FluidCheck: A Redundant Threading based Approach for Reliable Execution in Manycore Processors

RAJSHEKAR KALAYAPPAN, Indian Institute of Technology Delhi
SMRUTI R. SARANGI, Indian Institute of Technology Delhi

Soft errors have become a serious cause of concern with reducing feature sizes. The ability to accommodate complex, SMT cores on a single chip presents a unique opportunity to achieve reliable execution, safe from soft errors, with low performance penalties. In this context, we present *FluidCheck*, a checker architecture that allows highly flexible assignment and migration of checking duties across cores. In this paper, we present a mechanism to dynamically use the resources of SMT cores for checking the results of other threads, and propose a variety of heuristics for migration of such *checker threads* across cores. Secondly, to make the process of checking more efficient, we propose a set of architectural enhancements that reduce power consumption, decrease the length of the critical path, and reduce the load on the NoC. Based on our observations, we design a 16 core system for running SPEC2006 based bag-of-tasks applications. Our experiments demonstrate that fully reliable execution can be attained with a mere 27% slowdown, surpassing traditional redundant threading based techniques by roughly 42%.

CCS Concepts: •**Computer systems organization** → **Reliability; Processors and memory architectures; Redundancy;**

Additional Key Words and Phrases: Reliability, Checker architectures, Redundant multi-threading

ACM Reference Format:
Rajshekar Kalayappan and Smruti R. Sarangi, 2015. FluidCheck: A Redundant Threading based Approach for Reliable Execution in Manycore Processors. *ACM Trans. Architec. Code Optim.* V, N, Article 1 (January 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

For[1] the last 15 years, the designing of processors that have reduced susceptibility to soft errors is an area of active research. There is a wealth of research in creating radiation hardened circuits [Montesinos et al. 2007], and architectures to detect and correct soft errors (refer to the survey by Kalayappan et al. [Kalayappan and Sarangi 2013]). Unfortunately, techniques at the circuit level are still considered fairly intrusive, are known to increase area and power consumption disproportionately, and also provide insufficient error coverage. Consequently, efforts in both industry and academia have focused on creating novel processor architectures that detect soft errors by using redundant computational units such as additional threads or cores. The field has come a long way since the early proposals such as the *dynamic implementation verification architecture* (DIVA) [Austin 1999], *simultaneous redundant threading* (SRT) [Reinhardt and Mukherjee 2000] and *Active-stream/Redundant-stream Simultaneous Multithreading* (AR-SMT) [Rotenberg 1999]. However, the basic design

[1]New Paper, Not an Extension of a Conference Paper

Rajshekar Kalayappan and Smruti R. Sarangi are with the Department of Computer Science & Engineering, Indian Institute of Technology Delhi, New Delhi – 110016. E-mail: {rajshekark, srsarangi}@cse.iitd.ac.in
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 ACM. 1544-3566/2015/01-ART1 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

methodology is still the same. For each application thread, we create two threads: a *leader* and a *checker*. The leader thread executes all the instructions and supplies hints to the checker thread such as memory values, and branch outcomes. The checker thread checks either all or a subset of the instructions executed by the leader. It utilizes the hints to speed up its execution and reduce power. Research work has mainly focused on designing effective architectures to reduce the communication between the leader and checker [Subramanyan et al. 2010], increase the performance of the overall system [Chatterjee et al. 2000], and reduce power [Rashid et al. 2005].

To the best of our knowledge, the problem of designing a checker architecture for manycore processors (16+ cores) where each core can simultaneously run multiple threads is still open. We have tried to solve this problem by introducing a novel method called *FluidCheck*. We begin by explaining the basic features of our design in the context of a 16 core processor, where each core supports 4 way-SMT. Note that it is not necessary to run 4 threads on a core to realize its potential. An SMT processor optimally partitions the issue slots among its executing threads to maximize the utilization of resources. For example, if there are 2 threads running on a core, each thread will on an average get half of the issue slots. Now, if we want to run 24 applications, then we will have 24 leader threads, and 24 checker threads. Scheduling these 48 threads on 16 cores with the aim of maximizing the mean performance of a suite of applications is a very difficult problem. We need to ensure that high IPC leaders are not paired with high IPC checkers on the same core, there are as few thread migrations as possible, and no thread gets unnecessarily slowed down. We propose several scheduling algorithms to solve this problem. Additionally, in this scenario 24 leaders need to send *hints* to their checkers via the on-chip network (NoC). We need to minimize the information sent from leaders to checkers such that there are no traffic bottlenecks in the NoC, and the network energy (which can be significant) is minimized. We propose novel forwarding filters in this paper that reduce the amount of traffic.

We perform an exhaustive evaluation for different load scenarios with bag-of-tasks based applications from the SPEC CPU2006 benchmark suite. We show that FluidCheck achieves reliable execution with a performance penalty of 27.51% (under average load conditions), which surpasses the performance of traditional redundant threading based techniques by 42%. The energy consumed was also studied and found to be similar to the seminal proposals in this field (around 91% more than the unreliable execution case). The NoC bandwidth usage was found to be modest, with an average activity rate of around 0.56 flits/core/cycle.

We discuss related work in Section 2, look at the characteristics of applications and derive insights in Section 3, show our implementation in Sections 3.2 and 4, discuss the heuristics governing the dynamic migration of checker threads in Section 5, and show our evaluation results in Section 6.

2. RELATED WORK AND BACKGROUND

We can categorize checker architectures with respect to three important criteria as shown in Figure 1. First, we can have redundancy at either the core level (a separate core, or a dedicated checker), thread level (separate thread) or instruction level. Second, we can either aim to have full coverage (all the instructions are verified for correct outputs), or partial coverage (a subset of instructions are verified). Lastly, the different redundant entities can be fully *independent* of each other (traditional DMR), or one of them (known as the *checker*) can lag behind the other (*leader*), and be *assisted* by the latter with branch hints, and memory values.

We now look at the design space of checker architectures that is the most relevant to this paper. FluidCheck is a solution at the core level as well as the thread level. This is because the leader and the checker threads need not be running on the same core.

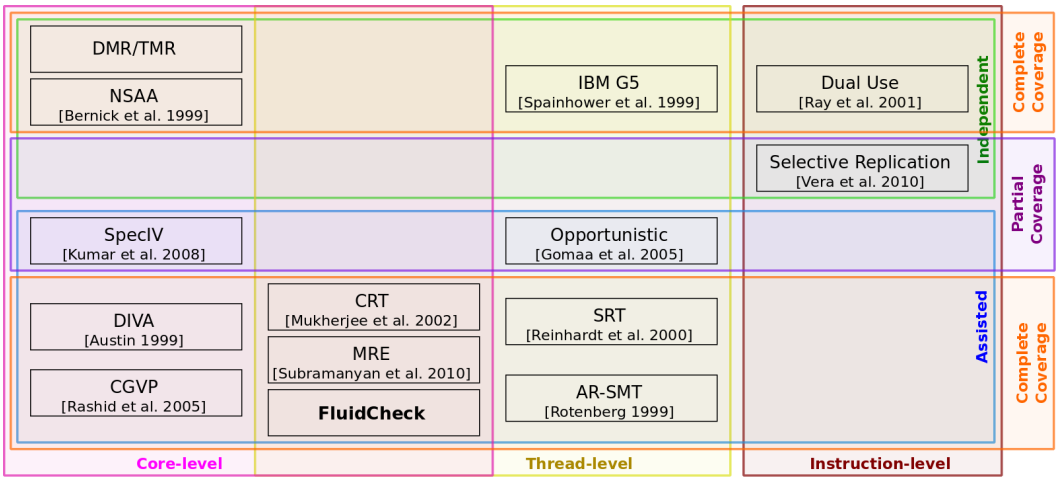


Fig. 1: Related work

Checker threads most of the time run on separate SMT capable cores. Our solution guarantees full coverage, and the leader threads assist the checker threads. As per Figure 1, the only other paper that fits all of these criteria is CRT [Mukherjee et al. 2002]. We shall thus discuss CRT and related proposals that use assisted checking with full coverage first. We shall then briefly describe more conceptually distant classes that use independent checkers, or provide partial coverage.

2.1. Assisted Checking with Full Coverage

DIVA [Austin 1999] is one of the seminal works in assisted checking. Here, a checker processor is attached to the end of a regular processor's (leader's) pipeline. The role of the checker is to re-execute each and every instruction and verify the results computed by the leader. Both the computation (ALU operations), and the communication (data flow across instructions) are checked. Whenever, there is a discrepancy the leader flushes its pipeline. Later variants of DIVA [Chatterjee et al. 2000] proposed several performance enhancing optimizations such as a small L0 cache to to aid the checker, a small checker register file and a checker store queue.

Instead of having dedicated checker hardware, alternative approaches advocated using an additional thread to perform the role of checking in multithreaded processors. The IBM G5 initially used such an approach and compared results every cycle. However, in modern SMT processors it is difficult to ensure that the leader and checker run in lockstep. Alternatively, (AR-SMT) [Rotenberg 1999] and *Simultaneous Redundant Threading* (SRT) [Reinhardt and Mukherjee 2000] propose to have a time lag between the leader and checker threads. This is done to allow the leader to warm up the caches for the checker, and also supply it branch hints. This optimization increases the performance of the system as a whole. Note that all of these early approaches including DIVA were limited to a single core.

Subramanyan et al. [Subramanyan et al. 2010] propose MRE where a single checker checks multiple leaders. This is possible in scenarios where leaders have a very low IPC, and the IPC of the checker is high because of the hints that it receives. Hukerikar et al. [Hukerikar et al. 2014] also propose a similar scheduling heuristic. These two works propose to partition the set of available cores into two categories: leader cores and checker cores. As the names suggest, the leader threads run on the leader cores and the checker threads run on the checker cores. We believe that this is an unneces-

sary restriction, and hence treat all cores alike. These two works are in fact a subset of the set of schedules that the FluidCheck framework allows.

CRT (chip level redundant threading) [Mukherjee et al. 2002] is the most relevant to our work. It tries to extend redundant threading to multiple cores. In the original paper, the authors present a solution for a two core system that is to reliably execute two applications. The leader of application A is scheduled along with the checker of application B on core 1. The leader of application B is scheduled along with the checker of application A on core B. Such a strategy serves to reduce the high occurrence of structural hazards seen in SRT. To pass data between the cores, the authors propose dedicated structures to buffer load values and branch outcomes.

