# Adaptive Execution Assistance for Multiplexed Fault-Tolerant Chip Multiprocessors

Pramod Subramanyan
Princeton University, Princeton, NJ
*psubrama@princeton.edu*

Virendra Singh
Indian Institute of Science, Bangalore, India
*viren@serc.iisc.ernet.in*

Kewal K. Saluja
University of Wisconsin-Madison, Madison, WI
*saluja@ece.wisc.edu*

Erik Larsson
Linköping University, Linköping, Sweden
*erik.larsson@liu.se*

***Abstract*—Relentless scaling of CMOS fabrication technology has made contemporary integrated circuits increasingly susceptible to transient faults, wearout-related permanent faults, intermittent faults and process variations. Therefore, mechanisms to mitigate the effects of decreased reliability are expected to become essential components of future general-purpose microprocessors.**

**In this paper, we introduce a new *throughput-efficient* architecture for multiplexed fault-tolerant chip multiprocessors (CMPs). Our proposal relies on the new technique of *adaptive execution assistance*, which *dynamically* varies instruction outcomes forwarded from the leading core to the trailing core based on measures of trailing core performance. We identify policies and design low overhead hardware mechanisms to achieve this. Our work also introduces a new priority-based thread-scheduling algorithm for multiplexed architectures that improves multiplexed fault-tolerant CMP throughput by prioritizing stalled threads.**

**Through simulation-based evaluation, we find that our proposal delivers 17.2% higher throughput than perfect dual modular redundant (DMR) execution and outperforms previous proposals for throughput-efficient CMP architectures.**

## I. Introduction

CMOS technology scaling fuelled by Moore's law is expected to continue for at least ten more years, continuing to provide us with a bounty of smaller, faster and lower power transistors. In the past, higher transistor counts were used to increase the performance of individual processor cores. However, increasing complexity and power dissipation of these cores forced architects to turn to chip multiprocessors (CMPs), which deliver increased performance at manageable levels of power and complexity. While technology scaling is enabling the placement of billions of transistors on a single chip, it also poses unique challenges. Integrated circuits are now increasingly susceptible to soft errors [19, 27], wear-out related permanent faults and process variations [3, 6]. As a result, engineers of the future will have to tackle the problem of designing reliable integrated circuits using an unreliable CMOS substrate.

Traditionally, fault-tolerant and high-availability systems have been limited to the domain of mainframe computers or specially-designed systems like the IBM zSeries and the Compaq NonStop®Advanced Architecture (NSAA) [5, 8]. These systems spare no expense to provide the highest possible level of reliability. While decreasing CMOS reliability implies that fault tolerance is likely to become important for the commodity market in the future [1], fault-tolerant systems for the commodity market have different requirements from traditional high availability systems. Most importantly, fault-tolerant systems for the commodity market must have low performance overhead, low energy overhead and low hardware cost.

Due to the trend of decreasing CMOS reliability, a number of proposals have attempted to exploit the inherent coarse-grained redundancy afforded by chip multiprocessors (CMPs) to provide fault tolerance [10, 11, 13, 18, 22, 28, 29, 32–35]. These proposals execute a single logical thread on two cores of a chip multiprocessor. Typically, one of these cores is configured as the leading core, while the other is configured as the trailing core. The leading core *assists the execution* of the trailing core by forwarding selected results of its execution. The results are used as *predictions* in the trailing core and help improve its performance. Results produced by the two cores are compared to detect errors.

The use of two cores to execute a single logical thread implies that the throughput of a CMP is reduced by half. Due to this *throughput loss*, a fault-tolerant system must have twice as many cores as an equivalent non-redundant system in order to provide the same throughput. Consequently, fault-tolerant systems have higher procurement costs, maintenance costs, cooling costs and energy costs. These high costs are unacceptable for general-purpose microprocessors designed for the commodity market.

This paper makes the two contributions to the state of the art. Firstly, our proposal introduces the concept of adaptive execution assistance. Our adaptive execution assistance mechanism dynamically configures the instruction outcomes forwarded from the leading core to the trailing core based on the characteristics of the workload being executed. These microarchitectural enhancements conserve power by limiting the execution assistance provided to workloads that show sufficient speedup for small amounts of assistance while simultaneously providing more assistance to workloads that need it. To the best of our knowledge, ours is the first proposal to *dynamically adjust execution assistance* based on workload behavior.

Our second contribution is the detailed design of a *throughput-efficient* fault-tolerant microarchitecture for future CMPs. Our design *multiplexes* [34] multiple trailing threads on a single trailing core using the technique of coarse-grained multithreading. This improves fault-tolerant CMP throughput at low hardware cost. We show how adaptive execution assistance is essential for the design of this microarchitecture and introduce a priority-based thread scheduling algorithm that further improves its throughput. As will be shown in our simulation-based evaluation, in the context of an network-on-chip like interconnect, our design provides 17.2% higher throughput than perfect dual modular redundant (DMR) execution and outperforms all the previous proposals for fault-tolerant CMPs that we examine.

### A. Overview

Figure 1 shows a high-level overview of a multiplexed fault-tolerant CMP. Sixteen cores are connected by mesh interconnect with four memory and I/O channels. We show three types of cores in the figure. One set of cores are shaded dark gray. These cores execute the leading threads of applications that require fault tolerance. The cores shaded light gray execute the trailing threads of applications that require fault tolerance, while the white cores execute non-redundant applications. Unlike previous fault-tolerant CMP proposals, a key difference is that the pool of leading cores is bigger than the pool of trailing cores.

