CASTED: Core-Adaptive Software Transient Error Detection for Tightly Coupled Cores

Konstantina Mitropoulou Vasileios Porpodas Marcelo Cintra[1]
School of Informatics
University of Edinburgh
{*K.Mitropoulou@sms., v.porpodas@, mc@staffmail.*}*ed.ac.uk*

***Abstract*—Aggressive silicon process scaling over the last years has made transistors faster and less power consuming. Meanwhile, transistors have become more susceptible to errors. The need to maintain high reliability has led to the development of various software-based error detection methodologies which target either single-core or multi-core processors.**

In this work, we present CASTED, a Core-Adaptive Software Transient Error Detection methodology that focuses on improving the impact of error detection overhead on single-chip scalable architectures that are composed of tightly coupled cores. The proposed compiler methodology adaptively distributes the error detection overhead to the available resources across multiple cores, fully exploiting the abundant ILP of these architectures. CASTED adapts to a wide range of architecture configurations (issue-width, inter-core delay).

We evaluate our technique on a range of architecture configurations using the MediabenchII video and SPEC CINT2000 benchmark suites. Our approach successfully adapts to (and regularly outperforms by up to 21.2%) the best fixed state-of-the-art approach while maintaining the same fault coverage.

***Keywords*-adaptation, error detection**

I. Introduction

The design of today's processors is based on the assumption that the underlying CMOS transistors and interconnect never fail. However, the constant demand for higher performance and less power consumption of modern microprocessors have led to smaller transistor technologies. The side-effect of this trend is that processors have become more susceptible to errors.

Transient errors (a.k.a. soft errors) occur only once and do not persist [29]. Although transient errors are temporal phenomena, they can alter the program's execution. The main sources of transient errors can be either external factors such as alpha particles and cosmic radiation or internal ones such as a fluctuating power supply and electromagnetic interference [6]. It has been proven in prior work [31] that these factors are critical for architectures that use small transistor technologies. Moreover, voltage scaling reduces the noise margins and increases the possibility of uncharged transistors [6].

[1]Marcelo Cintra is currently on sabbatical leave at Intel Labs.

*This work was supported in part by the EC under grant ERA 249059 (FP7).

Transient errors can cause whole systems to fail. For instance, in 2000, Sun Microsystems received several complaints from customers such as America On-line, eBay, and Los Alamos Labs that they experienced system failures because of transient errors [18]. Because of this, high-end systems like servers and systems where safety is an important design parameter, include some error detection and recovery mechanisms. The common design practice for those systems is to replicate the critical hardware components to ensure that if an error occurs in one component, the other will be able to detect it. This makes error detection as simple as comparing the outcomes of the identical components. A typical example of this approach is IBM's G4 and G5 processors [27] where two copies of instruction units and execution units are used.

Processors with hardware error detection are costly due to extra hardware and the high development cost. An alternative to this is software-based error detection. These methodologies operate at a higher abstraction level restricting the error detection only to errors that might affect the application's output. Moreover, they give the designer the flexibility to choose the program region that needs protection. The main concept of software error detection is also redundancy, but instead of replicating the hardware, we replicate the program code. Code redundancy guarantees that the code is executed more than once in time or in space. That is, the duplicated code can execute on the same hardware but at a different time, or at the same time on a different hardware unit. Because of the nature of transient errors, only one of the codes (original or replicated) is affected by an error.

In this paper, we introduce an adaptive way to map the error detection code (original and redundant code) to the cores. The proposed software-based error detection methodology takes advantage of the features (such as fast inter-core communication) of scalable tightly coupled cores [8][32][37] to optimally distribute the error detection overhead across cores/clusters while maintaining high reliability. In the text, we use the terms *cluster* and *core* interchangeably as the proposed technique can apply to architectures with tightly-coupled cores or to clustered VLIWs.

CASTED is an adaptive compiler mechanism that generates code for different architecture configurations. It suc-

ceeds this in two steps. Firstly, it generates the redundant code (replicated + checking code) for the detection of transient errors. Secondly, it schedules the error detection code taking into consideration the capabilities of the target architecture (issue width, inter-core delay). This suggests that depending on the conditions, the code can be: i) executed all within a single core ii) split into sections, some of which will run on one core and the rest on the other.

Existing approaches are non-adaptive, meaning that they assign either the whole program to a single core, or the original code to one core and the replicated code to another. This is because they target commodity single-core or multi-core architectures. We show that this is sub-optimal for our target.

To summarize, this paper's contribution is a novel error detection methodology that automatically generates error detection code and:

- adaptively distributes the error detection overhead across available resources
- optimizes it for a wide range of core counts, issue-widths and inter-core communication latencies
- achieves improved performance with no impact on the fault coverage.

II. Background and Motivation

A. Error Detection Overhead

Software-based error detection methodologies are expensive in terms of performance. The replicated and checking code increase code size by a factor greater than 2 [23]. Recent works try to reduce this overhead in many ways. In attempt to decrease the overhead of memory traffic, SWIFT[23] does not replicate store instructions. This can be safely done since the memory subsystem is protected by its own error detection mechanisms like ECC, parity checking etc. In addition, SWIFT improves performance by significantly reducing the number of checking points (synchronization points). Shoestring[9] further decreases performance overhead by reducing the number of replicated instructions even more, but relies on the operating system to detect a significant fraction of the errors.

CASTED tries to "hide" the error detection overhead by fully exploiting the available ILP of architectures with tightly coupled cores. These architectures look like Fig.1. They are wide-issue scalable clustered architectures (such as [8],[32],[37]). Such architectures differ from traditional monolithic (non-clustered) designs in that critical resources (e.g the register file) are partitioned into small parts. Each part along with other resources (e.g. functional units) are tightly connected together and form a cluster. Within a cluster the data transfers are fast and energy efficient. Across clusters there is an inter-cluster delay penalty.

Tightly coupled cores differ from traditional multi-core architectures in the inter-core communication delay. Contrary

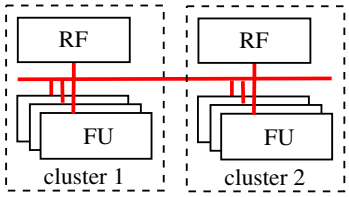


Figure 1. Target Architecture.

to the multi-core systems, data can be communicated across cores very fast, in just a few cycles (from one core's register file to the other). This feature is exploited by CASTED to distribute the error detection overhead across cores in a fine-grain fashion (that is distributing the workload at an instruction level granularity). This can boost performance since the original and replicated code have no true dependency and thus can run in parallel.