2.2. Independent Checking and Partial Coverage

Traditionally, dual-modular and triple-modular redundancy techniques are used with independent checker units. Since running processors in lockstep is difficult, HP NonStop systems [Bernick et al. 2005] slightly relax the requirement by having synchronization between processors on only I/O operations. In comparison, assisted checking allows for faster checker threads because the checker threads have a significantly higher IPC due to the passage of hints to it. Since checker threads suffer from a lesser number of cache misses, and branch mispredictions, they are also more power efficient with assisted checking as compared to independent checking. Researchers have also proposed checkers [Vera et al. 2010; Gomaa and Vijaykumar 2005; Kumar and Aggarwal 2008] that check only a subset of instructions. Warped-DMR [Jeon and Annavaram 2012] also recommends opportunistic redundancy, but in the realm of GPGPUs. [Wadden et al. 2014] further explores redundant multi-threading to improve reliability in GPU kernels.

The Slipstream [Sundaramoorthy et al. 2000] class of leader-follower architectures focuses on enhancing performance. A reduced or pruned version of the program forms the leader, and the checker runs the original program. The leader's execution is corrected by the checker when the former's control flow deviates from the correct path. The checker is assisted by the leader, and is hence able to keep pace with the latter. This results in a system with a higher performance. Additionally, since a subset of the instructions are redundantly executed, any transient error affecting these can be detected. DCE_FR [Zhou 2006] is an extension of the Slipstream design that provides full coverage. Here, those instructions that are skipped (marked invalid, according to the terminology in [Zhou 2006]) by the leader, are duplicated and executed twice by the checker. This ensures that all instructions are covered.

The FluidCheck architecture can be used with the various techniques that provide partial coverage. However, in this paper we choose to focus on a solution providing complete coverage.

2.3. Novelty and Contributions

None of the works mentioned in Figure 1 are designed for systems with 16 cores and beyond, where each core is capable of simultaneously running multiple threads. We tried to extend two of the most popular schemes namely SRT and CRT to scale to 16 cores (see Section 6). The results were however not very promising. As a result there was a need to design a new method to support redundant threading in large multicore processors with 16 cores and more. We believe that this has been an open problem up till now. Our specific contributions are as follows.

(1) We design an architecture that allows leaders and checkers to be scheduled across different cores in a manycore processor. Our scheme allows dynamic migration of leader and checker threads across cores. This feature has not been implemented in prior work notably CRT.

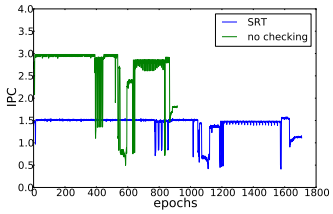


Fig. 2: *calculix* IPC study

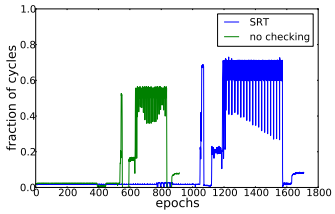


Fig. 3: *calculix* FU hazard study

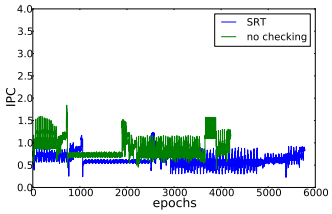


Fig. 4: *gcc* IPC study

(2) Our novel scheduling algorithms are tailored for dealing with two classes of threads: leaders and checkers. We show that it is beneficial to treat these classes differently. We propose novel pairing mechanisms of leaders and checkers belonging to different applications. Since prior work has considered much smaller systems ($\leq$ 2 cores), there was previously no necessity to design such scheduling mechanisms.
(3) Novel optimizations to reduce traffic between cores and the memory system: selective forwarding of cache lines, and forwarding filters (LFB and RFB).
(4) While implementing FluidCheck, we were bedeviled by subtle corner cases such as livelocks, deadlocks, and thrashing due to frequent migrations. Similar issues were also faced in the design of CRT. Effective reservation-based solutions (inspired by CRT) are adopted to solve all these problems.
(5) Lastly, we show an exhaustive evaluation for different load configurations with 100+ combinations of benchmarks. We show that FluidCheck, on an average, is 27.47% faster than solutions similar to CRT, and 42.29% faster than traditional redundant threading mechanisms such as SRT and AR-SMT.

3. MOTIVATION

We motivate our design by considering a small example, where we shall look at two representative SpecInt benchmarks: *gcc* and *calculix*. Further, let us consider a 2 core system, where each core is an SMT processor.

3.1. Study of Benchmarks: *calculix* and *gcc*

Let us now see what happens if we run the leader and checker of *calculix* on the same core using the SRT scheme [Reinhardt and Mukherjee 2000]. Figure 2 compares the instantaneous IPC (IPC averaged over a 100k cycle epoch) for two configurations: *no checking*(only leader thread running), and SRT (with one redundant checker thread). We observe that SRT is 86% slower than the unchecked run.

The main reason for the slowdown in this case is the contention for structures in the out-of-order pipeline between threads. Many phases of the *calculix* benchmark display an IPC, which is more than half of the pipeline width. In this case, we shall have slowdowns due to contention. As an example, Figure 3 shows the fraction of cycles in which instructions are ready but are not able to find a free functional unit. We can see that the number of such structural hazards is much more for the SRT scheme. Another contributor for the slowdown of SRT is the load-store queue (LSQ). We have more decode stalls because the LSQ is full. Due to reductions in IPC, there is a lower chance of a producer store and a consumer load being in the pipeline at the same time. This reduces the number of store$\rightarrow$load forwardings in the LSQ. The last contributor is the decreased per thread bandwidth to the data cache, and destructive interference in the memory hierarchy.

Now, let us consider the *gcc* benchmark. Figure 4 shows that the slowdown here is markedly lesser (37%). *gcc* inherently is a low IPC benchmark; hence, the number of structural hazards is much lower. The reason that *gcc* has a low IPC is because it has a low branch prediction rate. This increases the number of pipeline flushes. When the pipeline is flushed, there is a window of time, when most of the resources in a pipeline are fairly empty. This window of time is utilized by the checker thread.

3.2. Overview of the Proposed Scheme

We now use the intuitions gained in the previous section (Section 3) to create a redundant threading architecture that tries to run our two benchmarks namely *calculix* and *gcc*. We observed in the previous section that $calculix$ is a more demanding (higher IPC) benchmark as compared to gcc. We can naively combine the checker of gcc with the leader of $calculix$, and the leader of *gcc* with the checker of *calculix* (similar to CRT [Mukherjee et al. 2002]). However, this is not necessarily the best combination. Since *gcc* has frequent pipeline flushes due to branch mispredictions, there are large intervals of time when the pipeline is fairly unoccupied by instructions. We can use these windows of time to execute as many instructions from other threads as possible. Such a choice will not affect the IPC of the leader thread of *gcc*. In fact we have experimentally observed that by running the leader thread of *gcc* along with the checker threads of both *gcc* and *calculix* on the same core, and leaving the other core free to exclusively run the leader thread of *calculix*, turned out to be a faster solution. The main insight here is that it is not necessary to have any preset formula for distributing leaders and checker threads across the cores like CRT. Instead, we need to find out the specific combination that works best for a given set of benchmarks at a particular point of time.

This reasoning can be extended to a bigger suite of benchmarks. All kinds of combinations are possible. For example, it might be the case that running all the leaders separately and running all the checkers separately is the best mapping. The essential point of this discussion is that there are a myriad of ways in which threads can be scheduled to cores, and to maximize performance, our architecture needs to support all kinds of combinations of leaders and checkers on the cores. Moreover, programs have phases and the behavior across phases can be distinctly different. As a result, the most efficient mapping of threads to cores will change over time. It is thus necessary to periodically reschedule the threads (both leaders and checkers). Clearly, there are other considerations also such as the cost of process migration, bottlenecks in the NoC, and power consumption. FluidCheck takes these constraints into account and supports arbitrary combinations of leader and checker threads on cores. Moreover, it also periodically recomputes the mappings, and migrates threads across cores.

Figure 5 illustrates some interesting scenarios that occur in the course of the system's progress. Four single-threaded programs that need to be reliably executed, are initially run on a four-core SMT system. Panel (1) indicates a particular valid state that the system can be in. *L1* (in the figure) is the leader thread of the first program, and it is scheduled on core *A*, and its checker thread is scheduled on core *D* and is denoted by *C1*. The same notation is used to specify the schedule of the leader and checker threads of the other programs. As can be seen, the checker thread of a program can run on any of the available cores, including the same core as the leader itself (as in the case of program 4).

Panel (2) describes a scenario where the checker core is unable to keep up with the leader thread. Assume that core *D* decides to evict the checker thread, *C1*. Core *D* informs core *A* about this, which in turn, requests the central arbiter to assign it a different checker core, as illustrated in panel (3). The arbiter selects the best possible checker, core *C* in this example, from among the available cores and replies to core *A*.