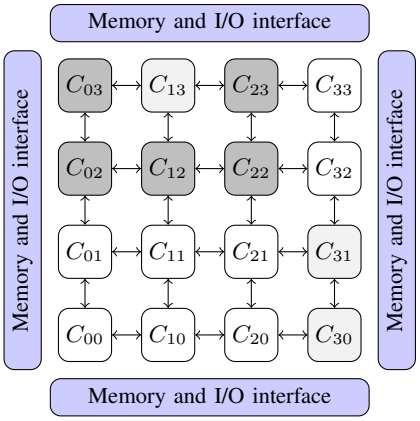


*Fig. 1:* High level overview of a Multiplexed Fault-Tolerant CMP.

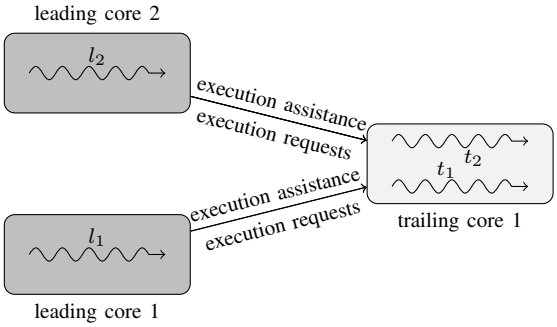


*Fig. 2:* Conceptual block diagram of 2-way multiplexing.

This is because trailing threads are *multiplexed* on a single trailing core through coarse-grained multithreading.

While many static and dynamic multiplexing configurations are possible, in this paper we explore a simple static 2-way multiplexing configuration with each trailing core multiplexing two threads.

Execution of a program begins with in the leading core. The leading core first executes a chunk of instructions and sends a request to the trailing core to redundantly execute the corresponding chunk. These requests are enqueued in the trailing core in a structure known as the Run Request Queue (RRQ). The trailing core picks entries from the RRQ and executes the chunk corresponding to the picked request. Two-way multiplexing is pictorially depicted in Figure 2

## II. Adaptive Execution Assistance

The objective of adaptive execution assistance is to tailor the assistance provided by the leading core to the trailing core based on the current phase of the workload that is being executed. Adaptive execution assistance has two important advantages over static schemes for execution assistance.

1) Adaptive schemes change and/or increase the execution assistance provided by the leading core to the trailing core for challenging workloads (i.e., workloads that do not benefit from a particular static scheme). As our results in §IV-D will show, different workloads perform well with different types of assistance. Hence, a static scheme does not scale to a diverse set of workloads.
2) Some workloads require more execution assistance (i.e., more instruction results to be forwarded) to achieve good performance. A static scheme would have to provide this higher amount to all workloads. This wastes interconnect power, core power, core-to-core bandwidth and chip area. In contrast, adaptive execution assistance only provides higher assistance to workloads that need it, increasing hardware efficiency.

### A. Design Options for Adaptive Execution Assistance

Dynamic configuration of the execution assistance mechanism is possible in a number of ways. One option is for the microprocessor to expose a set of performance monitoring counters which quantify the gain due to execution assistance. These counters can be periodically read by the OS or compiler generated code. Execution assistance policy decisions can be made based on these values by setting configuration registers of the microprocessor. The advantage of this approach is that software has fine-grained control over the execution assistance mechanism. The chief disadvantage of the approach is that software will have to be rewritten and/or recompiled to take advantage of these hardware features.

This paper explores a hardware-only mechanism for adaptive execution assistance. Our proposal evaluates the benefit due to assistance through simple and low-cost hardware mechanisms and applies fine-grained policy changes that are completely transparent to software. This approach presents two design challenges:

1) The hardware mechanism has to identify, at runtime, phases of programs that can benefit from a change to the execution assistance mechanism.
2) Once a bottleneck has been identified, the execution assistance mechanism has to supply the right kind of additional assistance that can improve performance of the current workload.

The following subsections describe our solution to these design challenges.

### B. Baseline Execution Assistance Mechanism

Our proposal uses critical value forwarding (CVF) as the baseline execution assistance mechanism [35]. Critical value forwarding identifies instructions on the *critical path of execution* and forwards the result of these from the leading to the trailing core.

Instructions on the critical path are identified using the *fanout2* heuristic. According to this heuristic, an instruction which produces a value consumed by two other inflight instructions is considered to be on the critical path and the result of such instructions are forwarded from the leading to the trailing core. Critical value forwarding breaks *data-dependence chains* in the trailing core, increasing opportunities to exploit instruction level parallelism (ILP), leading to higher instructions per clock (IPC).

Critical value forwarding special-cases branch/jump instructions. These instructions do not produce values to be consumed (i.e., have *fanout0*) but are nevertheless very important for performance. Branch, jump and call instructions that are *mispredicted* are treated as being on the critical path and the results of these instructions are forwarded from the leading to the trailing core. In the trailing core, these branch outcomes are used instead of branch predictions.

In this paper we introduce two *adaptive* enhancements to critical value forwarding: *adaptive branch forwarding* and *adaptive critical value forwarding*. The details of these enhancements are presented in the following subsections.

### C. Adaptive Branch Forwarding

The baseline execution assistance mechanism forwards the outcomes of mispredicted branches from the leading core to the trailing core. For many workloads this execution assistance is sufficient to ensure good performance in the trailing core. However, some of

| ABF Level | Branch Outcomes Forwarded |
|---|---|
| 1 | mispredicted only |
| 2 | mispredicted + every 8th branch |
| 3 | mispredicted + every 4th branch |
| 4 | mispredicted + every 2nd branch |
| 5 | all branches |

*TABLE I:* Levels for adaptive branch forwarding.

the workloads that we examine, such as *mesa_crafty*, *gzip_mcf* and *gap_crafty*, lose performance a due to a dispatch bottleneck caused by unresolved branches in the trailing core.

Adaptive branch forwarding mitigates this bottleneck by forwarding more branch outcomes from the leading to the trailing core. Note that branches outcomes forwarded from the leading core to the trailing core are correct unless affected by an error. Since an error in a forwarded branch will be detected at the next fingerprint comparison (see III-A), forwarded branches can be treated as resolved at the time of dispatch.