Depending on the system setup, the architecture might be configured in many ways. Parameters such as the issue-width, the inter-core delay and the number of available cores can change across designs. The challenge for CASTED is to fully take advantage of the available resources and to effectively distribute the error detection overhead no matter what configuration is used. CASTED uses these parameters to decide whether it is preferable to assign the whole error detection code in one core or it is more efficient to split the code into different cores. This adaptivity in the distribution of the error detection overhead across multiple resources is the main feature of our scheme. In the following section, we will explain why current error detection methodologies fail to do this.

B. Motivating Example

Existing software-based techniques do not map well on tightly coupled cores as shown in the motivating examples in Fig.2 and 3. The proposed approach is the first compiler-based error detection technique that exploits the fast interconnect of architectures with tightly-coupled cores to distribute the error detection overhead across the cores. It is also the first one to fully adapt to the system configuration (issue width of each core, communication cost). As a result the performance achieved is at least as good as the best performing of the existing techniques on any configuration.

The following examples demonstrate our methodology and show how adaptation works. Both examples (Fig.2 and 3) are based on some sample code with the Data Flow Graph (DFG) shown on the left of each figure. This code is referred to as the *original* code. Fig.2.c and Fig.3.c show the DFG of the error detection code. The error detection DFG shows some important attributes of the error-detection code: **i.** The error detection DFG is much larger (in node count) than the original DFG. This is because of a) the numerous replicated instructions (in blue) and b) the check instructions (CHK) just before each non-replicated (N.R.) node. **ii.** Its critical

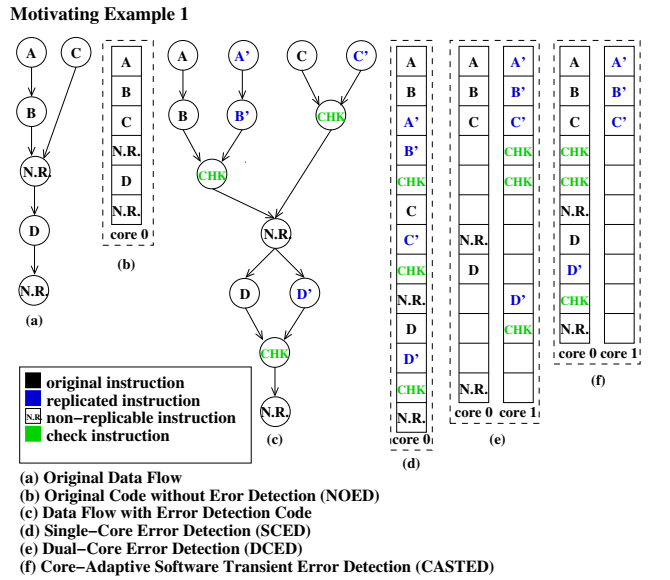


Figure 2. DCED outperforms the resource constrained SCED.

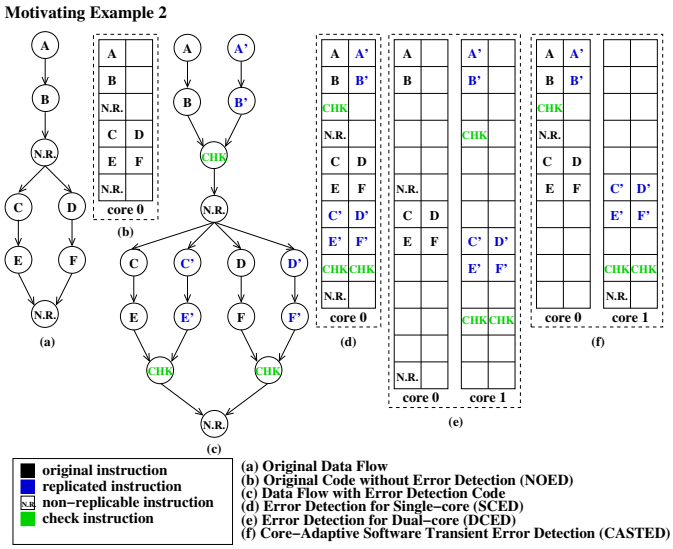


Figure 3. SCED outperforms DCED, which suffers from inter-core communication delay.

path is longer because of the check instructions (CHK). **iii.** Its ILP is higher compared to the original code. This is because the replicated instructions can be executed in parallel with their respective original instructions.

We compare against two non-adaptive methodologies that are influenced by the state-of-the-art for single-threaded and multi-threaded error detection. To quantify the performance of each scheme, we show the corresponding instruction schedule after applying the error detection algorithms (SCED, DCED and CASTED) on our target (Fig.2.d,e,f and Fig.3.d,e,f). The communication delay for this example is set to 1 cycle.

The example shows the schedules of: **i.** The original code (Fig.2.b and Fig.3.b) with no error detection (NOED). **ii.** The Single-Core Error Detection (SCED) approach (Fig.2.d and Fig.3.d) where the original and the replicated codes follow one another alternatively. **iii.** The Dual-Core Error Detection (DCED) approach (Fig.2.e and Fig.3.e) where the original code always runs on one core and the replicated and checking code (redundant code) always runs on the second one. The non-replicated (N.R.) instructions (store and control-flow instructions) are only executed in the main code because it is the only one that can access the memory. The verification is only done by the checker code. **iv.** The proposed Core-Adaptive Error Detection (CASTED) approach (Fig.2.f and Fig.3.f) where instructions of the error detection code (original code, replicated code and checks) are assigned to the cores in an adaptive way. This leads to better resource utilization.

In Example 1, each core is single-issue. This setup might be simplistic but helps us point out the shortcomings of the existing approaches. We observe that the dual-core case (Fig.2.e) outperforms the single-core one (Fig.2.d). This is due to the fact that the single-issue single core is **resource constrained** and as such it can not effectively execute both the original code and the replicated code. The dual-core case, has more resources and performs better. CASTED (Fig.2.f) makes better use of the resources. It assigns the instructions of the original and the replicated code to the first available core. For example, the checks and a part of the replicated code are executed in the main cluster as this will speed up the algorithm.