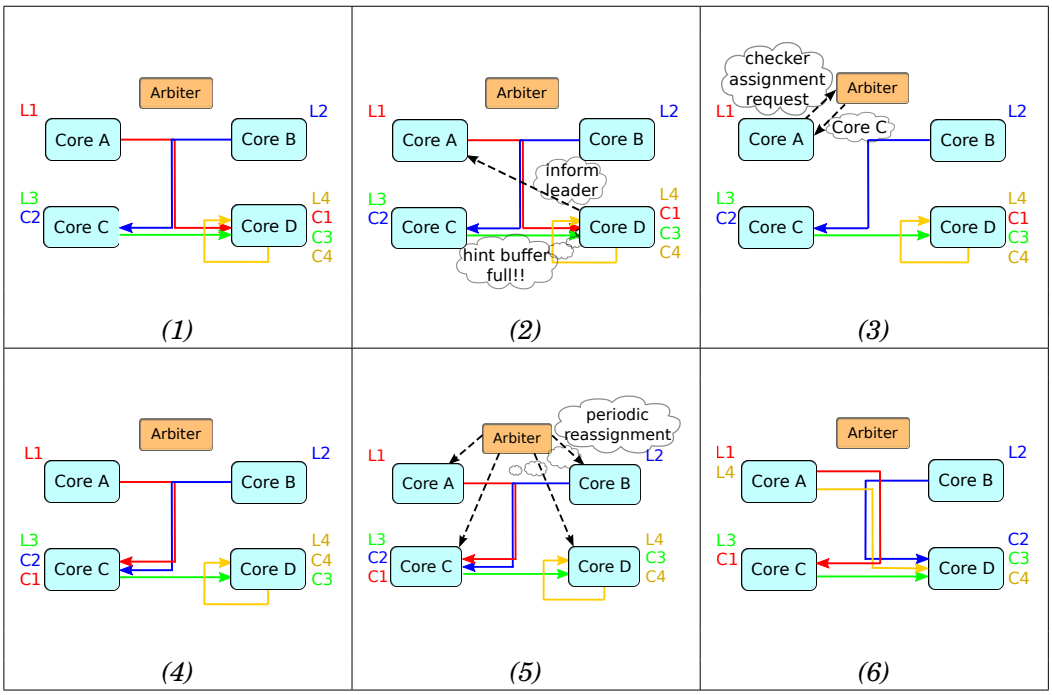


Fig. 5: Illustration of the proposed system

Core *A* sets up the relevant context at core *C*, and waits until the checker thread at core *D* finishes, before resuming its execution of thread *L1* (panel (4)).

The arbiter periodically reconsiders the schedule of threads, as shown in panel (5). This is done to accommodate phase changes. A new schedule after this shuffling is shown in panel (6).

4. ARCHITECTURE

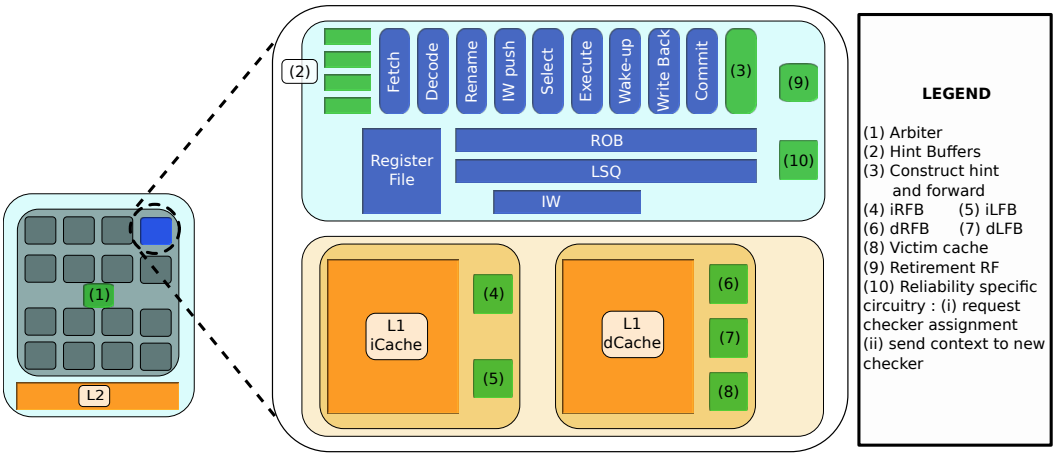


Fig. 6: Proposed architecture

4.1. Description of Physical Structures

Figure 6 shows the architecture of the entire system. The additional structures required are colored green and numbered. The key component is a centralized arbiter that maps the threads to cores. It is connected to the NoC. Each n-way SMT core contains n *hint buffers* (HB) to store the hints for the checker threads that the core might be running. We also have a small circuit at the end of the ROB to construct the hint packets. We have a host of structures (iRFB, iLFB, dRFB, dLFB) for forwarding cache lines from the leader to the checker. Additionally, to maintain register checkpoints, we have a retirement register file that is written to by the checker thread after successful verification of an instruction. At the L1 data cache, along with additional bits in each line, we have a victim cache for holding evicted lines with unchecked data (see Section 4.3.2). The reliability enhancing structures such as the arbiter, and the retirement RF are single points of failure. However, they can be radiation hardened by using a variety of circuit level techniques, or by using traditional TMR based redundancy techniques. Since they are not on the critical path, their latency is not a major concern.

4.2. Overview of Redundant Execution

Our model of assisted execution is largely inspired by DIVA [Austin 1999]. Each ROB entry is tagged with the thread id, and a single bit representing whether it is a leader or a checker. When a leader thread commits an instruction, the core creates a packet containing the branch outcome, operand values, and the contents of the instruction. This packet, tagged with the corresponding checker thread id is sent over the NoC to the assigned checker, which places it in the appropriate hint buffer. The instruction then updates the register state of the leader, and leaves the ROB. At this point we allow leader stores to propagate up to the private L1 cache, but not beyond it. If the NoC is unable to accept the request due to high traffic, the leader thread stalls.

The core executing the checker thread processes the packets in the hint buffer. For each hint packet, it extracts the PC, and then fetches the instruction again. After verifying its contents with the contents field in the hint packet, it proceeds to execute it, utilizing the received hints to resolve data and branch hazards. At the commit stage, the *computation* and *communication* are checked similar to DIVA [Austin 1999]. In specific, we check the result of the instruction, values of the operands (register/memory), branch outcome (if the instruction is a branch), and the value to be stored in memory. If there is any discrepancy, the leader thread is informed, and the checker helps it roll back to a safe checkpoint. The verification of store instructions, and the marking of cache lines as non-speculative (verified) at the leader is elaborated in Section 4.3. We now outline each of the steps in more detail.

4.2.1. Communication between the Leader and the Checker. For creating a checker thread on a core, the leader sends a sequence of flits to the checker core (see Figure 7). It first sends a *START* packet, then a sequence of *CONTEXT* packets containing a snapshot of its architectural register file and the starting PC, and finally sends an *CONTEXT-END* packet. The checker core then starts to execute the checker thread. During the execution of the checker thread, if the associated hint buffer fills up, it intimates the leader core of its inability to keep up by sending a *HELP* packet. The leader core sends it a *STOP* packet and stops sending further hints to it, and waits for the checker thread to drain its hint buffer. Meanwhile, the leader asks the arbiter for a new checker, and sets up the context at it through *START* and *CONTEXT* messages. The sending of hints to the new checker is begun only after the old checker completes checking of all the hints sent to it. If the old checker reports a discrepancy, the new checker is notified to abort its operation by the leader.

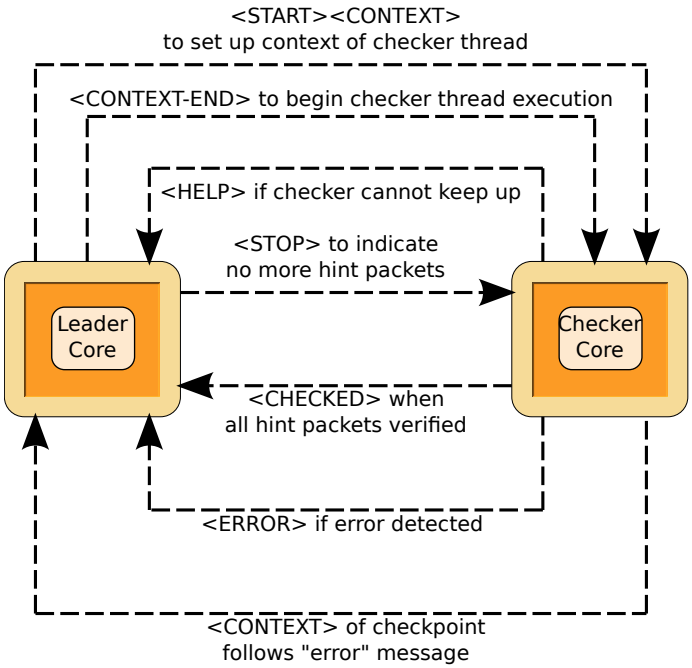


Fig. 7: Leader-checker communication

We now describe the core that executes the checker thread. The checker thread's run at a core comes to an end because of two reasons. The first reason is that the leader completed its execution, and the second is that the leader or arbiter decided to reassign the role of checking to another core. In either case, a checker thread is sent a *STOP* message. After this point, the checker thread finishes checking all the instructions in the hint buffer, and then sends a *CHECKED* message to the leader. The leader thread starts the new checker thread (if any) by sending it the *CONTEXT-END* message.

4.2.2. Maintenance of Register Checkpoints. The checker needs to maintain a consistent checkpoint for the architectural registers, and the PC. We can either create a radiation hardened register file [Montesinos et al. 2007] that is associated with minimal performance overheads, or we can have a separate radiation hardened retirement register file that is off the critical path. We prefer the latter option. The retirement RF is only updated after successful verification against a hint packet. Similarly, a dedicated PC register can keep track of the last correctly executed PC.

4.3. Checking Memory Instructions

This section describes the memory checkpoint hardware. We do not have any major contributions in this part of the architecture other than the forwarding filters. We use standard L1 cache based checkpointing schemes as proposed in [Prvulovic et al. 2002]. We assume that the leader's L1 cache contains speculative (unverified) data, and the checker is allowed to update memory state.

There are some standard performance optimizations proposed in prior work. The leader can forward cache lines to the checker core to increase the IPC of the checker thread. We forward a line from the leader to the checker upon an L1 cache miss. When forwarding from the leader core to the checker, we use a forwarding filter to reduce the number of forwardings.

4.3.1. Forwarding Filters. A core maintains two buffers – *recently forwarded buffer* (RFB) and *lines to be forwarded buffer* (LFB). The entries of both the structures are tagged with the leader thread id. The RFB contains the block addresses of lines recently forwarded. It follows a least-recently-used (LRU) replacement policy. Whenever there is an L1 cache miss at a leader thread, the corresponding line address is searched in the RFB. If found, its timestamp is updated. If not, it is added to the RFB and the corresponding cache line is placed in the LFB with its ready flag set to *false*. When the particular memory operation that caused the L1 miss reaches the commit stage, and its hint packet is constructed, the ready flag of the corresponding LFB entry is set to *true*. A small circuit dequeues ready entries from the LFB and forwards them to appropriate checker core through the NoC.

The role of the RFB is to remember which lines were recently forwarded and prevent their re-forwarding, as these lines are, with a high probability, still at the checker. This minimizes the amount of NoC bandwidth consumed. When a leader thread is assigned a new checker, the RFB needs to be cleared.