**Operation of Trailing Core**: Control of the adaptive branch forwarding mechanism is through a counter that tracks dispatch stalls in the trailing core. The counter is incremented when dispatch is stalled due to unresolved branches. It is decremented when dispatch is stalled due to any other reason. If the value of the counter is above a threshold, indicating that unresolved branches are the main bottleneck, at the time of next synchronization between the leading and trailing cores, the trailing core requests the leading core to increase the level of adaptive branch forwarding. If the value is below the threshold, the trailing core requests the leading to decrease the level of adaptive branch forwarding.

**Operation of Leading Core**: Depending on the current level of adaptive branch forwarding, the leading core either forwards just mispredicted branches or mispredicted branches and every $N^{th}$ branch. Levels used in our implementation are shown in Table I. Note that with each message received from the trailing core, the level is either increased or decreased, i.e., the system automatically relearns the level for each new program phase.

### D. Adaptive Critical Value Forwarding

Adaptive critical value forwarding attempts to increase execution assistance for workloads by monitoring retirement stalls in the *leading core*. Instructions results forwarded from the leading core to the trailing core are buffered in the instruction result queue (IRQ) in the trailing core (see §II-F). When the trailing core is unable to keep up with the leading core, the IRQ becomes full; eventually stalling retirement in the leading core.

Adaptive critical value forwarding attempts to mitigate these stalls by providing additional execution assistance to help the trailing core exploit more instruction-level parallelism. The default mechanism uses the *fanout2* heuristic to identify instructions on the critical path. When the leading encounters IRQ-full retirement stalls, it *also* uses the *ROB-stall* heuristic. The *ROB-stall* heuristic marks instructions which reach the head of the reorder buffer (ROB) without being executed (i.e., instructions stalling retirement) as critical.

When instruction retirement is stalled due to an IRQ-full stall, a counter is incremented. When instruction retirement is stalled due to any other type of stall, the counter is decremented. If the value of the counter is greater than the threshold, the *ROB-stall* heuristic is used along with the *fanout2* heuristic. Thus the result of an instruction that reached the reorder buffer head without being executed (i.e., a *ROB-stalling instruction*) *or* any instruction with two in-flight consumers (i.e., a *fanout2* instruction) is forwarded to the trailing core.

Increasing execution assistance from the leading core to the trailing core has two effects. First, there is greater contention for IRQ-entries in the trailing core. This can potentially reduce performance. Second, the availability of results for more instructions improves performance by breaking a greater number of data-dependence chains in the trailing core. Our results in §IV-D show that the second effect dominates the first for the workloads that we study.

### E. Priority-Based Thread Selection

The order in which execution requests are processed by the trailing core has significant effect on performance. This section presents a priority-based scheduling algorithm that assigns a higher priority to trailing core threads that are stalled in the leading core. This algorithm is implemented in hardware and determines which thread drives the fetch engine of the processor.

```
Input: currentThread, otherThread
Input: threadStalled[0], threadStalled[1]
Input: currentRunLength, maxRunLength
Output: selectedThread
if threadStalled[0] and threadStalled[1] then
    selectedThread ← currentThread
else
    if threadStalled[0] and (not threadStalled[1]) then
        selectedThread ← 0
    else if threadStalled[1] and (not threadStalled[0]) then
        selectedThread ← 1
    else
        if currentRunLength < maxRunLength then
            selectedThread ← currentThread
        else
            selectedThread ← otherThread
        end
    end
    update currentRunLength
    update currentThread
end
```

**Algorithm 1**: Priority-Based Thread Selection Algorithm

Algorithm 1 shows priority-based thread scheduling. The algorithm attempts to schedule the thread that is stalled in the leading core first. If both leading core threads are stalled, or if no threads are stalled, the algorithm prioritizes the currently executing thread to minimize the costs of context switching. The $maxRunLength$ parameter ensures fairness by forcing thread switching after a certain number of selections.

As will be shown in §IV-D, the priority-based scheduling algorithm improves performance over the round robin scheduling algorithm proposed in MRE [34] by 2.2% on average. The highest gains of 8.2% and 5.7% respectively are seen in the challenging workloads *crafty_sixtrack* and *gap_crafty*.

### F. Putting It All Together

Figure 3 shows a block diagram of a processor that supports multiplexing with adaptive execution assistance. Blocks which are shaded are our additions to a conventional out-of-order superscalar core. Blocks in blue are used in the trailing core, while the critical value identification heuristic is used only in the leading core. Fingerprinting circuitry (see §III-A) is used in both cores.

The branch outcome queue (BOQ) [23] holds branch outcomes and corresponding instruction tags received from the leading core. The BOQ is examined in parallel with the branch predictor. If an outcome is available in the BOQ, it is used instead of the prediction.

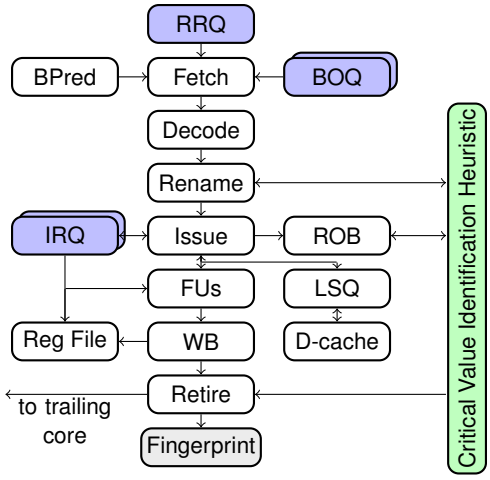


*Fig. 3:* Block digram of a multiplexed fault-tolerant core.

