On-demand Thread-level Fault Detection in a Concurrent Programming Environment

Jian Fu*†, Qiang Yang*†, Raphael Poss*, Chris R. Jesshope*, Chunyuan Zhang†
*Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands
Email: {j.fu, q.yang, r.poss, c.r.jesshope}@uva.nl
†School of Computer, National University of Defense Technology, Changsha, China
Email: {fujian, cyzhang}@nudt.edu.cn

***Abstract*—The vulnerability of multi-core processors is increasing due to tighter design margins and greater susceptibility to interference. Moreover, concurrent programming environments are the norm in the exploitation of multi-core systems. In this paper, we present an on-demand thread-level fault detection mechanism for multi-cores. The main contribution is *on-demand redundancy*, which allows users to set the redundancy scope in the concurrent code. To achieve this we introduce *intelligent redundant thread creation and synchronization*, which manages concurrency and synchronization between the redundant threads via the master. This framework was implemented in an emulation of a multi-threaded, many-core processor with single, in-order issue cores. It was evaluated by a range of programs in image and signal processing, and encryption. The performance overhead of redundancy is less than 11% for single core execution and is always less than 100% for all scenarios. This efficiency derives from the platform's hardware concurrency management and latency tolerance.**

I. Introduction

Nowadays, multi-core systems are mainstream and the number of cores integrated in a processor will increase due to the inevitable technological progress. However, these trends also make the future of multi-core processors increasingly susceptible to both hard and soft errors. The shrinking of feature size leads to more manufacturing defects, process variations, and early lifetime failures [1]. Also, the reduction of design margins and the transistor's threshold voltage can increase the soft error rate of certain noise environments dramatically [2].

Redundancy is a classic solution for tolerating faults [3]. The key practical issue is how and at what level to apply redundancy in multi-core processors. Spatial redundancy is not the most efficient approach, due to its large area and energy overhead [4]–[7]. Also, it lacks flexibility when redundancy is not necessary. In contrast, temporal redundancy has its advantages in area overhead and flexibility but is limited by the duration of error, which means that it cannot detect hard or soft errors lasting longer than the interval of two thread copies [8]–[11].

Although multi-core processors have a higher vulnerability, they also provide natural extra hardware for fault tolerance. So multi-core processors based thread-level redundancy (TLR) techniques, which force two copies of a semantic thread to run on different cores, combine the fault coverage of spatial redundancy with the efficiency of temporal redundancy [12]–[14]. They can detect both hard and soft errors without adding extra hardware. However, these TLR techniques focus on single-threaded environments. Much less attention has been paid to the design issues in concurrent programming environments. It is not a trivial extension to migrate TLR from a single-threaded environment to concurrent programming environments. In particular, already concurrent software natively uses multiple cores simultaneously, and TLR must thus manage simultaneous occupation of cores by concurrent software and its replicates.

Meanwhile, many emerging applications allow the discarding of individual sub-computations with small qualitative impact [15]. This is a form of "intrinsic" fault tolerance which does not require extra architectural support. Also, many commodity systems do not need high reliability and some can tolerant faults to some extent, e.g. the odd pixel in video decompression can not be noticed. Even for mission critical systems, fault tolerance is not needed all the time, especially when performance and energy are key considerations. In other words, most applications do not require investment in the cost of full redundancy. And the most elegant solution is that fault tolerance is provided only when necessary, i.e. redundancy on demand.

To address these two problems, this paper presents an on-demand, thread-level, fault detection framework in a concurrent programming environment. This vertical framework includes support from the programming model, compiler, ISA and micro-architecture, but none of the changes are on the critical path of the system. In other words, the fault tolerance mechanism is independent of the existing system. As with the other TLR mechanisms in multi-core processors, we propose two thread copies of a semantic thread, which are called master thread and redundant thread respectively. They are forced to run on fixed paired, adjacent cores in order to detect both hard and soft errors. The sphere of replication [9] includes the entire pipeline, register file and L1 cache of each core, under the assumption that the memory system is fault-free. Also, we adopt a relaxed input replication technique, so as to avoid significant changes to the existing cache hierarchies for redundant execution. The divergences of any load value, induced by relaxed input replication, can be corrected or recovered by the same mechanism employed for fault detection

[14]. Finally, the output of two thread copies is compared to check whether it is correct. Here we focus on the output to memory (i.e. *store*) only, I/O operations are not considered yet.