The role of the ready flag is to ensure an as-late-as-possible forwarding to reduce the frequency of the fairly common occurrence of a forwarded cache line being evicted from the checker's L1 cache before the corresponding instruction arrives for execution.

The forwarding filter is required to improve performance, and is not critical to correctness. If there is an RFB hit, and the line is not found in the checker's private L1, the miss is serviced by searching the shared L2.

4.3.2. Memory Checkpointing. Whenever the leader writes to its L1 cache, it marks the entry as speculative (unverified). In addition to the speculative bit, to support multiple leaders executing on a core, we tag every cache line with a 2 bit id (since we can have at the most 4 leaders executing on a cache). We assume bag-of-task applications and thus zero overlap between the write sets of two leaders.

Note that we do not allow a cache line to be evicted till it is marked non-speculative. A standard optimization proposed in prior work is to use a victim cache to save speculative lines that have been evicted from the cache. If the victim cache fills up, we force a leader-checker synchronization (*flush* operation – explained in the next paragraph). The checker operates as a normal thread, and does not need to use the victim cache.

We need to periodically mark speculative lines as non-speculative. By default, we perform a *flush* operation for every migration of the checker thread (a minimum of once every epoch). The leader thread flushes its pipeline, asks the checker to stop (by sending the *STOP* message), and requests the arbiter for a new checker. It waits for the *CHECKED* message from the previous checker. After receiving the message it marks all its speculative lines in its L1 cache and victim cache (identified by the 2 bit id) as non-speculative. We can use a gang clear mechanism.

Even if a leader-checker pair are executing on the same core, we do not have an issue because the checker never accesses speculative data. If there are two copies of a line in a set (one speculative, and the other non-speculative), the leader always accesses the speculative copy. The cache controller ensures that the thread gets access to the right version of data.

4.4. Rollback Mechanism

When a soft error is detected by the checker, the contents of its radiation hardened retirement register file form the checkpoint to rollback to. The checker flushes its pipeline and sends the leader an *ERROR* message. This is followed by the register checkpoint (*CONTEXT* message). The leader on receiving the *ERROR* message flushes its pipeline, sets up its context based on the received checkpoint, and begins execution from that point. We can avoid errors in the NoC while transferring checkpoints by

adding extra CRC bits to the message. These messages are not on the critical path since errors are very rare.

Regarding memory checkpointing, the stores committed by the leader core only propagate as far as the private L1 cache. When the checker core verifies the store, the leader is allowed to write the store to lower levels of the memory. Therefore, in the event of a soft error, the leader core can recover by simply purging all unverified lines.

4.5. Resolving Livelock Issues

We can unfortunately have some livelock situations in our architecture.

Let us consider a case where we have a leader of thread A, and a checker of thread B executing on the same core. Assume that the checker thread faces a decode stall because it can not get a free entry in the ROB or LSQ. In this case, it needs to wait for some ROB or LSQ entries to get freed. The leader of thread A might be waiting for some data from memory and it might take 100s of cycles for the checker to start again. Meanwhile, it is possible that the hint buffers of the checker fill up, and its leader (thread B) blocks. To avoid a potential deadlock situation, we force the leader (thread B) to flush all its instructions from the pipeline, and request the arbiter to reassign it a new checker.

Now, it is possible that in the new core, the checker of thread B faces decode stalls again, requiring yet another migration. In this manner it is possible to have a lot of migrations in a small window of time. We have observed this scenario in our simulations very frequently.

To avoid such livelock issues, we guarantee some progress to each checker thread. A leader instruction is allowed an entry in one of the pipeline structures such as the re-order buffer, load-store queue or instruction window, only if the occupancy of the structure is less than a certain fixed fraction (say 0.95, as used in all simulations). This guarantees a small window for any checker threads on the core to progress. In practice, this eliminates most of our livelocks, and we do not have repeated migrations.

4.6. Effect of *FluidCheck* on the Operating System

The hardware reveals to the OS a set of virtual cores. The OS, which only deals with leader threads, schedules them on the virtual cores. A hardware scheduler, in consultation with the arbiter logic, then schedules the leader threads and checker threads on the different physical cores.

All system calls and asynchronous interrupts are executed by both the leader and checker threads. When the leader decodes an I/O instruction, it creates a data packet that contains the details of the I/O operation. The leader sends the I/O data packet to the checker. Simultaneously, the leader sends a hash of the I/O data packet to a dedicated module in the I/O system such as the Southbridge chip on Intel motherboards. The checker checks the consistency of the I/O data packet (note that the checker has verified all instructions up to the I/O operation at this point) and then forwards it to the I/O system. The dedicated I/O module computes a hash of the I/O packet, compares it with the hash that it has received from the leader, and if both of them match, schedules the I/O operation. Obtaining the results of a system call is not very complicated. Most operating systems write the results of system calls to dedicated regions in the process' memory. After returning from a system call the leader reads the results using regular memory reads. The checker does the same, when it is checking the instruction.

When an asynchronous interrupt arrives, it is sent to both the leader and the checker. The leader flushes all of its buffers, sends an INTERRUPT packet to the checker, and waits for the checker to verify all the outstanding instructions. Once the checker successfully verifies all instructions up to the INTERRUPT packet, it sends a CHECKED message back to the leader, and configures itself to expect hints corre-

sponding to the interrupt handler. The leader, on receiving the CHECKED message, jumps to the interrupt handler. Once the interrupt handler has been executed, the transition back to the application context is done by both the leader and the checker threads without any explicit communication.

5. ARBITER LOGIC

The arbiter is invoked whenever we wish to map threads (leader/checker) to a subset of the available cores. It can either be invoked when it is necessary to migrate a blocked checker thread or periodically at epoch boundaries. In any case, the mechanism for mapping threads is the same. We first describe the method to map a single thread to an SMT core that is available. An n-way SMT core is said to be *available* when less than n threads are currently running on it.

5.1. Mapping a Single Thread

The first step is to define a metric that captures the activity of a thread or a core. We define *activity* in two ways. We first define activity as the average IPC of a thread from the start of an epoch. In the case of a core, we define this metric as the sum of the average IPC (in the last epoch) of all the threads running on it (measured from the start of the current epoch). However, using IPC as a measure of activity is not the best solution. We thus defined a metric called weighted IPC (WIPC) that is defined as follows. It is equal to the IPC, when the thread is a leader, and it is equal to $x \times IPC$, when the thread is a checker. In this case, x is a parameter called checker importance. We shall vary it in our experiments, and find the best configuration. For a core, we define $WIPC(x)$ as the sum of the weighted IPCs of all the threads (leader or checker) running on the core.

To map a single thread to an available SMT core, we rank all the cores in ascending order of activity (using the IPC or WIPC metric). Subsequently, we choose the first core that has free SMT slots available and map the thread to that core.

When IPC is used as the metric to measure activity, the arbiter type is termed *minIPC*. When WIPC is used, the arbiter type is termed *minWIPC_x*, where x is the checker importance. The two metrics require a per-thread instruction counter at each core. The values of the counters are communicated to the arbiter at the end of each epoch.

5.2. Mapping a Set of Threads

We need to periodically remap all the threads at the end of every epoch. We consider three scheduling policies. The first scheduling policy is called *Pinned Leaders* (SP-PL). Here, for the leader threads, a statically determined random mapping of threads to cores is followed. Regarding the checker threads, in every epoch, we sort them in descending order of activity and map each thread according to the scheme mentioned in Section 5.1. Note that the mapping of leaders to cores is fixed, and it cannot change dynamically over time. The intuition was to have an equitable distribution of leader computation across cores, in terms of number of threads. The checker threads are then distributed every epoch in an attempt to achieve an equitable distribution in terms of core activity.

The second scheduling policy is called *Unpinned Leaders* (SP-UL). There is no static mapping for any of the threads. Every epoch, we sort *all* the threads in descending order of activity, and then map the threads to cores (according to the scheme described in Section 5.1). Here, the intuition is to distribute threads in such a manner that all cores display a similar activity. It is important to note that here we do not differentiate between leader and checker threads. The focus is on achieving a distribution wherein

a similar activity (be it in terms of leader or checker instructions) is observed at each core.

The last scheduling policy, *Unpinned - All Leaders First* (SP-UALF), is the most powerful. It allows full flexibility like in SP-UL, but gives additional importance to leaders. This is desirable as leaders are more demanding of resources (argued further in Section 6.1.2). Every epoch, we first sort all the leader threads in descending order of activity, and then map them to cores (as described in Section 5.1). We then do the same with all the checker threads. This way the leaders are given the first shot at appropriating resources for themselves, and the checkers are then distributed across the cores in an equitable fashion (in terms of core activity).

5.3. Thread Migration and NoC Issues

We evaluated many schemes that additionally take thread migration and NoC locality into account. However, they were not found to yield statistically significant improvements given the additional complexity in implementation. The reason is as follows. We assume a 2D torus topology. This gives an average leader-checker distance of 2 hops, for a 16 core system. Thus, leaders and checkers are typically quite close to each other. The different hint streams do not affect each other significantly as the time spent by them in the network is quite short. Secondly, the forwarding filters reduces the number of flits sent. Thirdly, all our three heuristics (SP-PL, SP-UL, SP-UAFL) rely on the activity of phases. The activity does not change significantly across most phase boundaries, and thus thread migrations are relatively rare (it was observed that on an average, a thread migrates once every 383 million cycles). Consequently, the network traffic is not jittery and the different streams have minimal effect on each other. We shall discuss the actual NoC traffic numbers, and stalls due to the NoC in Section 6.4.4.

5.4. Fetch Policies

SMT processors are typically very flexible in the issue, commit, and dispatch stages, and apportion the slots dynamically between the threads. However, their fetch units are typically very traditional and fetch from the different threads in a round robin order [Tullsen et al. 1995]. We extend the scheme *Full Simultaneous Issue* (proposed in [Tullsen et al. 1995]) in FluidCheck. Our aim is to give more fetch slots to high performing threads. The proposed scheme is as follows. We consider a block of 16 fetch cycles in an SMT core. Next, we apportion the 16 fetch cycles among the threads proportional to their activity. If the activities of n threads are $A_1 \dots A_n$, then the number of fetch slots that the i^{th} thread gets is $A_i / (\sum_{k=1}^{n} A_k) \times 16$. We alternately round the fetch slots using the floor and ceiling functions such that the total fetch slots across the n threads add up to 16. This scheme is similar to the proposals in [Tullsen et al. 1996].