The instruction result queue (IRQ) [35] holds instruction results and their corresponding tags received from the leading core. The IRQ is examined at the time of instruction dispatch. If a value is available in the IRQ, it is written immediately to the destination physical register allowing dependent instructions to begin execution. Note that when this instruction eventually executes in the trailing core, it writes its computed value into the destination physical register for a second time.

### G. Hardware Cost

The primary area cost due to our proposals are the 512-entry IRQ and BOQ structures. Using CACTI 5.3 [37] we estimate the area of these structures to be about $0.05mm^2$: less than 1% the area of a single processor core in 32nm technology, and about 0.015% of the area of the entire chip. Besides these two queues, the RRQ, two counters and fingerprinting circuitry also consume a small amount of additional area. The priority-based scheduling algorithm can be implemented with a small number of flops, gates and a counter to track the $currentRunLength$. Therefore, we expect that these hardware overheads will be negligible.

Since the microarchitectural structures introduced for multiplexed fault tolerance can be dynamically turned off, all cores can be used for non-redundant execution without any power or performance penalty.

## III. Functional Description of Fault Tolerance Mechanisms

This section discusses four important issues that need to be addressed for any fault-tolerant system: fault detection, fault isolation, fault recovery and fault coverage.

### A. Fault Detection

Faults are detected by comparing *fingerprints of execution* generated independently by the two cores. A fingerprint is a CRC-based hash of register file updates, load/store addresses, store values and branch targets [28]. It is computed at the time of instruction retirement and is a deterministic function of the code and input for a single-threaded program. For multithreaded programs, our proposal for partial load replication (PLR) [35] causes both leading and trailing threads resolve data races in an identical manner ensuring deterministic fingerprinting.

Since a fingerprint compresses the execution history of a program into a single checksum value, there is a possibility that errors may be undetected due to *fingerprint aliasing*. Fingerprint aliasing occurs when two *different* execution histories result in the same fingerprint, leading to errors going undetected. However, a number of previous studies have concluded that the probability of fingerprint aliasing is minuscule [13, 28, 34] for errors rates that are likely to be observed currently and in the near future.

**Detecting Errors In Forwarded Values**: If the leading core forwards an erroneous value to the trailing core, the error will be detected during fingerprint comparison. To see why this is true, assume that an instruction $l_n$ in the leading core forwards an erroneous value to the corresponding instruction $t_n$ in the trailing core. Assume without loss of generality that $l_n$ is the earliest instruction that forwards an erroneous value. Therefore, under a single-error assumption, when $t_n$ executes in the trailing core, it will compute the correct result because all of its input operands will be correct. Since $l_n$ and $t_n$ will compute different results, the fingerprints computed in the two cores will be different, detecting the error.

### B. Fault Isolation

A fault can occur at any point during execution, but it is detected only when fingerprints are compared. Fault isolation ensures that fault does not propagate outside the cores to I/O devices or main memory.

For this, the state bits stored with each L1 cache line are augmented by two bits. One bit tracks *unverified* cache lines. A cache line is marked as unverified each time it is written to. All unverified bits are flash cleared when a fingerprint comparison succeeds. The cache replacement algorithm does not victimize unverified lines. This ensures fault isolation because freshly-updated data does not leave the L1 caches before verification.

A second bit, called the *C2C* bit, tracks lines obtained through cache to cache transfers. Loads which execute from unverified and C2C lines are not re-executed in the trailing core. For such loads, the leading core supplies the value of the load to the trailing core where it is used without verification ensuring deterministic fingerprinting even in the presence of data races [31, 35].

### C. Fault Recovery

Recovering from a fault essentially means restoring register and memory values to their state at the time of the previous checkpoint. Restoration of register state is easily done through register checkpointing mechanisms. Such mechanisms are already present in contemporary microprocessors for two reasons: (1) to recover from soft errors during execution and (2) to save the state of idle cores being put to sleep for power reasons[15].

Our proposal saves and restores memory state from the L2 cache of the microprocessor. This is possible because all the lines that have been written to (i.e., modified) since the last checkpoint are contained in the L1 cache. These lines are also marked unverified. Thus, flash invalidating all unverified lines is sufficient to restore memory state. A subtle implementation detail here is that each time a verified line is marked as unverified, the verified version of the line must be written to the L2 cache.

### D. Fault Coverage

Our proposal provides full fault coverage for errors that occur inside the processor cores with the exception of some parts of the memory accesses logic. The reduction in coverage of memory access logic is because the trailing core does not fully re-execute load instructions that are involved in data races. Our experiments with the SPLASH2 [38] suite of programs showed that more than 92% of load instructions are fully re-executed in the trailing core, bounding the loss in fault coverage of memory-access circuitry to only 8% on average. We assume that L1 and L2 caches are protected by error correcting codes.

| Configuration | # of Cores | Comments |
|---|---|---|
| CRT-4 | 4 | This configuration is based on chip-level redundantly threaded (CRT) [18] processors proposed by Mukherjee et al. It uses four cores to execute two logical threads redundantly. |
| CRT-3 | 3 | This asymmetric configuration, which is a modification of CRT, uses only three cores to execute two logical threads. Of these cores, the lone redundant core uses simultaneous multithreading (SMT) to multiplex two trailing threads for execution. |
| MRE-3 | 3 | This is the multiplexed redundant execution (MRE) proposal from [34] which also uses three cores to execute two logical threads. However, the third core uses coarse-grained multiplexing rather than simultaneous multithreading, reducing hardware cost. |
| MuxCVF-3 | 3 | This proposal improves MRE by replacing its execution assistance mechanism with critical value forwarding (CVF) [35]. CVF identifies instructions on the critical path of execution and forwards the results of these from the leading core to the trailing core. On average, it provides higher speedup and requires lower communication bandwidth than MRE's policy of forwarding all load values and branch outcomes. |
| MuxCVF+ABF-3 | 3 | Adaptive branch forwarding (ABF) (see §II-C) improves MuxCVF by adapting the number of branches forwarded from the leading core to the trailing core based on the characteristics of the workload. |
| MuxAEA-3 | 3 | Adaptive Execution Assistance (AEA) (see §II-D) improves MuxCVF by incorporating adaptive branch forwarding and adaptive critical value forwarding. These techniques dynamically vary the execution assistance supplied by the leading core to the trailing core at runtime based on identified execution bottlenecks. |
| MuxAEA+PP-3 | 3 | The priority pick (PP) scheme improves MuxAEA throughput by prioritizing threads stalled in the leading core. |

