MNOC system achieves a highly favorable balance between fault tolerance capability and hardware cost, establishing its viability for energy-sensitive applications.[1]

8. Synthesis and Conclusion

The implementation of the Fault-Tolerant Task Mapping (FTTM) algorithm requires deep interaction with the Noxim simulator's SystemC source code. Successful replication depends upon three critical steps: establishing the SystemC environment and Noxim platform, structurally modifying the NoC architecture to model the 6-port MNOC, and integrating the dynamic FTTM logic via a dedicated Manager Core module.

The FTTM methodology, driven by the Optimal Distance metric and minimization of Total Communication Energy, provides a dynamic fault-tolerance solution capable of managing permanent core failures during runtime.[1] This dynamic approach, facilitated by the Manager Core's ability to execute Algorithm 2 for spare core placement, yields robust recovery capabilities, contrasting favorably with static mapping techniques.

Quantitative analysis consistently validates the system's efficacy. The FTTM algorithm, especially when paired with the MNOC platform, demonstrated clear advantages across key performance indicators: the lowest mapping time (99184 ms), minimal latency (84592 clock cycles), and superior communication energy conservation (9500 μJ at 0FC) when compared to three other contemporary fault-tolerant algorithms (FASA, FTDTMS, FTNOC).[1] Furthermore, hardware validation confirmed that these performance gains are achieved alongside excellent area utilization (minimal LUTs/FFs) and low on-chip power consumption, overcoming the anticipated overhead associated with implementing the redundant MNOC architecture.[1]

Future research derived from this work aims to extend the FTTM and MNOC concept into 3-D NoCs, leveraging the demonstrated resilience and energy efficiency in more complex, high-density many-core systems.[1]

Works cited

1. Enhancing_Reliability_and_Energy_Efficiency_in_Many-Core_Processors_Through_Fault-Tolerant_Network-on-Chip.pdf
2. noxim download | SourceForge.net, accessed November 27, 2025, https://sourceforge.net/projects/noxim/
3. Fork of the Noxim NoC simulator for the exploration of 3D networks. - GitHub, accessed November 27, 2025, https://github.com/arximboldi/noxim-3d
4. davidepatti/noxim: Network on Chip Simulator - GitHub, accessed November 27, 2025, https://github.com/davidepatti/noxim
5. Noxim simulator installation: fatal error - Ask Ubuntu, accessed November 27, 2025, https://askubuntu.com/questions/943025/noxim-simulator-installation-fatal-error
6. tgff/README at master · copies/tgff - GitHub, accessed November 27, 2025, https://github.com/copies/tgff/blob/master/README
7. Extension of Noxim for Neuromorphic Computing - GitHub, accessed November 27, 2025, https://github.com/hpkhanh/Noxim-Extension
8. GitHub - habedi/SystemCAccessNoxim: All you need to build and run SystemC and AccessNoxim on your system, accessed November 27, 2025, https://github.com/habedi/SystemCAccessNoxim
9. Install SystemC 2.2.0 on Ubuntu 14.04 for executing Noxim Simulator, accessed November 27, 2025, https://askubuntu.com/questions/523971/install-systemc-2-2-0-on-ubuntu-14-04-for-executing-noxim-simulator
10. Noxim Tutorial: A Systemc Cycle-Accurate Simulator For On-Chip Networks | PDF - Scribd, accessed November 27, 2025, https://www.scribd.com/document/512167701/Noxim-Tutorial
11. Modification of the Mesh Topology · Issue #93 · davidepatti/noxim - GitHub, accessed November 27, 2025, https://github.com/davidepatti/noxim/issues/93
12. Transport Layer Assisted Routing for Non-Stationary Irregular mesh of

thermal-aware 3D Network-on-Chip systems - SciSpace, accessed November 27, 2025, https://scispace.com/pdf/transport-layer-assisted-routing-for-non-stationary-48lmdbrm96.pdf

13. TGFF Task Graphs for Free - CECS, accessed November 27, 2025, https://www.cecs.uci.edu/~papers/compendium94-03/papers/1998/codes98/pdffiles/codes98_097.pdf
14. Amirhossein-Esmaili/Energy_Aware_Task_Scheduling_in_MPSoCs: This software provides methods for energy-aware static scheduling of deadline-constrained task graphs in multiprocessor system-on-chip (MPSoC) platforms through integrated dynamic power management (DPM), and dynamic voltage and frequency scaling (DVFS). - GitHub, accessed November 27, 2025, https://github.com/Amirhossein-Esmaili/Energy_Aware_Task_Scheduling_in_MPSoCs
15. TGFF (Task graphs for free) - Tools and Benchmarks for Real-Time Systems, accessed November 27, 2025, https://www.ecrts.org/forum/viewtopic990d.html?f=1&t=51&sid=d74079af129d5480a5ac4fd1778eecc1