6. EVALUATION

The proposed checker architecture was evaluated using the cycle accurate architectural simulator Tejas [Sarangi et al. 2015; Malhotra et al. 2014]. The NoC has been modeled in detail, and has been included in the energy estimations. For modeling power, we use McPat [Li et al. 2009], and Orion 2 [Kahng et al. 2009]. For estimating the delay and power of FluidCheck structures, we use the models in Cacti 5.3 [Thoziyoor et al. 2008]. We scale the results to 22nm by using the methods suggested in [Huang et al. 2011]. All our simulation tools have been rigorously validated with native hardware (see the original publications).

Our simulated system consists of 16 cores. Each core is a 4-way SMT (the implementation of SMT is as explained in Section 5.4). The simulation parameters were derived from the designs of Intel Sandybridge, and IBM Power7 (see Table I). Each core has

Parameter	Value	Parameter	Value
Cores	16	Technology	22 nm
Frequency	3.6 GHz		
Pipeline			
Retire Width	4	Integer RF (phy)	160
Issue Width	6	Float RF (phy)	160
ROB size	168	Predictor	Tournament (PAG-PAP)
IW size	54	Bmispred penalty	14 cycles
LSQ size	64	Multi-threading	4-way
iTLB	128 entry	dTLB	128 entry
Integer ALU	4 units	Int ALU latency	1 cycles
Integer Mul	1 unit	Int Mul latency	2 cycles
Integer Div	1 unit	Int Div latency	4 cycles
Float ALU	2 units	FP ALU latency	2 cycles
Float Mul	1 unit	FP Mul latency	4 cycles
Float Div	1 unit	FP Div latency	8 cycles
L1 i-cache, d-cache			
Write-mode	Write-back	Block size	64
Associativity	8	Size	32 KB
Latency	3 cycles		
Shared L2			
Write-mode	Write-back	Block size	64
Associativity	8	Size	12 MB
Latency	45 cycles		
Main Memory Latency			
Technology	DDR3	Latency	200 cycles
NoC and Traffic			
Topology	2-D Torus	Routing Alg.	dynamic X-Y routing
Flit size	32 bytes	Hop-latency	2 cycles
Hint size	16 bytes	Cache line size	64 bytes
Auxiliary structures size (number of entries)			
Hint Buffer	512	Victim Cache	32
RFB	64	LFB	64

Table I: Simulation parameters

a fetch width of 4, issue width of 6, has a 32 KB i-cache and d-cache. We additionally, have a 12 MB L2 cache, and a 2D torus based NoC.

We evaluate three classes of workloads: *low*, *medium* and *high*. *Low* refers to a configuration where we simulate 16 applications (16 leaders + 16 checkers). Similarly, *medium* refers to 24 applications (total: 48 threads), and *high* refers to running 32 applications (total: 64 threads). For each run we randomly choose a set of benchmarks from the SPEC CPU2006 suite, and additionally run each configuration for 100 times. We report the geometric mean values.

For each application we use "Pinpoints" [Patil et al. 2004] to find the representative portions of the SPEC CPU2006 benchmarks. On an average, a benchmark's trace consisted of 335 million instructions, duly weighted as prescribed by Pinpoints. Note that Pinpoints finds those regions of a benchmark that are representative of the entire program. Instead of simulating hundreds of billions of instructions (which is not feasible), we just need to run the code sections identified by PinPoints. The authors of Pinpoints [Patil et al. 2004] have validated the toolset against real hardware (Itanium 2 processor). They have shown the error to be less than 8% for integer benchmarks and less than 3% for floating point benchmarks from the SPEC CPU2006 suite. Using Pinpoints or its sister software SimPoints is a standard approach.

Additionally, we explore the effect of spare cores. We simulate workloads having 15 and 14 benchmarks, resulting in 1 and 2 spare cores respectively (Figure 17). Presence of spare cores results in a significant drop in the penalty on performance.

The comparison of the various schemes is done on the basis of the slowdown in the simulated execution time as compared to the case of unreliable execution (no checker threads). Mathematically, the slowdown experienced by a workload W is given by:

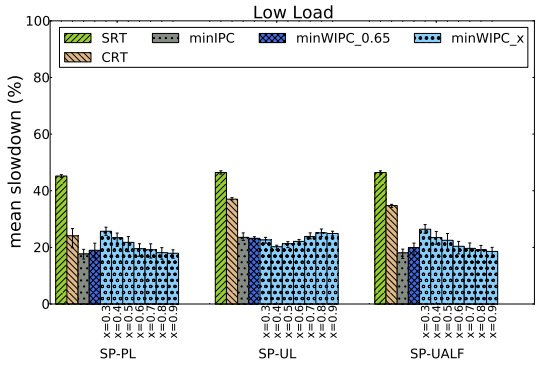


Fig. 8: Mean slowdown - *low load*

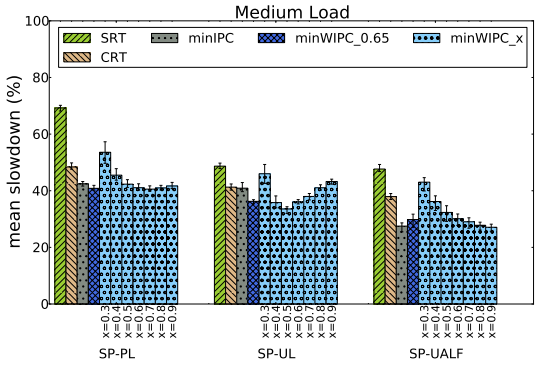


Fig. 9: Mean slowdown - *medium load*

$$slowdown = \sqrt[|W|]{\prod_{b \in W} \frac{cycles\ taken\ to\ reliably\ execute\ b}{cycles\ taken\ to\ unreliably\ execute\ b}} - 1$$

We set the epoch size to 100k cycles. We compare our proposals against seminal works in the area – AR-SMT [Rotenberg 1999]/SRT [Reinhardt and Mukherjee 2000] and CRT [Mukherjee et al. 2002]. These works are implemented diligently in our many-core framework. In AR-SMT/SRT, a leader-checker thread pair is scheduled on the same core. In CRT, the checker thread is scheduled on a *partner* core (adjoining core on the NoC). For both SRT and CRT, hints are forwarded to the hint buffer at the checker core (via the network-on-chip for CRT). Cache line forwarding, as explained in Section 4.3.1, is enabled as well for both the schemes.

6.1. Performance Evaluation

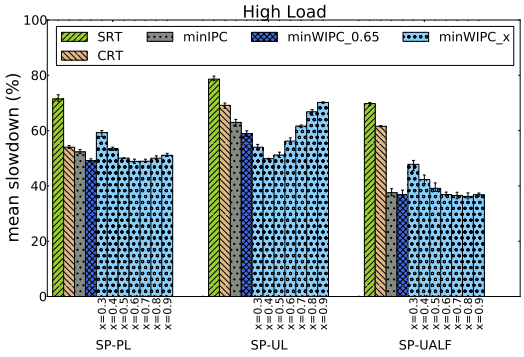


Fig. 10: Mean slowdown - *high load*

Figures 8,9 and 10 compare the mean slowdown of FluidCheck against AR-SMT/SRT and CRT, under *low* load, *medium* load and *high* load conditions respectively. We evaluated our three scheduling policies: *Pinned Leaders* (SP-PL), *Unpinned Leaders* (SP-UL) and *Unpinned – All Leaders First* (SP-UALF). Since each experiment is run for a

100 times, we show error bars that indicate the minimum and maximum values of the measured metric. The total deviation from the mean is in the range of 1-7%.

The general trend is that SRT suffers from significant performance penalties, as high as 80% for high workloads. CRT achieves a lesser slowdown as compared to SRT, and FluidCheck vastly outperforms both SRT and CRT – experiments revealed up to 65.3% improvement over SRT and up to 52.8% over CRT. We will now discuss the improvement obtained under different load conditions and with different scheduling policies.

6.1.1. Load Analysis. Under *low* load conditions, the checker threads find greater room to move in an unintrusive fashion. This is highly favorable to FluidCheck, providing it with ample flexibility to move threads around to make best use of the resources. These factors result in a minimal slowdown. The proposed techniques provide reliable execution with a mere 18% penalty on performance on average (under SP-UALF). In comparison, SRT and CRT are far more expensive – SRT shows a mean slowdown of 46.4% and CRT shows a mean slowdown of 34.76%.

As the load increases, we see the penalties becoming more pronounced. The introduction of checker threads now results in a greater competition for resources. Under *medium* loads, FluidCheck shows a slowdown of 27.51% on average (under SP-UALF) – 42.29% faster than SRT, and 27.47% faster than CRT. Under *high* loads, FluidCheck shows a slowdown of 36.2% (under SP-UALF) – 47.23% faster than SRT, and 40.2% faster than CRT.

6.1.2. Scheduling Policy Analysis. We observe that the SP-PL policy displays the largest performance penalty. This is expected as the policy is quite simple in the scheduling of leader threads to cores. Since it simply follows a statically determined mapping, it is relatively powerless as compared to the other two policies. FluidCheck, with the SP-PL policy displayed slowdowns of 19.53%, 40.6% and 50% for *low*, *medium* and *high* loads respectively.

Among the other two, we note that the slowdowns are significantly lower when adopting the SP-UALF policy as compared to the SP-UL policy. In other words, scheduling all leader threads before the checker ones proves beneficial. The reason for this is that checker threads are high performing – for the same sequence of instructions, a checker thread places a lower demand on resources. Thus, giving more importance to the more demanding leaders by scheduling them first displays significant gains.

FluidCheck, with the SP-UL policy displayed slowdowns of 20.22%, 33.55% and 49.8% for *low*, *medium* and *high* loads respectively. The corresponding slowdowns with the SP-UALF policy were significantly lower – 18%, 27.51% and 36.2% respectively.

All further discussions in this section are based on experiments employing the SP-UALF policy.