*TABLE II:* List of Evaluated Configurations

## IV. Evaluation

In this section, we present a simulation-based evaluation of our proposal. To gain an understanding of the performance impact of our proposals and further put our results in context, we evaluate the configurations listed in Table II. We present both single-threaded and multiprogrammed evaluation results.

### A. Methodology

Our evaluation is conducted using a modified version of the SESC [24] execution-driven simulator. The simulator models an out-of-order superscalar microprocessor in a detailed manner and fully executes "wrong-path" instructions. All the microarchitectural structures required for multiplexed execution including unverified bits in the L1 data cache are simulated. Details of the CMP configuration are shown in Table III.

For single-thread performance evaluation, we use twenty benchmarks from the SPEC CPU 2000 suite. For each benchmark we execute a single SimPoint [26] of length one billion instructions. For the multi-threaded results, we constructed a suite of thirteen 2-program workloads from the SPEC CPU 2000 suite that provide a representative sampling of speedup behaviour due to critical value forwarding [31]. Each thread in these workloads is fast-forwarded by three billion instructions. A total of one billion instructions are executed.

### B. Interconnect Model

We simulate an interconnect that is an approximation of future network-on-chip based multiprocessors. We assume messages from one of the cores redundantly executing a thread reach the other core after exactly three hops. Each hop results in random delay that uniformly varies between four and eight cycles. Although we do not show detailed results here due to a lack of space, we found that increasing the number of hops and changing hop latency had minimal impact.

Previous proposals like Slipstream [36], CRT [18] and Reunion [29] have assumed the existence of a dedicated interconnect between the two cores performing redundant execution. These proposals also optimistically assume that the interconnect latency is only a few cycles. Although these latencies may be achievable for future chip multiprocessors if adjacent cores are used for redundant execution, this may not always be possible for the following reasons:

1) If redundant execution is turned-on dynamically, it may not be possible to allocate adjacent cores because one of the cores of a pair may already be executing an application that cannot be rescheduled.
2) Software may explicitly "pin" threads to cores using processor affinity system calls [7, 17].
3) In chips affected by intra-die variation, it may be necessary to use "slow" cores for redundant execution [31]. In such a scenario, the trailing core has to be chosen among a subset of available cores, increasing the likelihood that adjacent cores are not used for redundant execution.

### C. Evaluation Metrics

To determine the slowdown when compared to non-redundant execution, we use the weighted speedup metric proposed by Snavely and Tullsen [30]. Weighted speedup is nothing but the average of the slowdown suffered by each thread due to fault-tolerant execution.

$$WTSP = \frac{1}{N_{threads}} \sum_{i=1}^{N_{threads}} \frac{IPC_{fault-tolerant}(i)}{IPC_{non-fault-tolerant}(i)}$$

To evaluate the CMP throughput increase due to our proposals we use the normalized throughput per core (NTPC) metric from [34]. NTPC is defined as the ratio of the sum of normalized slowdown of each thread due to fault-tolerant execution to the number of cores executing the workload.

$$NTPC = \frac{1}{N_{cores}} \sum_{i=1}^{N_{threads}} \frac{IPC_{fault-tolerant}(i)}{IPC_{non-fault-tolerant}(i)}$$

Note that NTPC is just the WTSP metric scaled by the number cores. For an ideal DMR system, the NTPC is 0.5 because the number of cores is double the number of threads and there is no slowdown due to redundant execution.

### D. Multiplexing Performance

Figure 4 compares the weighted speedup of the workloads for each of the configurations shown in Table II. CRT-4 has mean slowdown of 14.7%, CRT-3 has a mean slowdown of 21.2% and MRE-3 experiences a mean slowdown of 19.2%. MuxAEA+PP-3 is the best

*TABLE III:* CMP configuration

| | | | | | |
|---|---|---|---|---|---|
| Fetch/issue/retire | 4/4/4 | Mem/Int/FP units | 4/6/4 | Branch predictor | hybrid/16k/16k/16k |
| ROB size | 128 instructions | I-cache | 32k/64B/4-way/2 cycles | BTB | 4k entries/4-way |
| Integer/FP window | 64/32 instructions | D-cache | 64k/64B/4-way/2 cycles | RAS | 32 entries |
| Load/store queue | 32 instructions | Private L2 cache | 2 MB/64B/8-way/24 cycles | IRQ/LVQ size | 512 |
| Interconnect latency | 48 cycles | Memory | 400 cycles | BOQ size | 512 |
| Checkpointing interval | 50k instructions | DVFS update interval | 1 $\mu s$ | DVFS update latency | 100 $ns$ |
| DVFS voltage levels | 0.5 - 1.0 V | DVFS frequency levels | 1.5-3.0 GHz | # of DVFS levels | 6 |

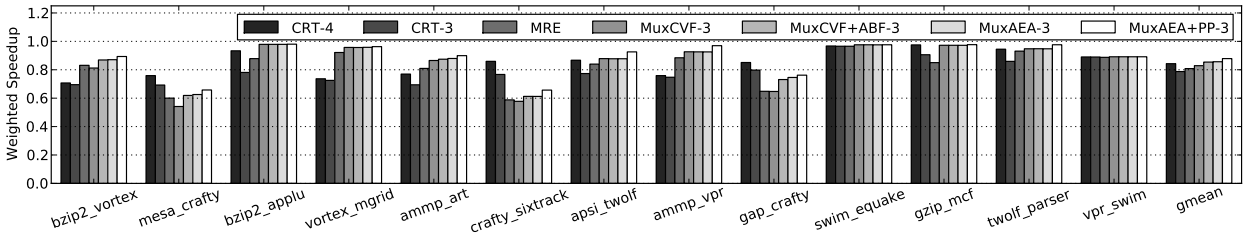


*Fig. 4:* Weighted Speedup (WTSP) for multiplexing configurations and CRT-4.