Example 2 shows a case where the **inter-core delay** can become the performance bottleneck for the dual-core case. In more detail, in Example 2, each core is two-wide issue. We observe that the single-core case (Fig.3.d) outperforms the dual-core (Fig.3.e) for two reasons: **i.** It is wide-issue enough to accommodate the ILP of the error detection code with just a few cycles of overhead. **ii.** The dual-core case suffers from inter-core communication delay due to the sub-optimal fixed placement of the original and checker code instructions on the first and second core respectively. CASTED (Fig.3.f), on the other hand, powered by the adaptive placement of the instructions to the cores, performs as well as the best of the two approaches. This is because it is delay-aware and assigns the instruction to the cores in such a way that the delay does not become the bottleneck. It is worth noting that not only the replicated instructions but also the check instructions are moved across cores, in an attempt to minimize the cycle count.

III. CASTED

CASTED is a software-based technique, implemented in the compiler back-end. It comprises of two algorithms: **(a)** The first one is a single-threaded software error detection algorithm with a high fault-coverage. This generates the

redundant code and the check instructions. **(b)** The second algorithm is the one responsible for the adaptivity of our scheme. It is based on [7] and is the one that assigns the instructions to the cores.

A. Target Architecture

CASTED works on tightly coupled cores such as RAW[32], VOLTRON[37] and VLIW clusters [8]. In this work, we use a clustered VLIW architecture with configurable resources and inter-cluster communication latency (Fig. 1). Both clusters operate in lockstep execution. Each cluster can access the other cluster's register file but this has an increased latency (the inter-cluster communication latency) since it has to go through the interconnect. More details for the configurations are provided in IV-A.

B. Sphere of Replication

CASTED assumes that the memory subsystem (caches included) is protected by its own mechanisms like Error Correcting Code (ECC), parity checking or other mechanisms. Therefore the data fetched from the memory is considered correct. This is why the sphere of replication(SoR) in CASTED is limited within the processor only. This is common practice in the majority of software-based error detection methodologies [9][23][34][36].

The error detection algorithm (Algorithm 1) we used in this work is one that makes a good compromise between performance and reliability.

The instructions that are not replicated are the following:

1) Control Flow instructions (e.g. branches, function calls): the control flow is followed by only one of the cores.
2) Store instructions: We do not replicate store instructions, because this requires memory partition and increases the memory footprint and bandwidth. We guarantee that they will not let corrupted data to be written in the memory by checking their operands before their execution.
3) Compiler-generated instructions.

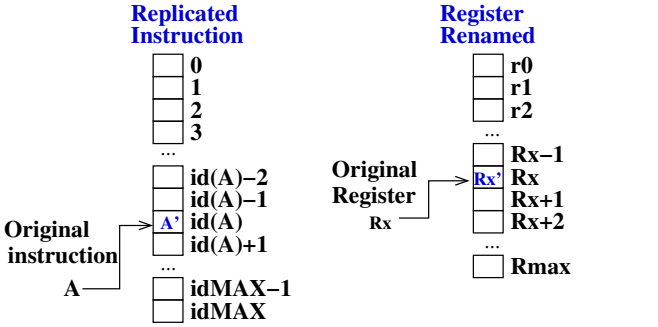
In the paper, we refer to control-flow and store instructions as non-replicated instructions.

CASTED does not replicate the code of the library functions linked into the output when these libraries are supplied as binaries.

C. Error Detection

The Error Detection algorithm in CASTED works as follows:

i. The algorithm (Algorithm 1) first **replicates** the instructions (Algorithm 1, line 10) that are not in one of the above categories. This is done by emitting an exact duplicate of the original instruction. The new replicated instruction is placed just before the original one. Each original instruction that gets replicated has its replicated instruction inserted into a table as shown in Fig.4.a



a. Replicated Instructions Table. **b. Register Renamed Table.**

Figure 4. The table data-structures used by the error detection algorithm.

ii. The next step is the replicated code's **isolation**. The replicated code is isolated from the original one, so that it does not write over the original code's registers. Register isolation does not let the replicated code affect the original code's execution in any way. This is done by register renaming the replicated instructions (Algorithm 1, line 21). In short, the algorithm iterates over all original instructions in the program (line 23,24) and for each of them it retrieves the corresponding replicated instruction from the table (see step 1) (INSN_DUP line 26). Next, it renames all registers written by the replicated instructions along with each of their uses (line 27). All renamed registers are filled into the table data-structure of Fig.4.b. The algorithm uses this data in step iii.

iii. Next, the **checks** are emitted wherever required (Algorithm 1, line 46). This function scans all the instructions and finds all the non-replicated ones. For each one of the non-replicated instructions, it finds the registers read by them. For each register it emits a compare instruction, right before the non-replicated instruction that reads this register. The check is a compare instruction that compares the original register against the corresponding renamed one (it gets it by accessing the data-structure of Fig.4.b). The compare instruction is followed by a jump instruction that diverts the program's execution in case of an error.

D. Adaptivity

CASTED adaptively assigns the code to the available cores, using the Bottom-Up-Greedy (BUG) clustering algorithm [7] (Algorithm 2). As its name suggests, it is a greedy algorithm that makes the clustering decision based on the completion cycle of the instruction into consideration; each instruction gets assigned to the core where it will execute the earliest. The completion cycle heuristic is aware of the inter-cluster delays and can therefore adjust its behavior on any architecture configuration.

In more detail, the algorithm walks through the Data Flow

Algorithm 1. Error Detection Algorithm

```
/*Main function (entry point)*/
relaxed_main ()
{
  replicate_insns ()
  register_rename ()
  emit_check_insns ()
}

/* Check whether an instruction can be replicated. If so, then emmit a copy before the original instruction.*/
replicate_insns ()
{
  for INSN in instructions
     Skip if INSN i) control-flow
                 ii) store
                 iii) special non-replicate
     Emit an exact duplicate of INSN just before it
     Register the duplicate into the data structure
}

/* Code isolation.*/
register_rename (INSN_ORIG,INSN_DUP)
{
  for INSN in instructions
     Skip duplicates
     INSN_ORIG = INSN
     INSN_DUP = duplicate_of (INSN_ORIG)
     rename_writes_and_uses (INSN_ORIG, INSN_DUP)
}

/* Rename the writes of the INSN and its uses.*/
rename_writes_and_uses (INSN_ORIG, INSN_DUP)
{
  for REGW in registers written by INSN_ORIG:
     if (INSN_ORIG has no duplicates)
         Create COPY_INSN: NEW_REG = REGW
         Emit COPY_INSN after INSN_ORIG
         Rename the uses of REGW with NEW_REG for duplicated instructions.
      else
         NEW_REG = the renamed register of REGW if already renamed.
         Register rename the REGW of INSN_DUP to NEW_REG
         Rename the uses of REGW with NEW_REG for duplicated instructions.

}

/* Find and inject the check instructions. */
emit_check_insns ()
{
  for INSN in instructions:
     skip all but the non-replicated instructions.
     for each REG read by INSN:
         Get REG_RENAMED: the renamed REG from the data structure.
         Emit CHECK_INSN before INSN comparing REG with RENAMED_REG.
}
```

Graph (DFG) in a topological order, by giving preference to the instructions in the critical path. For each instruction, it calculates the value of the completion cycle and selects the core that corresponds to the lowest cycle. The completion cycle is resource aware. After the core assignment decision has been made, that specific resource (that is the cycle and the chosen core) is marked as used in the reservation table.