In summary then, the main contribution of this paper is *on-demand redundancy* in the context of a concurrent programming environment. When and where a program should be duplicated to give high reliability can be specified by users or the run-time environment. This makes the system more efficient and flexible as the granularity of redundancy is a thread, which can be specified anywhere in the hierarchical, multi-threaded programming environment.

In order to achieve this we have introduced *intelligent redundant thread creation and synchronization*. Usually, we want to exploit concurrency as much as possible in concurrent programming environments. So hierarchical concurrency and thread independence are two features of concurrent programming environments that affect the design of thread duplication. We only allow the master thread to create both master and redundant threads' child threads in order to avoid a thread explosion in a hierarchical, concurrent programming model. Additionally the master thread maintains synchronization between redundant parent and child. This requires some changes to the existing system.

The rest of this paper is organized as follows. Section II introduces some general conceptions of concurrent programming environment. Section III presents the on-demand redundancy framework. Section IV describes the strategy for redundant thread creation and its synchronization with its redundant parent. The output comparison is described in section V. Section VI shows the results analysis and related experimental setup. We discuss related work in section VII, and finally, section VIII provides our conclusions.

II. CONCURRENT PROGRAMMING ENVIRONMENT

The key idea in concurrent programming environment is concurrency, which is a property of systems in which several workloads are executing simultaneously, and potentially interacting with each other. Conceptually, such systems can be modeled as a fork-join queue within a closed queueing network. Each incoming workload is split into N tasks at the fork point, and each of these tasks queues for service at a concurrent service node before joining a queue for the join point. It is possible to have a nested fork-join queue in order to exploit concurrency as much as possible. So a concurrency tree is a very appropriate structure that to show the concurrency organization in a concurrent programming environment. The concurrency tree is akin to process tree in unix. It is inevitable that there are communications (or synchronizations) between sibling nodes or parent and child nodes.

Considering the concurrency tree in a concurrent programming environment, we present an on-demand redundancy strategy based on the granularity of a given node in the concurrency tree and the sub-tree it defines. In this strategy, we duplicate any sub-tree defined by the user over which it is required to implement fault detection. In order to avoid a node explosion in this hierarchical concurrency tree, we only allow the master node to create child nodes of both master and redundant nodes. However, the synchronization between redundant parent and child is broken as the redundant child is created by master parent node, which means the redundant parent node can not be terminated. To address it, we pair master and redundant nodes at the fork point, and the synchronization between redundant parent and child can be achieved via the master parent node.

Besides the concurrency expression, another important conception in concurrent programming environment is space scheduling, which is spreading concurrent software workloads to hardware parallel execution resources. Space scheduling can be done either in software or in hardware. Here, we fix the hardware parallel execution resources in pairs to execute master and redundant workloads in order to detect both hard and soft errors. For example, if HW1 and HW2 are a fixed pair, then all the master workloads in HW1 have their copies executed in HW2 and vice versa.

The System Virtualization Platform (SVP) [16] is a concurrent programming environment designed by the Computer Systems Architecture group in University of Amsterdam. It is a set of system services and language interfaces for the exploitation of concurrency on many-core processor chips. In this concurrency model, each concurrent node in a concurrency tree is called a *thread*, and all the same level concurrent nodes that are *created* by a parent node are called a *family* of threads. Every thread can create families of its own, making the model hierarchical. The hardware execution resource is called a *place*, which is allocated at run-time prior to the family being created. Any communication and synchronization between threads does not happen via memory, but through special hardware-supported channels called *globals* and *shareds*. Global channels are written once by the parent thread and are available for being read to each thread in the family. Shared channels are defined between every consecutive pair of threads in the family. The SVP memory model is conceptually a single, flat address space with a restricted consistency model, allowing greater freedom for the implementation.

In this paper we have implemented our fault detection framework in the Microgrid execution platform [17]–[19]. It is a multi-core system that provides dedicated logic able to coordinate single-issue, in-order hardware multi-threaded RISC cores into computation clusters on chip. It has a highly scalable and configurable many-core architecture. Its machine language provides new instructions to manage concurrency, which are described by the SVP model. SVP combines dataflow synchronization with imperative programming, aiming for the efficient use of parallelism in general-purpose workloads. The platform is provided with the SL programming language [20], which is a interface language to program this platform. SL is designed as an extension to the standard C language (ISO C99/C11). It includes primitive constructs to bulk create threads, bulk synchronize on termination of threads, and communicate using word-sized data-flow channels between threads. It is intended for use as the target language for higher-

level parallelizing compilers. Although our fault detection framework should be general for all concurrent programming environments, we will use some dedicated syntax or concepts in the description of fault detection mechanism that come from the Microgrid, SL language, and the assembly language of their cores, since the Microgrid is the experimental platform in this paper.