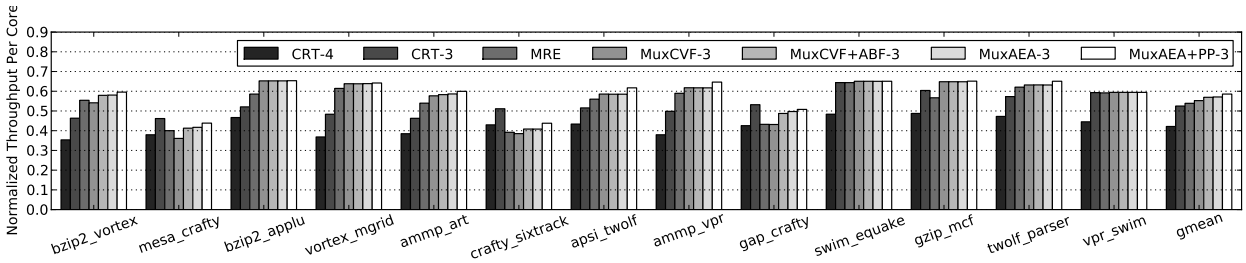


*Fig. 5:* Normalized Throughput Per Core (NTPC) for multiplexed configurations and CRT-4.

performing proposal, and it has mean slowdown of 12.1%. *Note that this configuration uses only 3 cores to execute two logical threads but outperforms CRT-4 which uses four cores.*

We can also see that critical value forwarding (MuxCVF) improves performance over MRE by 2.1%. Adaptive branch forwarding further improves performance by 2.5%. Adaptive critical value forwarding adds 0.3% of performance to adaptive branch forwarding. Finally, 2.2% performance is gained through priority-based thread selection.

Figure 5 shows the NTPC metric for the same configurations. As expected, CRT-4 has the lowest throughput because it uses four cores to execute two logical threads while all the other configurations use three cores to execute the same number threads. NTPC of MuxAEA with priority-based thread selection is 0.586, showing that NTPC delivers 17.2% higher throughput than perfect dual modular redundant system.

When three cores are used to execute two logical threads redundantly, the highest achievable NTPC is 0.67. A number of workloads such as bzip2_applu, vortex_mgrid, ammp_vpr, swim_equake, gzip_mcf and twolf_parser approach this limit.

### *E. Bandwidth Requirements*

Figure 6 shows the bandwidth requirements for the configurations. Clearly, CRT-4, CRT-3 and MRE have the highest bandwidth requirements. This is because these proposals transmit each load value and branch outcome from the leading to the trailing core. In contrast, critical value forwarding and adaptive execution assistance transmit the values of instructions on the critical path of execution. As a result, these configurations require only half the bandwidth of CRT and MRE.

Furthermore, adaptive execution assistance is able to increase performance over critical value forwarding at only a small additional bandwidth cost. A related observation here is that adaptive execution assistance pays a bandwidth cost only for workloads that need it, thereby optimizing interconnect power.

### *F. Single-Threaded Application Performance*

Figure 7 shows the normalized IPC for the four different proposals that are evaluated in this paper. Normalized IPC is defined as the ratio of the instructions per clock (IPC) of fault-tolerant execution to the IPC of a non-fault-tolerant baseline. On average both MuxCVF and MuxAEA have a mean slowdown of only 0.5%. MRE has a mean slowdown of 11.3% while CRT has a mean slowdown of 5.6%.

For this configuration there is no difference between MuxCVF and MuxAEA because AEA is an adaptive scheme that increases the execution assistance provided by the leading core when the trailing core is unable to execute as fast as the leading core. For the single-threaded workloads, the trailing core is easily able to keep pace with the leading core, so none of the features of adaptive execution assistance are dynamically activated.

The negligible performance loss due to single-threaded execution is an important result and demonstrates that for challenging workloads which do not perform well under multiplexing, performance can be regained through non-multiplexed execution. We envision the implementation of an operating system mechanism that dynamically deconfigures multiplexing based on performance-counter measurements in MuxAEA CMPs.

### *G. Discussion of Results*

Our proposal provides higher throughput than CRT-4, CRT-3 and MRE; which are the previous proposals for throughput-efficient fault-tolerant CMPs incorporating execution assistance. This higher throughput comes along with the advantage of lower bandwidth requirement. Core-to-core bandwidth is likely to be a bottleneck in

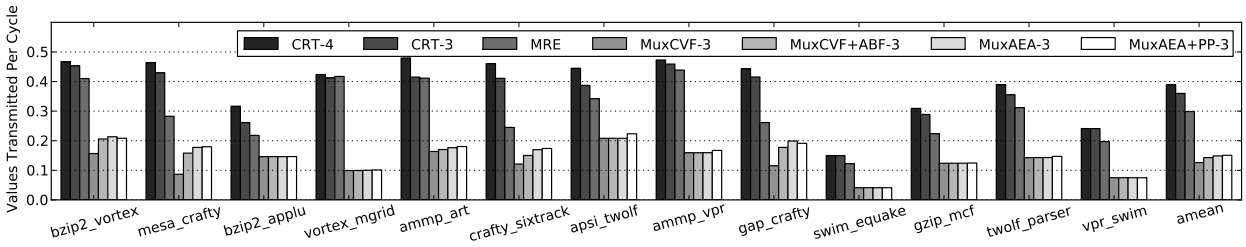


*Fig. 6:* Bandwidth requirements for multiplexing configurations and CRT-3.