In Fig.2.f, CASTED observes that the execution of the replicated instructions(A', B' and C') in the second cluster is beneficial for performance as the communication latency overlaps with the execution of the checks. Similarly, executing D' in the second cluster is expensive because of the communication delay. Therefore, CASTED places D' in the first cluster. Moreover, contrary to existing schemes, checks can migrate from one cluster to the other when appropriate (Fig.2).

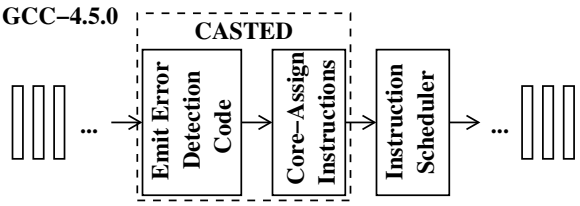


Figure 5. CASTED passes in the back-end of GCC

Unlike the other approaches, CASTED balances the use of hardware resources. For example, Fig.2 shows the execution of non-replicated instructions on both cores. In the case of memory instructions, this improves memory level parallelism (MLP).

Algorithm 2. Bottom-Up-Greedy (BUG) assignment algorithm

```
/* The main function (simplified) of BUG algorithm */
bug (node)
{
  if node is leaf OR node is assigned
     return;
  /* Visit the instructions in topological order giving preference to the critical path */
  for node's predecessor sorted by critical path
      bug (predecessor)

  /* Calculate the completion cycle heuristic */
  sorted cores,sorted cycles=compl_cycle(node)
  /* Assign NODE to CORE and CYCLE */
  node.core = FIRST (best cores)
  node.cycle = FIRST (best cycles)

  /* Reserve issue slots in reservation table */
  reservation set(core,\author{} cycle)
}
```

IV. Evaluation and Analysis

A. Experimental Setup

The CASTED system is implemented as two back-end passes in GCC-4.5.0 [1] compiler infrastructure. We implemented both the error detection and the core-assignment (adaptivity) algorithms in separate passes placed just before the first instruction scheduling pass, as illustrated in Fig.5.

The target architecture is a clustered VLIW with the Itanium2 [25] instruction set. The processor configuration is listed in Table I. We simulate the execution on a modified SKI IA-64 **simulator**[2]. The modified simulator is a cycle-accurate Itanium2 simulator with a full cache memory hierarchy, the same as the one of Itanium2.

Processor: IA64 based clustered VLIW				
Clusters:	2			
Issue width:	configurable			
Instruction Latencies:	configurable			
Register File:	(64GP, 64FL, 32PR) per cluster			
Branch Prediction:	Perfect			
Cache: Levels 3 (same as Itanium2 [17])				
Levels :	L1	L2	L3	Main
Size (Bytes):	16K	256K	3M	∞
Block size (Bytes):	64	128	128	-
Associativity:	4-Way	8-way	12-way	-
Latency (cycles):	1	5	12	150
Non-Blocking:	YES	YES	YES	-

Table I
PROCESSOR CONFIGURATION.

MediaBench2	SPEC CINT2000
cjpeg	175.vpr
h263dec	181.mcf
mpeg2dec	197.parser
h263enc	

Table II
BENCHMARK PROGRAMS

We evaluated our software error detection scheme on 7 benchmarks. 4 are from the Mediabench II video [10] and 3 from the SPEC CINT2000 [13] **benchmarks**, as listed in Table II. We ran the benchmarks to completion. All benchmarks were compiled with optimizations enabled (-O1 flag) and with instruction scheduling enabled. We turned off the late stages of the Common Subexpression Elimination (CSE) and Dead Code Elimination (DCE) optimizations that get called after the CASTED passes. This is common practice ([23]) to prevent these optimizations from removing the replicated code and it has been shown that it has negligible impact on performance (1.5% in the worst case and 0.3% on average in our case). These optimizations are not disabled in the performance evaluation of the No Error Detection code (NOED).

B. Performance Evaluation

We evaluate the performance of CASTED by comparing it against the Single-Core Error Detection (SCED), the Dual-Core Error Detection (DCED) and the single-core No Error Detection (NOED)(this is the unmodified code). The performance results for all benchmarks for various issue widths and inter-core delays are shown in Fig.6 and 7. These results are normalized to NOED for each issue width (that is all issue 1 results are normalized to NOED-issue 1, all issue 2 to NOED-issue 2, etc.).

1) SCED Slowdown: The first observation to be made is the variation in the slowdown of SCED compared to NOED across benchmarks and configurations. It varies from 1.34 to 2.22, and is 1.7 on average. Such variation can be attributed to the variation in the quantity of the error checking code and the variation of register spilling it causes. For example, the more non-duplicated instructions (e.g stores and branches) the code has, the more checks the error detection algorithm adds. In SCED, both the original and the error detection code run in one core. Therefore, the performance is only affected by the issue-width. In general, SCED's performance improves dramatically as the issue width increases. As we explained in section II-A and in the motivating examples of Fig.2 and 3, the redundant code has no dependencies with the original code and can run in parallel in an ILP fashion. Once the resource constraints are no longer the bottleneck, the execution speeds up. In other words, the more available resources we have, the better performance SCED achieves with the exception of h263enc which will be discussed next.

2) SCED Scalability: Fig.8 shows the scaling of NOED, SCED, DCED and CASTED performance as the issue-width increases. This is a metric of the ILP, the steeper the curve, the more the ILP. In most cases, SCED scales better than NOED (Fig.8) which results in a decrease in the SCED-NOED performance difference as the issue-width increases. This can be clearly observed in the majority of benchmarks in Fig.6 and 7. This difference in scaling between SCED and NOED is a measure of the additional ILP of the redundant code. In applications with low ILP (e.g. 181.mcf), the original code (NOED) scales poorly with the issue-width (as there is low ILP). However, SCED scales better than NOED because of the extra ILP.