III. On-demand redundancy

We sketch a simple SL [20] function to explain how to implement on-demand redundancy and its related support from the programming model, compiler and ISA. Listing 1 shows a simple SL function and how it is augmented with on-demand redundancy support and compiled to a sequence of instructions. This is only part of a complete program. The programming model uses the notation *sl_create* to dynamically define a family of threads on an index range. The on-demand redundancy additions to SL and assembly languages when creating the redundant function are highlighted.

Listing 1 (a) shows the simple summation function where *sl_create* is augmented to use the parameter '*ftmode*' (i.e. fault tolerance mode). There are two functions: *sum* is responsible for the summation and *t_main* is for creating a family of *sum* threads to expose the concurrency explicitly in software. The parameter '*ftmode*' is the only attribute that is added to the programming model. It is used to determine the redundancy state of the family that will be created. There are three states:

NORMAL – Redundancy is not necessary, so its child family will not be duplicated;

START – Current thread is not duplicated, but its child family will be duplicated. It is the beginning of the redundancy scope in the program;

REDUNDANCY – Current thread is duplicated, and its child family will also be duplicated.

Compiler support is relatively straightforward. It compiles the SL code in listing 1 (a) to assembly showed in listing 1 (b, c, d) according to the fault tolerance mode. For readability, we use symbolic register names rather than numbered registers in the assembly. Listing 1 (b) shows the code generated in normal situation (i.e. without fault tolerance support). The instruction *allocate* attempts to allocate the computing resources (i.e. place, which is a set of cores in the implementation of Microgrid.) on which to create the family of threads according to its parameters: place identifier and some other flags in R_{place} and R_{flag}. If successful, the family identifier (*fid*) of the allocated family entry on the first core will be written into R_{fid}. Then the properties of this family will be set through *set* instructions, such as the total number of threads, and the number of hardware threads per core. Finally, this family of threads will be created using the instruction *create* [21].

Listing 1 (c) presents the assembly generated at the redundancy scope's beginning. Generally, the compiler will duplicate all the instructions relating to family creation and initialization, such as *allocate, set, create, put* etc. However, there are some issues that need to be considered. The instruction *allocate* for redundant family should generate a different place identifier compared to the master family, as we dispatch each to a different core for hard error detection. The master and redundant families need to be connected through the instruction *pair*. The most important point is that the current thread, which is executing the assembly of listing 1 (c), only synchronizes with its master child family. Also the data returned from the child family of threads read by the instruction *gets* will not be verified against its redundant child family therefore the thread reads from its master child family only. This is because the redundancy scope began from the current thread's child and hence the current thread is out of redundancy sphere and is not be protected.

Listing 1 (d) gives the assembly created within the redundancy scope. Because it is within the scope, it means that both redundant and master threads will execute it. So we do not extend the assembly as for the redundancy start point, which is executed by one thread only. Three instructions, which are italic and bold in the code, are extended to ISA for an elegant thread duplication solution in our concurrent programming environment. More details about thread duplication strategy within redundancy scope and these extended instructions will be discussed in section IV.

The fault tolerance mode only can transit from *NORMAL* to *START*, to *REDUNDANCY*. And *START* is a temporary state that occurs once in the transition from *NORMAL* to *REDUNDANCY*. The transition is irreversible, as it is not worth supporting a reversal mode transition in hardware. Note that the reverse transition occurs naturally on termination of the family created in fault tolerant mode. In effect, what this scheme does is to label a node in the concurrency tree and every thread below this label is within the sphere of redundancy. This can be done at any number of nodes to create redundancy for those critical regions. Thus the correct redundancy scope can be easily achieved with much lower design complexity at the software level by arranging thread structure properly.

IV. Intelligent redundant thread creation and synchronization

As described in section III, the thread duplication strategy cannot be migrated without change from start point to the body of redundancy because only one thread executes the thread duplication code at the start point, but in the body of redundancy, both master and redundant threads execute the thread duplication. Obviously, we cannot allow both master and redundant threads to duplicate its child family, as this will lead to redundant child family explosion.