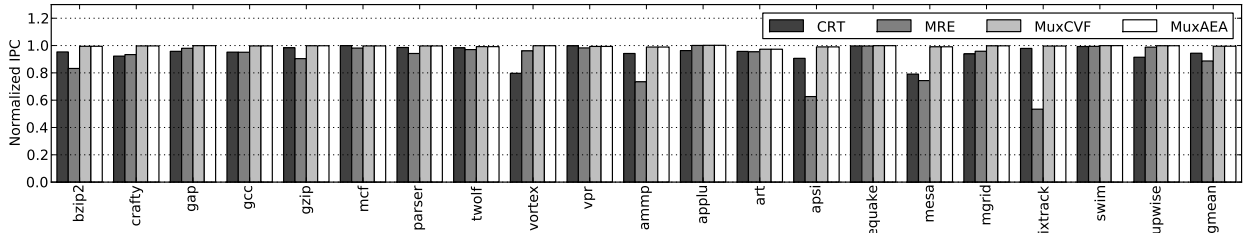


*Fig. 7:* Normalized IPC for SPEC CPU 2000 benchmarks.

future CMPs [12]. Hence our proposal is better suited for the NoC-like interconnects in CMPs of the future.

Although we do not show the results here, we found that our proposal's energy consumption is similar to that of previous proposals like CRT and MRE. We leave a detailed examination of energy consumption issues to future work.

Our proposal uses coarse-grained multithreading to multiplex threads. Compared to simultaneous multithreading (SMT), coarse-grained multithreading has lower hardware cost. A study by Lee and Brooks [14] investigating the power-performance efficiency of SMT processors found that SMT processors require a wide (e.g. 8-way) and deep superscalar pipelines to achieve optimality, while CMPs are able to achieve this for narrower issue widths and shallower pipelines. The requirement for a wide superscalar with deep pipelines significantly increases design complexity and leads to large layout blocks and additional circuit delays [20]. Therefore, an important advantage of our design is that it alleviates the need for SMT for throughput-efficient fault tolerance without compromising on performance.

## V. Related Work

The concept of execution assistance was first explored in AR-SMT [25], DIVA [4] and Simultaneous Redundantly Threaded (SRT) processors [23]. AR-SMT and DIVA forwarded all values from the leading to the trailing thread while SRT forwards the result of all load and branch instructions. In contrast, critical value forwarding, focuses on the few instructions that are on the *critical path* of execution. Critical value forwarding achieves most of the speedup obtained by forwarding all instructions at a fraction of the bandwidth cost. Our proposal also distinguishes itself from these mechanisms by *adaptively* determining the instruction results to be forwarded.

A class of proposals like Slipstream [36], Paceline [11], and Performance-Correctness Decoupled Architectures [9] attempt to exploit the idea of execution assistance to improve the performance of multicore microprocessors. These mechanisms use some form of speculation in the leading core and use the trailing core to recover from mis-speculation. Our proposal differs from this body of work in two ways. Firstly, we do not target the problem of multicore performance improvement, eliding the need for these speculative mechanisms in the leading core. Secondly, these proposals all use a static algorithms for execution assistance, unlike our adaptive execution assistance mechanism which tailors the execution assistance provided based on the workload.

A related proposal is Necromancer [2] which uses faulty cores to provide execution assistance to fully-functional cores. Even though faulty cores cannot execute applications on their own, the execution assistance they can provide significantly speeds up the trailing core. This technique is especially effective when the faulty core is a "big" core (an out-of-order superscalar core) while the non-faulty core is an in-order core. Rashid, Saluja and Ramanathan [21] propose an architecture where the majority of the functional units of a superscalar are used to execute the leading thread, with the trailing thread using the remaining units. This proposal amortizes the cost of redundant execution over the multiple functional units that are inherent to out-of-order superscalar microprocessors.

Using the core-level redundancy inherent in chip multiprocessors for fault tolerance is a well studied idea [2, 10, 13, 16, 18, 22, 29, 32–35]. Chip-level Redundantly Threaded processors (CRT) execute the leading and trailing threads on different cores for transient and permanent fault tolerance. The leading core supplies execution assistance to the trailing core by forwarding all load values and branch instructions. Store instruction results are forwarded from the trailing core to the leading core where they are compared to detect errors. CRT can only detect faults, it cannot recover from them. CRTR enables fault recovery as well as fault detection [10].

RECVF [35] introduced the technique of critical value forwarding and showed how it could be exploited for energy-efficient redundant execution. This paper improves critical value forwarding by introducing a dynamic execution assistance mechanisms that adapts to the characteristics of the workload. A second difference is that we use the execution assistance mechanism to improve fault-tolerant CMP throughput, while the proposal in [35] uses critical value forwarding to improve energy-efficiency of fault-tolerant CMPs. As our results in §IV-D showed, the techniques proposed in this paper improve performance over critical value forwarding by 5.0%.

The technique of multiplexing was introduced in multiplexed redundant execution (MRE) [34]. This work uses the techniques of

adaptive execution assistance and priority-based thread selection to improve over MRE's performance by 7.1% (see §IV-D).