On the other hand, h263enc (Fig.6) is a benchmark where SCED does not scale as expected. This is because the redundant code has low ILP due to the frequent checking. As shown in Algorithm 1, the checking code consists of compare and jump instructions. Therefore, the more checks the code has, the more sequential the code becomes and according to Amdahl's law the error detection code should scale worse than NOED.

3) DCED Slowdown: The baseline dual-core performance (DCED)(Fig.6 and 7) also varies compared to the performance of NOED across benchmarks and configurations. The slowdown is between 1.31 and 3.32 (2.1 on average). There are two factors that contribute to that. The first one is the issue width. The second and most important factor is the inter-core delay. The bigger the delay, the worse the performance. This is due to the fact that DCED performs regular inter-core communication which becomes a performance bottleneck as the inter-core delay increases.

4) DCED Scalability: The scalability of DCED according to Fig.8 is worse than that of SCED. As explained previously, SCED performs better as the issue-width increases because it spreads instructions across more issue-slots. DCED has a head start. Even at issue 1, it has exploited a large part of the ILP of the redundant code as it executes it in a different core. From that point on, there is little room for improvement. This explains the strange phenomenon where

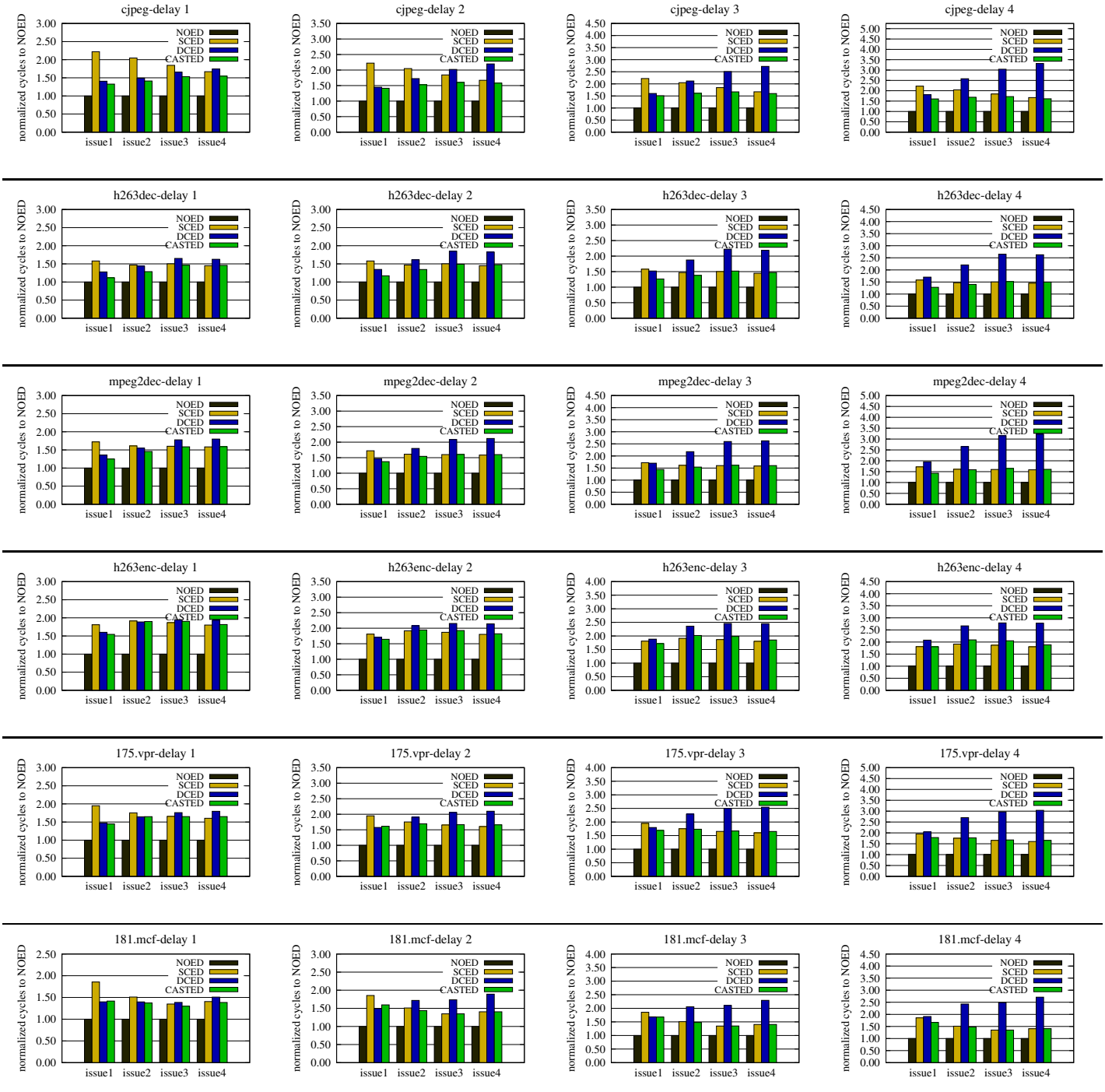


Figure 6. Performance for delays of 1 to 4 and issue-width per cluster in the range of 1 to 4, normalized to NOED for each issue-width (part 1).

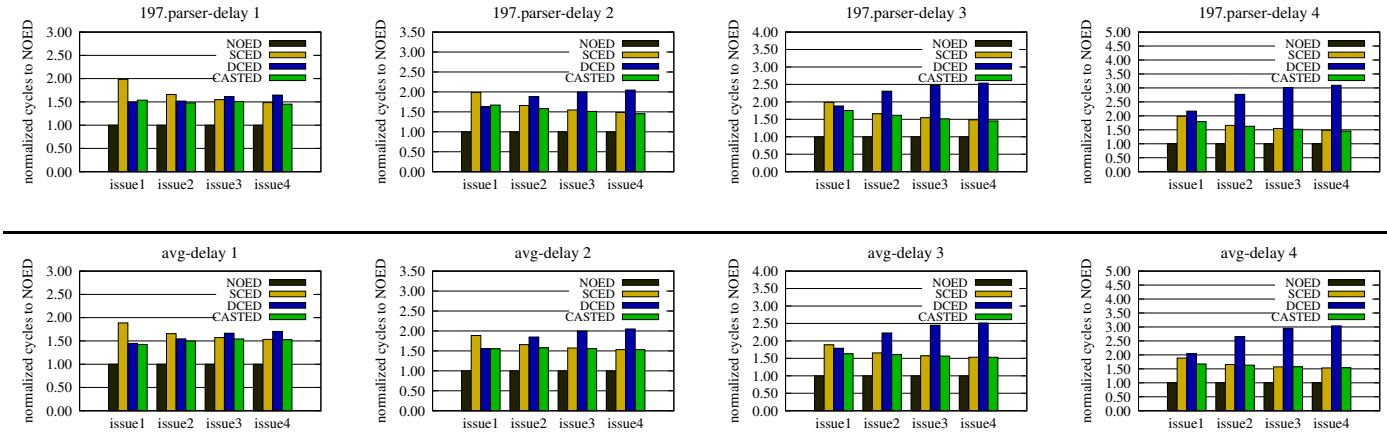


Figure 7. Performance for delays of 1 to 4 and issue-width per cluster in the range of 1 to 4, normalized to NOED for each issue-width (part 2).