6.1.3. Varying Checker Importance in minWIPC_x. As discussed earlier, giving leader threads more importance is beneficial. We study the variation in the slowdown observed when the checker importance is varied. Figures 8, 9 and 10 show the effect of varying the importance (x) in the minWIPC schemes. Gains of up to 22% were obtained through prudent selection of the checker importance. A checker importance value of 0.65 was found to perform consistently well across different load conditions and scheduling policies. We shall henceforth use 0.65 as the default checker importance for the $minWIPC$ configuration.

6.1.4. Comparison with Standard Thread Scheduling Algorithms. We compare FluidCheck's scheduling schemes with several seminal proposals – *Data Cache Conflict Scheduling* (DCCS) [Settle et al. 2004], *IPC-based Scheduling* (IPCS) [Parekh et al. 2000], *Ready*

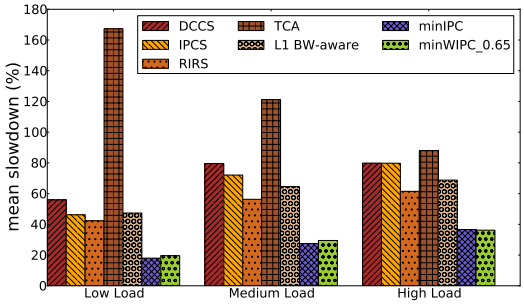
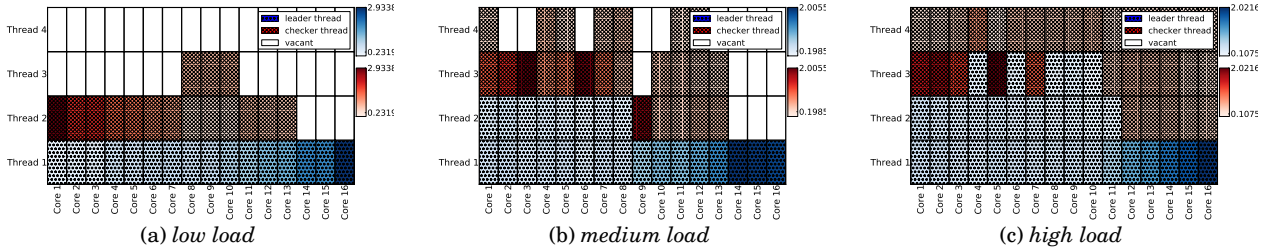


Fig. 11: Per-benchmark slowdown



(a) *low load* (b) *medium load* (c) *high load*

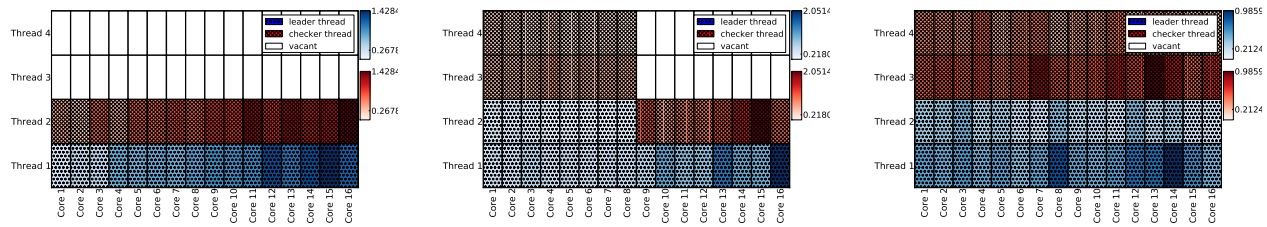
Fig. 12: Thread mapping snapshot (*minIPC*, SP-UALF)

Inflight Ratio Scheduling (RIRS) [El-Moursy et al. 2006], *Thread to Core Assignment* (TCA) [Acosta et al. 2009] and *L1 bandwidth aware allocation* [Feliu et al. 2013]. The results are shown in Figure 11. It must be noted that the TCA algorithm is geared towards a system where the number of threads to be run is equal to the number of hardware threads, that is, the *high* load scenario. Consequently, it shows poor performance in the *low* and *medium* load scenarios.

The slowdown achieved by the best of these schemes, RIRS, is 104% greater than that achieved by FluidCheck (in the *medium* load scenario). FluidCheck vastly outperforms these schemes (57.45% (*low* load) and 40.5% (*high* load) faster than RIRS) as they suffer from the critical drawback of not differentiating between leader and checker threads.

6.2. Deeper Insights

6.2.1. Balanced Load Distribution. An equitable distribution of the load across cores maximizes throughput. The heuristics *minIPC* and *minWIPC* aim to achieve this equitable distribution. Figure 12(a),(b) and (c) show snapshots taken during different FluidCheck (*minIPC*) runs. Each column in the grid corresponds to a core, and each cell in a column represents an SMT thread. Now, the color of the cell indicates the kind of thread: blue (with the circles pattern) for leaders, red (with the crosses pattern) for checkers. The intensity of the color is proportional to the IPC of the corresponding thread. The figures show that if the intensity is summed up over the columns, then the 16 sums are roughly equal. Figure 13(a),(b) and (c) show snapshots taken during different CRT runs. The snapshots were taken when the constituent benchmarks were in the same phases as when the *minIPC* snapshots were taken. The standard devi-



(a) *low load* (b) *medium load* (c) *high load*

Fig. 13: Thread mapping snapshot (*CRT*, SP-UALF)

		unchecked	SRT	CRT	minIPC	unchecked	SRT	CRT	minIPC	unchecked	SRT	CRT	minIPC
		Workload : Low				Workload : Medium				Workload : High			
Branch Mispred./instruction		0.009	0.009	0.009	0.009	0.010	0.010	0.010	0.010	0.010	0.011	0.009	0.011
LSQ full		≈0	0.011	0.023	0.013	≈0	0.016	0.023	0.019	≈0	0.013	0.019	0.018
IW full		0.001	0.001	0.002	0.002	0.001	0.001	0.002	0.002	0.001	0.001	0.001	0.001
ROB full		≈0	0.009	0.010	0.005	≈0	0.007	0.009	0.008	≈0	0.004	0.007	0.008
NoC busy		≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0	≈0
leader data hazard*		0.882	0.528	0.459	0.241	0.724	0.419	0.340	0.245	0.682	0.345	0.297	0.246
leader structural hazard**		0.024	0.019	0.008	0.010	0.032	0.025	0.010	0.013	0.026	0.020	0.008	0.010
iCache hit rate	**leaders**	0.996	0.996	0.995	0.996	0.994	0.991	0.988	0.989	0.987	0.984	0.979	0.984
	checkers		0.999	0.999	0.999		0.999	0.999	0.999		0.999	0.999	0.999
dCache hit rate	**leaders**	0.838	0.820	0.830	0.828	0.832	0.793	0.816	0.812	0.822	0.786	0.797	0.796
	checkers		0.980	0.990	0.992		0.965	0.979	0.980		0.962	0.971	0.975

The units of all the stall/hazard statistics are: average stalls/hazards per leader instruction (varies from 0 to 1).
* leader data hazard: cycles when no instruction was issued, and the functional unit has a leader instruction with operands not ready
** leader structural hazard: cycles when no instruction was issued, and the functional unit has a leader instruction with operands ready but has no FU to execute on

Table II: Stall statistics and cache hit rates

ation of the cumulative IPCs on each core is 0.55 in this case for the *medium* load scenario. This is much greater than a similar figure for FluidCheck, which is 0.31. The low standard deviation in FluidCheck is indicative of the fact that the execution load is equally distributed among the cores, which is essentially the crux of the idea behind FluidCheck. Since FluidCheck is better in load balancing, we see a reduced pressure for resources in cores, and this leads to a higher aggregate performance.

	SP-PL	SP-UL	SP-UALF
***minIPC* - Workload : Medium**			
BMispred/inst.	0.010	0.010	0.010
LSQ full	0.028	0.010	0.019
IW full	0.002	0.002	0.002
ROB full	0.002	0.004	0.008
NoC busy	≈0	≈0	≈0
leader data hazard	0.372	0.330	0.245
leader structural hazard	0.034	0.014	0.013
* see Table II for definitions of the metrics			

Table III: Stall statistics – across different scheduling policies

6.2.2. Explanation of the Trends. In this section, we will try to explain the reason for the better performance of *FluidCheck*. Table II shows the frequency of occurrence of different events in the pipeline that can lead to idle cycles, and the hit rates in the L1 caches. For the pipeline, we look at the branch misprediction rate per program instruction, and the number of cycles (divided by the number of instructions) that different structures in the pipeline (LSQ, instruction window (IW), ROB, and NoC buffers) are full and cannot accept new instructions. Additionally, we also look at data and structural hazards for leader threads. Data (structural) hazards are quantified by the number of cycles (divided by the number of instructions) that leader instructions are not issued because of the unavailability of operands (FUs).

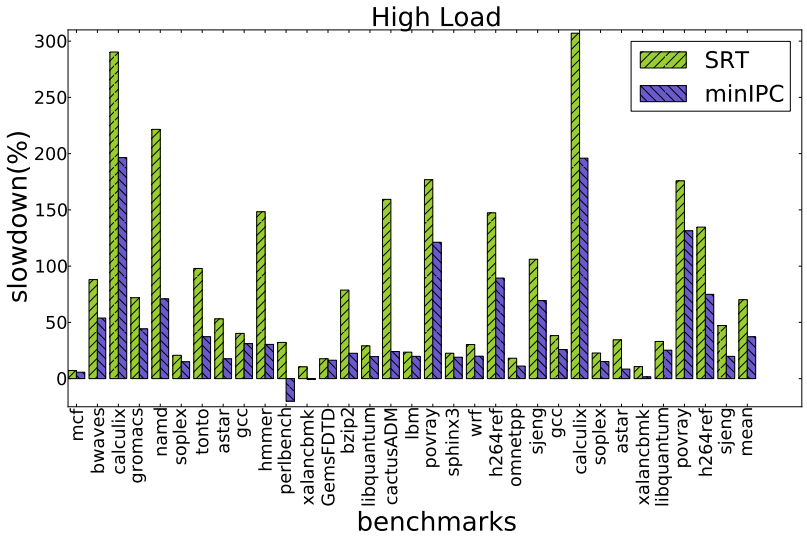


Fig. 14: Per-benchmark slowdown

Other than the data and structural hazards suffered by the leader threads, all the other numbers are roughly comparable across the four configurations: unchecked, SRT, CRT, and *FluidCheck*(minIPC). We believe that the differences in speedup is primarily because of the lower rate of hazards and resultant higher IPCs of leader threads in *FluidCheck*. For example, for the *Medium* workload, the average leader data hazard rate was found to be 0.245 in the case of *minIPC*. This is quite low compared to those of CRT and SRT, which displayed data hazard rates of 0.34 and 0.419 respectively. In other words, the probability that a leader thread of SRT could not find an instruction to execute is almost twice that of FluidCheck. By means of a more balanced load across cores, FluidCheck ensures a higher flow of instructions through the instruction windows of all the cores. Consequently, the select logic in each has a larger pool of instructions to work with, reducing the probability of no instructions being issued.