## VI. Conclusion

Decreasing CMOS reliability in future technology nodes has resulted in a pressing need for low-cost fault-tolerant general-purpose chip multiprocessors (CMPs). In this paper, we presented the design of a throughput-efficient fault-tolerant CMP. An enabling technique for this design is our proposal of adaptive execution assistance. Adaptive execution assistance tailors the instruction results forwarded from the leading core to the trailing core based on the program phase of the workload that is being executed. We also introduced a new priority-based thread scheduling algorithm that further increases fault-tolerant CMP throughput. Our evaluation showed that in the context of an NoC-like interconnect, our design provides 17.2% higher throughput than perfect dual modular redundant execution. Our proposal provided higher performance at a lower bandwidth cost than all previous fault-tolerant CMP proposals that we examined.

## References


[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems With Commodity Multi-core Processors. In *Proc. of the 34th Int'l Symp. on Comp. Arch.*, pages 470–481, 2007.

[2] Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Necromancer: Enhancing system throughput by animating dead cores. In *Proc. of the 37th Annual Int'l Symp. on Comp. Arch.*, ISCA '10, 2010.

[3] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao. Reliable Systems on Unreliable Fabrics. *IEEE Design and Test*, 25(4):322–332, 2008.

[4] Todd Austin. DIVA: A Reliable Substrate For Deep Submicron Microarchitecture. Design. In *Proc. of the 32nd Int'l Symp. on Microarchitecture.*, pages 196–207, 1999.

[5] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop® Advanced Architecture. In *Proc. of 35th Int'l Conf. on Dependable Systems and Networks*, pages 12–21, 2005.

[6] S. Y. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[7] Microsoft Corp. SetThreadAffinityMask Function. MSDN Libary., 2011.

[8] M.L. Fair, C.R. Conklin, S. B. Swaney, P. J. Meaney, W. J. Clarke, L. C. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber. Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990. *IBM Journal of Research and Development*, 2004.

[9] A. Garg and M. Huang. A Performance Correctness Explicitly-Decoupled Architecture. *Proc. of the 38th Int'l Symp. on Comp. Arch.*, pages 306–317, 2008.

[10] M. Gomma, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. *Proc. of the 30th Int'l Symp. on Comp. Arch.*, pages 98–109, 2003.

[11] B. Greskamp and J. Torrellas. Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking. In *Proc. of the 16th Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 213–224, 2007.

[12] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proc. of the 32nd Int'l Symp. on Comp. Arch.*, 2005.

[13] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In *Proc. of the 37th Int'l Conf. on Dependable Systems and Networks*, 2007.

[14] B. Lee and B. Brooks. Effects of Pipeline Complexity on SMT/CMP Power-Perf. Efficiency. *Workshop on Complexity Effective Design in conjunction with 32nd Int'l Symp. on Comp. Arch.*, 2005.

[15] M. Mack, W. Sauer, S. Swaney, and B. Mealey. IBM Power6 Reliability. In *IBM Journal of R&D, 51(6)*, 2007.

[16] N. Madan and R. Balasubramonian. Power-efficient Approaches to Redundant Multithreading. *IEEE Transactions on Parallel and Distributed Systems*, pages 1066–1079, 2007.

[17] Linux System Calls Manual. sched_setaffinity Function, 2011.

[18] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. *Proc. of the 29th Int'l Symp. on Comp. Arch.*, pages 99–110, 2002.

[19] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The Soft Error Problem: An Architectural Perspective. In *Proc. of the 11th Int'l Symp. on High Perf. Comp. Arch.*, pages 243–247, 2005.

[20] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. *Proc. of the 7th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, 1996.

[21] Faisal Rashid, Kewal K. Saluja, and Parameswaran Ramanathan. Fault Tolerance through Re-Execution in Multiscalar Architecture. In *Proceedings of the 2000 Int'l Conf. on Dependable Systems and Networks*, DSN '00, pages 482–491, 2000.

[22] M. W. Rashid, E. J. Tan, M. C. Huang, and D. H. Albonesi. Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance. In *Proc. of the 14th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 315–328, 2005.

[23] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *Proc. of the 29th Int'l Symp. on Comp. Arch.*, pages 25–36, 2002.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator. `http://sesc.sourceforge.net/`, 2005.

[25] E. Rotenberg. AR-SMT: A Microarchitecture Approach to Fault Tolerance in a Microprocessor. *Proc. of 29th Int'l Symp. on Fault-Tolerant Computing*, pages 84–91, 1999.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *Proc. of the 10th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[27] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. *Proc. of the 32nd Int'l Conf. on Dependable Systems and Networks*, pages 389–398, 2002.

[28] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding Soft Error Detection Latency and Bandwidth. *Proc. of the 9th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, pages 224–234, 2004.

[29] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. *Proc. of the 39th Int'l Symp. on Microarchitecture.*, pages 223–234, 2006.

[30] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proc. of 8th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, 2000.

[31] Pramod Subramanyan. Efficient Fault Tolerance in Chip Multiprocessors Using Critical Value Forwarding. *M.Sc (Engg.) Thesis, Indian Institute of Science*, 2011.

[32] Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. Power-Efficient Redundant Execution for Chip Multiprocessors. *Proc. of 3rd Workshop on Dependable and Secure Nanocomputing held in conjunction with DSN 2009*, 2009.

[33] Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. Energy-Efficient Redundant Execution for Chip Multiprocessors. *Proc. of 20th ACM Great Lakes Symp. on VLSI*, 2010.

[34] Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. Multiplexed Redundant Execution: A Technique for Efficient Fault Tolerance in Chip Multiprocessors. *Proc. of Design Automation and Test in Europe*, 2010.

[35] Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. Energy-Efficient Fault Tolerance in Chip Multiprocessors Using Critical Value Forwarding. *Proc. of 40th Int'l Conf. on Dependable Systems and Networks*, 2010.

[36] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *Proc. of the 9th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, pages 257–268, 2000.

[37] S. Thoziyoor, N. Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. *Technical Report HPL-2008-20, HP Labs*, 2008.

[38] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization And Methodological Considerations. In *Proc. Of The 22nd Int'l Symp. on Comp. Arch.*, 1995.