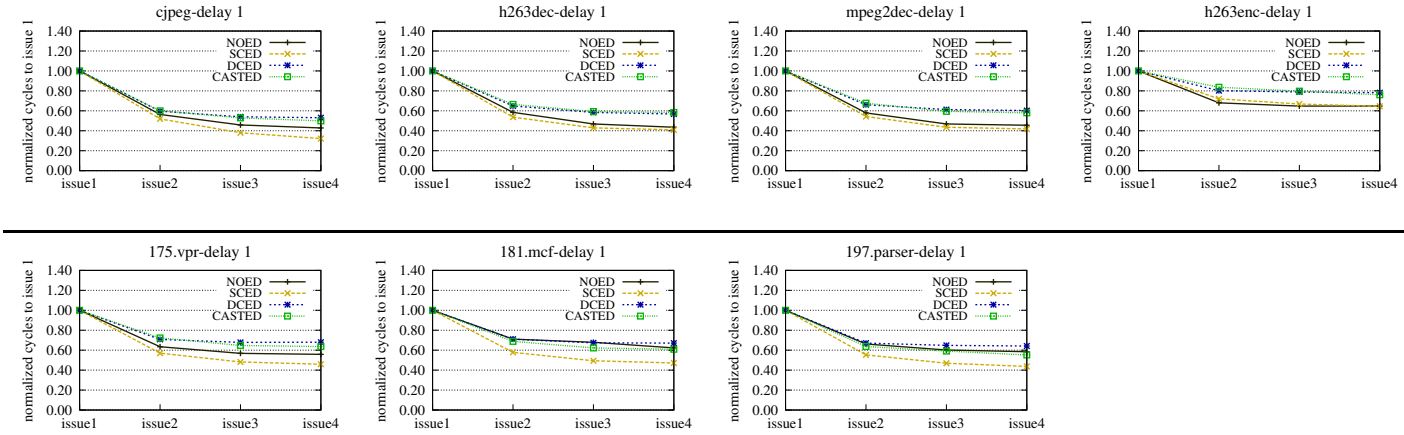


Figure 8. Benchmark ILP scaling.

the overhead of DCED against NOED increases as the issue-width increases in Fig.6 and 7.

5) SCED vs DCED: A more interesting comparison is DCED against SCED. SCED performs better in wider-issue cases because the heavy SCED code expands effectively to the available issue-slots. DCED however, cannot reach these levels of performance, as it always suffers from the inter-core latency upon checks. Things become worse for DCED when the delay is greater than or equal to three. In these cases, the communication cost between the two cores is that big that DCED performs poorly. On the other hand, when the issue-width and the inter-core latency remains low, DCED easily outperforms the resource constrained SCED.

6) CASTED: In the majority of cases, CASTED can at least match the performance of the best performing (SCED or DCED) and in some cases it can even outperform the best. For instance, in Fig.6 h263dec-d1 for issue-width 1, the best non-adaptive is DCED and CASTED behaves similar to this technique. The ability of CASTED to adapt the error detection code in every configuration has a positive impact on its slowdown against NOED. The slowdown varies from 1.19 to 2.1 (1.58 on average). Upon low issue widths, CASTED adapts to DCED which is less resource constrained than SCED.

Furthermore, in some cases CASTED outperform the best non-adaptive. This is because CASTED not only distributes the error detection code across cores (as DCED does) but it also distributes the original code if profitable. This leads to performance improvements of up to 11.4% (in cjpeg for issue 2 delay 2). As the issue-widths and delays increase, DCED is not the preferable method anymore. Instead SCED becomes the most efficient approach. As we can see, at that point, CASTED no longer adapts to DCED, but instead it adapts to SCED. In this case too, CASTED can outperform SCED due to the exploitation of the available resources on the distant core. The performance improvements are up to

21.2% (in cjpeg issue 2 delay 3).

C. Fault Coverage Evaluation

The fault injection results presented in this paper are generated using SKI IA-64 simulator [2]. The simulator was modified to inject errors at the output registers of instructions, which is common practice in the literature [23][36][34].

The fault coverage results are produced with Monte Carlo simulations. Initially, each original binary is profiled in order to count the number of dynamic instructions. Then, the fault injection takes place as follows: a dynamic instruction is randomly selected and one of its outputs is randomly picked for injection and a random bit of the register output is flipped. Errors are injected into general purpose, floating point and predicate registers. This process is repeated 300 times for each benchmark.

Original binaries are injected with one error per run. The binaries that support error detection are much larger (2.4x larger on the average than the originals). A fair comparison between the original code and the error detection code requires keeping the error rate fixed. Thus, the error detection codes are injected with one error per the number of dynamic instructions of the original binary.

The output of each Monte Carlo trial is classified into one of the following five categories:

1) *Benign Errors* (aka as masked errors) are the errors that do not affect program's output and they produce the same output and exit code as the good execution.
2) *Detected* are the errors that CASTED algorithm successfully detects.
3) *Exceptions* are indications of transient errors [35]. Since they can be easily caught by a custom exception handler, they are usually part of the detected errors (as in [36]). In our case however, we show them as a separate type of errors for clarity.
4) *Data Corrupt Errors* are the errors that cause wrong outputs without being detected.
5) *Time out Errors* are the errors that result in infinite execution and they are detected by the time-out feature of our simulator.

Fig.9 verifies that CASTED is as good as other high reliability methodologies. In most of the cases, there are not data-corruption or time-out errors. The presence of data corruption errors after applying CASTED, SCED or DCED is mainly attributed to the fact that these techniques cannot detect errors that occur in the system's library functions since the compiler does not have access to the library source codes to protect them. On the contrary, in some related work ([9][23][34][36]) system libraries are excluded from fault injection, which is somewhat unrealistic. If the source code of the system libraries is available, they can also be compiled with CASTED and be protected against transient errors.