Additionally, SRT displays a high structural hazard (not finding enough functional units) rate, as it places both the leader and checker threads of an application on the same core. Since their resource requirements are similar, they place a greater demand on the same resources. For example, for the *Medium* workload, *minIPC* reports a structural hazard rate of 0.013, while SRT reports a rate of 0.025.

To summarize, the main reason for the high performance displayed by FluidCheck is that our heuristics try to equitably balance the load across cores. As a result the SMT cores can fairly allocate issue slots and ensure good execution throughputs for the instructions of all the threads. A different view to the observations can be obtained in Figure 14. This illustrates the slowdown suffered by each benchmark with a *high* load, under the SRT and FluidCheck schemes. The slowdowns are markedly lower in FluidCheck as the threads have been so scheduled so as to balance the use of the resources, and consequently maximize the throughput of the system.

The different scheduling policies are compared in Table III. Here as well, we see that the main difference in the schemes is the number of leader data hazards. For the *Medium* workload, SP-UALF has the lowest rate of data hazards, and also has the best performance (see Figure 9).

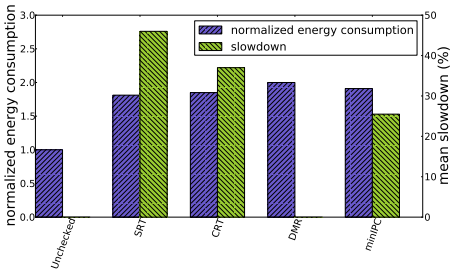


Fig. 15: Energy consumption

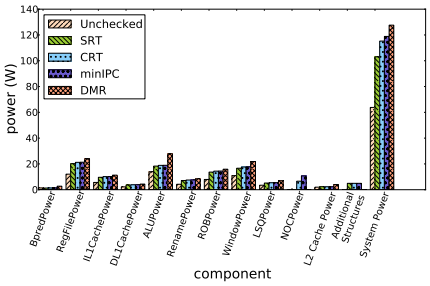


Fig. 16: Power consumption

6.2.3. Thread Migrations. We observe that in our schemes, on an average a thread migrates once every 383 million cycles. This can be attributed to the fact that workloads typically have long phases of unchanging behavior. Consequently, arbitration at successive epochs typically results in the same mapping being repeated. This has two implications: (1) the NoC traffic is not jittery, allowing the system to settle on stable and balanced routing paths, and (2) expensive TLB flushes (which are required when a thread migrates) are minimized.

6.2.4. Common Mode Failures. Suppose *FluidCheck* reports a false negative (masks an error) due to a common mode failure. Note that false-positives are not an issue (only performance penalty). A false-negative is possible if there is a failure in the error reporting circuitry. Since this circuitry is off the critical path, we assume that it is made of radiation hardened transistors, which are slow yet very reliable. This is a standard assumption in this field. It is also possible that both the leader and checker instructions get corrupted in exactly the same way. If the leader and checker threads are running on different cores, the probability of this is infinitesimally small. If the two threads are mapped to the same core, a lag of around 50-150 cycles is ensured between them. It is very unlikely that a single (or very rarely multiple) ephemeral particle strike will cause two instructions a hundred cycles apart to get corrupted in exactly the same way. However, if this is a concern, then we can run the leader and checker threads of each application on different cores (performance penalty of $<$0.1% as compared to the numbers shown in Section 6.1).

6.3. Energy Consumption

In this section, we evaluate the power and energy overheads of our scheme with respect to the baseline, *Unchecked*, which does not have any redundancy. The checker threads contribute to an increase in energy. However, the overhead is not 100%. This is because a checker thread is more energy efficient than the leader – it does not use the branch predictor, structures for load latency speculation, and the wakeup-select logic. However, additional energy is consumed in the functional units, NoC for forwarding hints and the L1 caches. The additional structures required to facilitate assisted re-execution and error detection also pose an overhead.

Figure 15 shows the normalized energy consumption and slowdown as compared to the baseline (*medium* workload). We also compare with an idealized DMR scheme where we assume that no energy is consumed in communicating and comparing results. We observe that the energy consumption of DMR is quite high (100%) as compared to the assisted schemes CRT, SRT, and FluidCheck. The assisted schemes were found to be comparable – with an average energy overhead of roughly 81-91%, with FluidCheck at 91%.

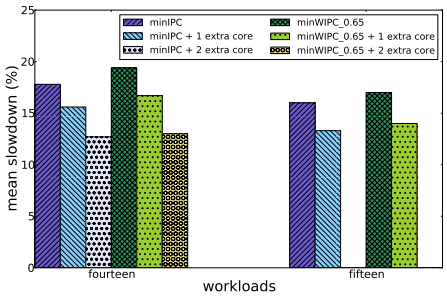


Fig. 17: Effect of spare cores

Figure 16 shows a breakup of the power consumption across different components. DMR is clearly the most inefficient primarily because it does not use hints and uses full re-execution. In comparison FluidCheck is 10.74% more power efficient. The power consumptions of SRT, CRT, and FluidCheck (*minIPC*) are comparable: between 103 to 118 Watts. The main contributors to the power overhead are the register files, ALUs, and the instruction window. The NoC power is roughly 10.7W (out of 118W) and is thus not a dominant component of the power consumption. This is primarily because the size of our network is relatively small: 4×4 torus, and since the traffic patterns do not change for a long time (see Section 6.2.3), our routing paths stabilize in a way that contention is minimized. Additionally, the forwarding filters reduce the amount of traffic in the NoC. The additional structures in FluidCheck to provide reliability (hint buffers, forwarding filters, radiation hardened register files, victim caches and the arbiter) consume a modest 5.27W.

We observed similar trends for the other *low* and *high* workloads. DMR proves to be the most expensive. The three assisted checking methods perform comparably, with FluidCheck being marginally higher due to the increased NoC usage. The average energy overheads of FluidCheck with *low* and *high* workloads were found to be 84.7% and 93% respectively.

6.4. Sensitivity Studies

6.4.1. Effect of Spare Cores. In this section, we study the effect of having some cores free such that they can be used to schedule highly demanding leaders or checkers. FluidCheck can seamlessly integrate such free cores. Figure 17 shows two configurations when we have fourteen and fifteen applications respectively. Like all our previous experiments, we compare the performance to a baseline design that does not have redundant threads.

The extra cores significantly improve performance, allowing us to achieve total coverage with a slowdown as low as 12%. When workloads consisting of 14 SPEC CPU2006 benchmarks each were employed, two spare cores are available. Using them both, the *minIPC* heuristic saw a 28.6% improvement, while the *minWIPC* heuristic saw a 33% improvement. When workloads consisting of 15 SPEC CPU2006 benchmarks each were employed, one spare core is available. The *minIPC* heuristic saw a 16.9% improvement, while the *minWIPC* heuristic saw a 17.6% improvement.

6.4.2. Performance of Cache Forwarding. Table II shows the hit rates observed at the L1 level in the different schemes. We see that there is a small drop in the leader d-cache hit rates under FluidCheck as compared to unchecked execution (1.2%, 2.4% and 3.2% under *low*, *medium* and *high* loads respectively). This is due to the additional checker threads evicting leader lines from the cache. We see that the cache forwarding

		minIPC	minWIPC_0.65	minIPC	minWIPC_0.65	minIPC	minWIPC_0.65
		Workload : Low		Workload : Medium		Workload : High	
iLFB	Mean	0.209	0.223	0.482	0.487	0.583	0.534
	Max	64	64	64	64	64	64
dLFB	Mean	20.806	22.298	25.954	26.516	27.882	24.690
	Max	64	64	64	64	64	64
Victim Cache	Mean	0.292	0.002	0.170	0.111	≈0	≈0
	Max	32	32	32	32	13	20

Table IV: Occupancy of Auxiliary Structures

	minIPC	minWIPC_0.65	minIPC	minWIPC_0.65	minIPC	minWIPC_0.65
	Workload : Low		Workload : Medium		Workload : High	
Mean Bandwidth usage (bytes/core/cycle)	17.304	17.134	18.305	18.105	18.874	18.533
Mean leader-checker distance (hops)	2.503	2.44	2.219	2.108	2.224	2.05
Mean d-cache forwardings (per memory inst)	0.164	0.162	0.124	0.122	0.094	0.094
Mean i-cache forwardings (per instruction)	0.018	0.018	0.018	0.018	0.012	0.012

Table V: Communication Statistics

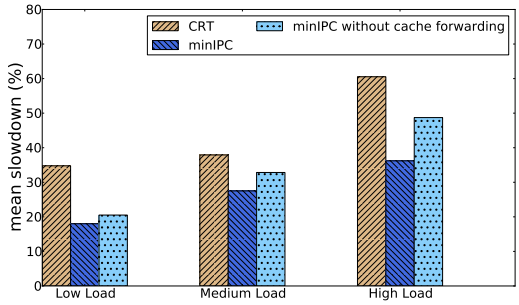


Fig. 18: Benefit due to cache forwarding

mechanism enables the checker threads to enjoy a high hit-rate ($> 97\%$). Note that cache forwarding was enabled for SRT and CRT as well, to make the comparison fair.

To further emphasize the importance of cache forwarding, we studied the effect of disabling it. We consider three configurations in Figure 18: CRT (with forwarding), *minIPC* (with forwarding), and *minIPC* (without forwarding). First, we can conclude from the figure that even without forwarding *minIPC* has lower slowdowns that CRT with forwarding (for all three types of loads). It has roughly 13.4-41% lower slowdown. We get some amount of additional improvement when we turn on forwarding for *minIPC*. In this case, the slowdowns reduce by 12.3%, 16.3%, and 25.7% for the *low*, *medium*, and *high* configurations respectively.

For every leader memory instruction, an average of 0.12 lines are forwarded from the leader's d-cache (*medium* load), and an average of 0.02 lines per leader instruction are forwarded from the i-cache (see Table V). We do not need to forward frequently because the RFB helps in detecting the temporal locality in accesses. The reduction in forwarding helps conserve NoC bandwidth, reduce congestion and performance degradation (see Tables II and V). The average occupancies of the LFBs is quite low (see Table IV), indicative of a low pressure on the NoC. The mechanism is also robust to the problem of forwarding being too early, which was discussed in Section 4.3.1.

6.4.3. Performance of the Victim Cache. The speculative L1 cache requires a victim cache. If the victim cache fills up, then we need to force a *flush* operation (see Section 4.3.2).

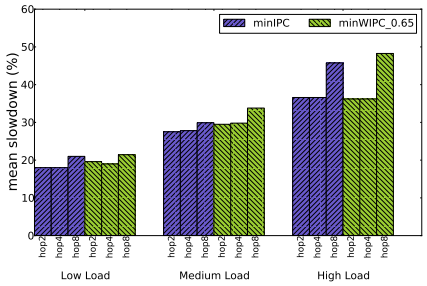


Fig. 19: Varying hop latency

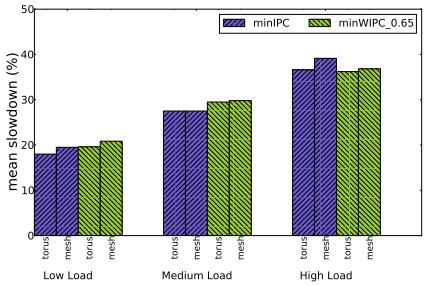


Fig. 20: Varying NoC topology

This is detrimental to performance. However, this scenario is not very frequent as indicated in Table IV. Although the occupancy of the victim cache reaches its maximum limit of 32 occasionally, the frequency is rare. The average occupancy was < 1 in all our experiments. For larger workloads, we see the frequency dropping even further – this is because the leaders are slower owing to the high competition for resources. As a result, the chance of the leader(s) on a core filling up the victim cache within an epoch reduces. We can thus conclude that the performance penalty due to the victim cache filling up is negligible, and is greatly outweighed by the gain achieved due to it by allowing a speculative L1 cache at the leader.

6.4.4. NoC and Performance. Experiments were performed where the leader-checker proximity was factored into the arbiter heuristics. These, however, did not offer significant gains as far as performance was concerned. We thus did not find the NoC to be a bottleneck in our experiments. Being a 2-D torus structure, the distance between the leader and checker is around 2 hops on an average. Due to spending lesser time in the network, the packets encounter negligible congestion. Bandwidth utilization statistics are provided in Table V. The forwarding filters at the caches greatly reduce the load on the NoC. An average of only around 0.56 flits are forwarded by a core each cycle. Also, as discussed in Section 6.2, long phases of coherent behavior in typical workloads result in a stable mapping of leaders and checkers. This enables smooth non-jittery traffic in the NoC, and reduction in clogging of routers.

Further investigations were performed by increasing the NoC hop latency and changing the NoC topology. The observations in the first experiment are shown in Figure 19. As the hop latency is doubled from 2 to 4 cycles, the effect on the slowdown is negligible. When further doubled to an unrealistic value of 8 cycles, we see an increase of about 8% in the runtime (*high* load scenario). Thus, we see that for all reasonable operating values of the hop latency, the slowdown remains unperturbed. This is in line with the view of the leader and checker threads' execution as one long pipeline. An increased inter-hop latency is consequently hidden by the pipelining effect.

Figure 20 shows the effect of changing the NoC topology to a mesh. We see that there is no marked increase in the slowdown. The increase in runtime is limited to around 2%. This clarifies that the NoC is not a bottleneck in the system. These experiments further bear testimony to the scalability of the system.

However, as reported in Section 6.3, the impact of NoC usage on the power consumption is not negligible. If power is to be made the focus, or if the network is inadequately designed (lower link width, or simpler topology), then metrics such as the proximity between the leader and the checker can be factored into the arbiter heuristics. This will open up interesting avenues for further exploration.

6.4.5. Discussion on FluidCheck's Scalability. Experiments were performed with 32 and 64 core systems and no marked increase in the slowdowns were observed (within 1% of the 16 core case). However, while applying FluidCheck to much larger chips (say one with 256 cores), we may face NoC related issues. The distances between a leader and the corresponding checker core can be quite large. This may have undesirable effects on the performance and the power consumption of the system. In such a scenario, a tiling approach may be adopted. Going back to the 256 core (16×16) example, we recommend it be logically partitioned into tiles of 16 cores (4×4) each, and schedule threads on these tiles. The slowdowns observed should be the same as the 16-core chip case.

6.4.6. Time to Detection Study. A fault injection experiment was performed to determine the time taken to detect the occurrence of a soft error. Random bit flips were introduced in the latches in the latches of the fetch, decode and execute stages. The average time to detect the induced error was 129.17, 114.89 and 166.23 cycles respectively (medium workload, SP-UALF policy).

7. CONCLUSION AND FUTURE WORK

Manycore processors, with multi-threading capabilities, present an attractive platform for reliable execution in soft-error prone environments. In this context, FluidCheck was introduced – a highly flexible redundant-threading framework, that seeks to improve the throughput of the system by attempting a balanced consumption of resources across cores. As compared to the mean slowdown of 47% and 37% achieved by the seminal prior works SRT and CRT respectively (*medium* load), the architecture and arbiter schemes discussed in this paper achieve total coverage with a mere 27% penalty in performance.

Single-threaded workloads were studied in this work. *FluidCheck* can be extended by incorporating the techniques described in [Rashid and Huang 2008] to allow adding multi-threaded workloads to the mix. The *FluidCheck* architecture opens up interesting avenues for further exploration. In the future we wish to explore designs with more cores, more threads per core, and more elaborate NoCs.

REFERENCES

C. Acosta, F.J. Cazorla, A. Ramirez, and M. Valero. 2009. Thread to Core Assignment in SMT On-Chip Multiprocessors. In *SBAC-PAD*.

T.M. Austin. 1999. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Micro*.

D. Bernick, B. Bruckert, P.D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. 2005. NonStop®advanced architecture. In *DSN*.

S. Chatterjee, C. Weaver, and T. Austin. 2000. Efficient checker processor design. In *Micro*.

A. El-Moursy, R. Garg, D.H. Albonesi, and S. Dwarkadas. 2006. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *IPDPS*.

J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. 2013. L1-bandwidth Aware Thread Allocation in Multicore SMT Processors. In *PACT*.

M.A. Gomaa and T.N. Vijaykumar. 2005. Opportunistic Transient-Fault Detection. *SIGARCH Comput. Archit. News* (2005).

W. Huang, K. Rajamani, M.R. Stan, and K. Skadron. 2011. Scaling with Design Constraints: Predicting the Future of Big Chips. *Micro* (2011).

S. Hukerikar, K. Teranishi, P.C. Diniz, and R.F. Lucas. 2014. An evaluation of lazy fault detection based on Adaptive Redundant Multithreading. In *HPEC*.

H. Jeon and M. Annavaram. 2012. Warped-DMR: Light-weight Error Detection for GPGPU. In *MICRO*.

A.B. Kahng, B. Li, L. Peh, and K. Samadi. 2009. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration. In *DATE*.

R. Kalayappan and S.R. Sarangi. 2013. A survey of checker architectures. *Comput. Surveys* (2013).

S. Kumar and A. Aggarwal. 2008. Speculative instruction validation for performance-reliability trade-off. In *HPCA*.

S. Li, J. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Micro*.

G. Malhotra, P. Aggarwal, A. Sagar, and S.R. Sarangi. 2014. ParTejas: A Parallel Simulator for Multicore Processors. In *ISPASS*.

P. Montesinos, W. Liu, and J. Torrellas. 2007. Using Register Lifetime Predictions to Protect Register Files against Soft Errors. In *DSN*.

S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. 2002. Detailed design and evaluation of redundant multithreading alternatives. *SIGARCH Comput. Archit. News* (2002).

S. Parekh, S. Eggers, and H. Levy. 2000. *Thread-Sensitive Scheduling for SMT Processors*. Technical Report. University of Washington.

H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel®Itanium®Programs with Dynamic Instrumentation. In *Micro*.

M. Prvulovic, Z. Zhang, and J. Torrellas. 2002. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA*.

M.W. Rashid and M.C. Huang. 2008. Supporting highly-decoupled thread-level redundancy for parallel programs. In *HPCA*.

M.W. Rashid, E.J. Tan, M.C. Huang, and D.H. Albonesi. 2005. Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance. *PACT* (2005).

J. Ray, J.C. Hoe, and B. Falsafi. 2001. Dual use of superscalar datapath for transient-fault detection and recovery. In *Micro*.

S.K. Reinhardt and S.S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *ISCA*.

E. Rotenberg. 1999. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing*.

S.R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. 2015. Tejas: A Java Based Versatile Microarchitectural Simulator. In *PATMOS*.

A. Settle, J. Kihm, A. Janiszewski, and D. Connors. 2004. Architectural support for enhanced SMT job scheduling. In *PACT*.

L. Spainhower and T.A. Gregg. 1999. IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective. *IBM J. Res. Dev.* (1999).

P. Subramanyan, V. Singh, K.K. Saluja, and E. Larsson. 2010. Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors. In *DATE*.

Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. 2000. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *ASPLOS*.

S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi. 2008. *CACTI 5.1*. Technical Report. HP Laboratories.

D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ACM SIGARCH Computer Architecture News*.

D.M. Tullsen, S.J. Eggers, and H.M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*.

X. Vera, J. Abella, J. Carretero, and A. González. 2010. Selective replication: A lightweight technique for soft errors. *ACM Trans. Comput. Syst.* (2010).

J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. 2014. Real-world Design and Evaluation of Compiler-managed GPU Redundant Multithreading. In *ISCA*.

Huiyang Zhou. 2006. A Case for Fault Tolerance and Performance Enhancement Using Chip Multi-Processors. *CAL* (2006).