Another interesting point extracted from Fig.9 is that encoding benchmarks (cjpeg, h263enc) are less prone to errors. This is intuitive as there is some data compression (masking) involved. Finally, we observe that most of the errors are exceptions. This is desirable since exceptions can be easily detected by an exception handler.

In Fig.10, it is shown how CASTED error detection algorithm behaves under different architecture configurations for the h263dec benchmark. The fault coverage, as expected, is not affected by the underlying architecture configuration and CASTED retains the same level of reliability. The variation in fault-coverage results is mainly attributed to statistical deviation. Overall, Fig.6,7 and Fig.9,10 validate our previous claim that CASTED can adapt to different architecture configurations without any impact on reliability.

V. Related Work

Code redundancy can take various forms: instruction, thread and process redundancy. EDDI[20] and SWIFT[23] are two techniques based on **instruction**-level redundancy where the error detection is done by code duplication. The main advantage of these approaches is that they provide error detection with no additional hardware support. However, the replicated code and the checking code have significant impact on performance. In [9] and [14], they tackle this problem by reducing the number of replicated instructions. Shoestring[9] is based on the assumption that some transient errors can be detected by the operating system. As a result, a decent amount of instructions need not be replicated or checked. In [14], the number of replicated instructions is selected according to the desired threshold of performance degradation (or fault-coverage). SRMT[34] has its redundant code in a second thread which runs on a different core from the main thread. In these approaches, the communication overhead becomes performance bottleneck. DAFT[36] improves this by decoupling the execution of the main thread from the checker. All schemes are summarized and compared against CASTED in Table III.

Process level redundancy was pioneered by Shye[26] and optimized by Zhang[11]. The application gets launched twice and the two processes run simultaneously. An error is detected upon program termination or by faulty I/O operations of any of the processes. Process level redundancy has small overhead since it checks less values than other techniques, but this comes with the cost of maintaining multiple memory states.

Thread level redundancy was introduced by AR-SMT [24]. This work proposed the idea of redundant multi-threading (RMT) on SMT cores. The active thread executes the program and puts its results on a delay buffer. The redundant thread re-executes the same instruction stream and compares the results that it produces with the ones from the delay buffer. The committed state of the redundant thread is also used as a recovery checkpoint.

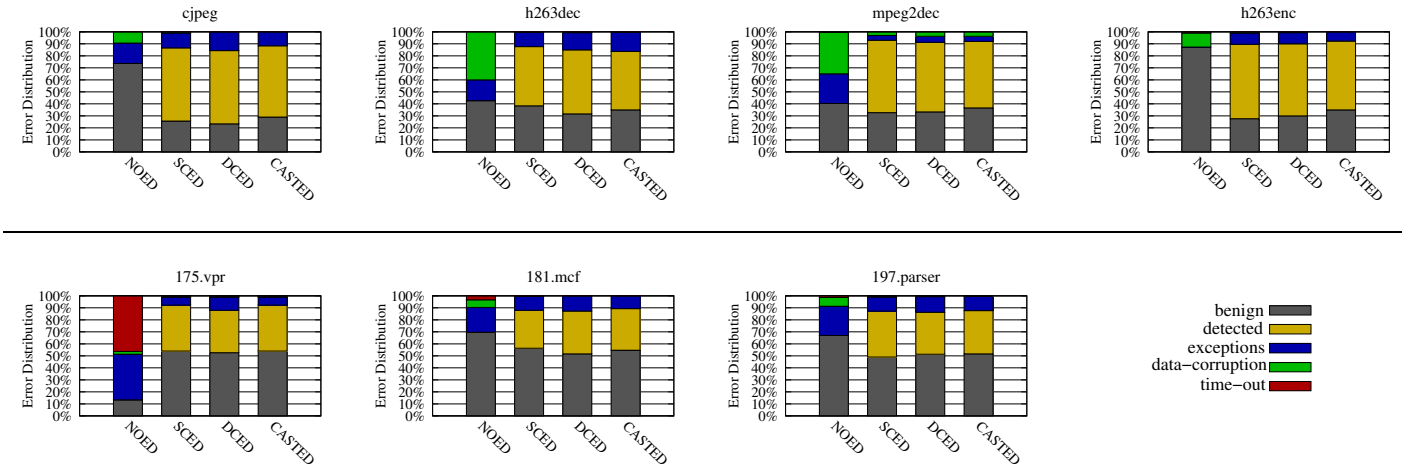


Figure 9. Fault-coverage for all benchmarks for issue-width=2 and delay=2.

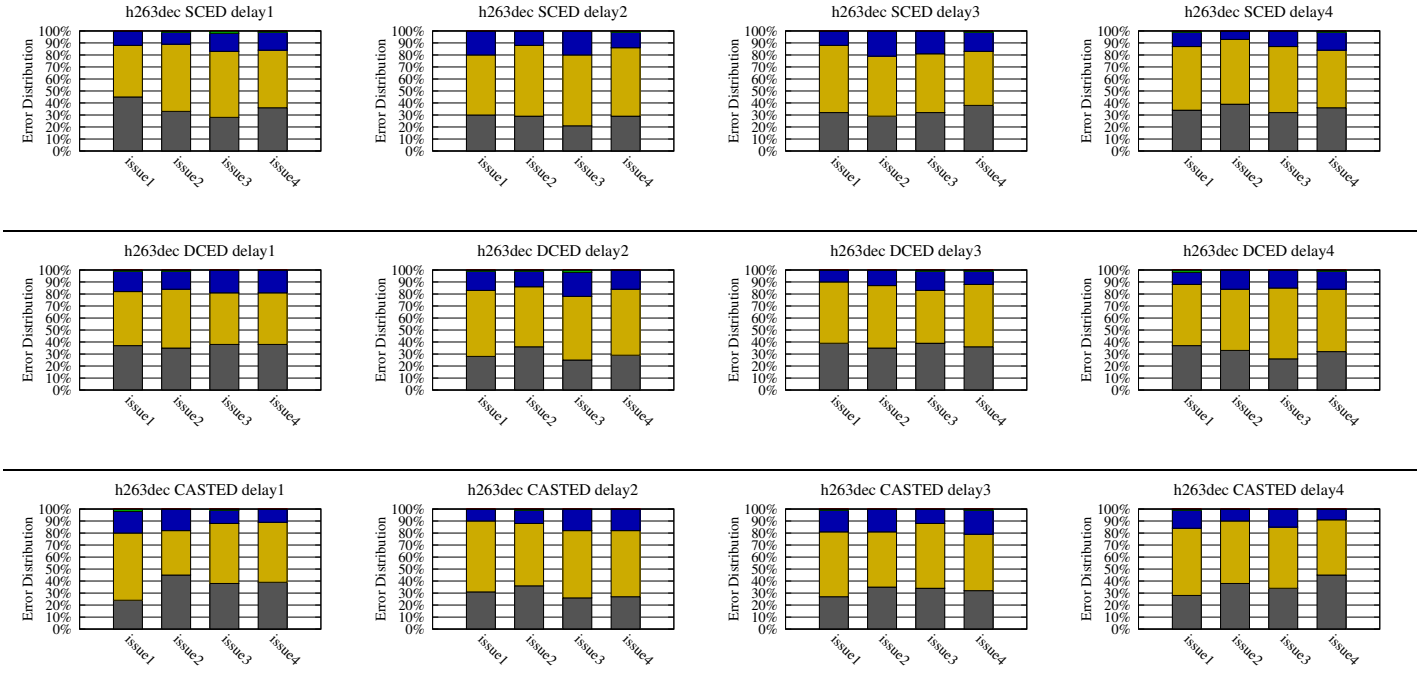


Figure 10. The fault-coverage of h263dec benchmark for NOED,SCED,DCED and CASTED for issue 1 to 4 and delay 1 to 4 .

	Speed up Factors	**Target architecture**	**Code placement**
EDDI[20]	-	wide single-core	fixed
SWIFT[23]	reduction of checking points	wide single-core	fixed
SHOESTRING[9]	partial redundancy	single-core	fixed
Compiler-assisted ED [14]	partial redundancy	single-core	fixed
SRMT[34]	partially synchronized threads	dual-core	fixed
DAFT[36]	decoupled threads	dual-core	fixed
CASTED	adaptivity	tightly-coupled cores	adaptive

Table III
COMPILER-BASED ERROR DETECTION SCHEMES.

There are several works that are based on AR-SMT and extend it. [22] introduces Simultaneous and Redundant Threaded (SRT) processors that take advantage of an SMT processor's extra thread contexts. Similarly, Mukherjee[19] used the SMT idea on CMPs proposing Chip-level Redundant Threading (CRT). SRTR[33] and CRTR[12] are extended versions of the latter two methodologies in the sense that they include recovery mechanisms. La Frieda[15] and Smolens[28] presented techniques that exploit the idle cores for redundant thread execution. The main disadvantage of redundant multi-threading is that it reduces the system's total throughput because it occupies more thread contexts and hardware resources.

In **hardware** error detection, correctness is checked on hardware. Hardware-based designs include the watchdog processors by McCluskey[16] and by Austin[4]. The main idea is that a smaller and simpler in design processor, which is considered safer, follows the execution of the main processor.

Commercial high-reliable processor systems are designed with hardware redundancy, too. G5[27][30], is designed with replicated instruction and data execution units that have their outcome checked. HP NonStop series processors [5] are designed in TMR (triple-modular redundancy) and hardware voters guarantee the correct execution of the program. In Power 6 [21] and in Fujitsu's SPARC64 [3], they use parity and residue checking to protect their systems against transient errors.

VI. Conclusion

This paper introduces CASTED, a novel software-based error detection scheme for architectures with tightly-coupled cores. CASTED effectively distributes the impact of the error detection overhead across the available resources and successfully adapts to the requirements of each configuration. This improves performance without affecting the fault coverage across the architecture configurations. It reduces the overall slowdown by 7.5% against the single-core error detection and 24.7% against the dual-core case.

References

[1] GCC: GNU compiler collection. *http://gcc.gnu.org*.

[2] SKI, An IA64 instruction set simulator. *http://ski.sourceforge.net*.

[3] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, et al. A 1.3-GHz Fifth Generation SPARC64 Microprocessor. *DAC*,2003.

[4] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *MICRO*, 1999.

[5] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. *DSN*, 2005.

[6] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *MICRO*, 2003.

[7] J. Ellis. Bulldog: A compiler for VLIW architectures. *Technical report, Yale Univ.*, 1985.

[8] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. *Technical report, Hewlett Packard Laboratories*, 1999.

[9] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. *ASPLOS*, 2010.

[10] J. Fritts, F. Steiling, and J. Tucek. Mediabench II Video: Expediting the next generation of video systems research. *SPIE*, 2005.

[11] Y. Ghosh, J. Huang, J. Lee, S. Mahlke, and D. August. Runtime asynchronous fault tolerance via speculation. *CGO*, 2012.

[12] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *ISCA*, 2003.

[13] J. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 2000.

[14] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Transactions on Embedded Computing Systems*, 2009.

[15] C. LaFrieda, E. Ipek, J. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. *DSN*, 2007.

[16] A. Mahmood and E. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 1988.

[17] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *MICRO*, 2003.

[18] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the number of fatal soft errors in Los Alamos national laboratory's supercomputer. *IEEE Transactions on Device and Materials Reliability*, 2005.

[19] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *ISCA*, 2002.

[20] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 2002.

[21] K. Reick, P. Sanda, S. Swaney, J. Kellington, M. Mack, M. Floyd, and D. Henderson. Fault-tolerant design of the IBM Power6 microprocessor. *MICRO*, 2008.

[22] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. *ACM SIGARCH Computer Architecture News*, 2000.

[23] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software implemented fault tolerance. *CGO*, 2005.

[24] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *International Symposium on Fault-Tolerant Computing*, 1999.

[25] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE MICRO*, 2000.

[26] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. *DSN*, 2007.

[27] T. Slegel et al. Ibm's s/390 G5 microprocessor design. *MICRO*, 1999.

[28] J. Smolens, B. Gold, B. Falsafi, and J. Hoe. Reunion: Complexity-effective multicore redundancy. *MICRO*, 2006.

[29] D. Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers*, 2009.

[30] L. Spainhower and T. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 1999.

[31] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. *DSN*, 2004.

[32] M. Taylor, J. Kim, et al. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *MICRO*, 2002.

[33] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. *ISCA*, 2002.

[34] C. Wang, H. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. *CGO*, 2007.

[35] N. Wang and S. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 2006.

[36] Y. Zhang, J. Lee, N. Johnson, and D. August. DAFT: Decoupled Acyclic Fault Tolerance. *PACT*, 2010.

[37] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. *HPCA*, 2007.