As a result, allocations in redundant threads are not allowed. The master thread will allocate double resources as before but the redundant thread will do nothing (i.e. allocate in figure 1). However, usually a redundant parent thread has to wait for the synchronization of its children to continue or end its execution (i.e. sync in figure 1). We use the master parent thread to bridge this gap. There are two family identifiers returned for every allocation in the master parent thread. One will be sent to the redundant parent thread so the redundant parent thread

```
int array[len];
sl_def(sum, void, sl_shparm(int, s))
{
    sl_index(i);
    sl_setp(s, sl_getp(s)+array[i]);
}
sl_enddef

sl_def(t_main, void)
{
    sl_create(,,start,limit,step,
                block,,ftmode,sum,
                sl_sharg(int, s));
    sl_seta(s, 0);
    sl_sync();
    int result = sl_geta(s);
}
sl_enddef
```
(a)

```
<main>:
allocate    R_place,R_flag,R_fid
setlimit    R_fid,limit
setblock    R_fid,block
create      R_fid
puts        0,R_fid,0
sync        R_fid,$1
mov         $1,$31
gets        R_fid,0,$1
release     R_fid
```
(b)

```
<main>:
allocate    R_place,R_flag,R_fid
allocate    R_rplace,R_flag,R_rfid
pair        R_fid,R_rfid
setlimit    R_fid,limit
setlimit    R_rfid,limit
setblock    R_fid,block
setblock    R_rfid,block
create      R_fid
create      R_rfid
puts        0,R_fid,0
puts        0,R_rfid,0
sync        R_fid,$1
mov         $1,$31
gets        R_fid,0,$1
release     R_fid
release     R_rfid
```
(c)

```
<main>:
allocate/r  R_place,R_flag,R_fid,R_rfid
pair        R_fid,R_rfid
rmtwr       R_rfid
setlimit    R_fid,limit
setblock    R_fid,block
create      R_fid
puts        0,R_fid,0
sync        R_fid,$1
mov         $1,$31
gets        R_fid,0,$1
release     R_fid
```
(d)

Listing 1. A simple summation function with fault tolerance related parameter '*ftmode*' (a), its original assembly without fault tolerance attributes (b), the assembly at the redundancy's start point (c), and the regular assembly within the redundancy's scope (d).

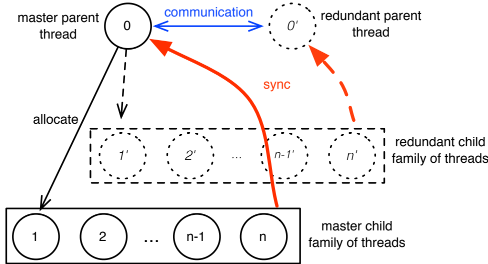

Fig. 1. Intelligent redundant thread creation and synchronization

eventually gets its resource from the corresponding master parent thread without doing any allocation. This is achieved through the communication protocol shown in figure 1. The synchronization link is built between redundant parent and child family. This is the main idea behind the intelligent redundant thread creation and synchronization. Because of this, all the other instructions related to thread creation can run without any modification.

This strategy is supported in the ISA and at the micro-architecture level. Three new instructions are added for thread duplication and its communication. However, these new instructions have a different operation in master and redundant threads, as shown in table I. The instruction *allocate/r* in master thread is responsible for allocating cores for both master and redundant child families. The redundant thread only waits on its register for the redundant child family's identifier, which is sent by the master thread. Then the instruction *pair* in the master thread makes the master and redundant child family known to each other, which is necessary for the communication between master and redundant threads shown in figure 1. Finally, the redundant child family's identifier is returned to the master parent thread and will be sent to redundant parent by the instruction *rmtwr*. There is no operation for *pair* and *rmtwr* in the redundant thread. Up to now, two places for the master and redundant child families are allocated, and the master and redundant parents receive master and redundant child family identifier, respectively. Considering the assembly in listing 1 (d), all the subsequent thread creation related instructions are depend on its family identifier (R_{fid}). So all these instructions do not need to be modified to fit thread duplication, as we have already connected parent thread and child family in both the master and redundant group, although both master and redundant child family are allocated by the master parent thread.

V. Output Comparison

Like the other thread-level redundancy techniques, we must compare the results of master and redundant threads to detect faults. To achieve this, a comparison buffer is added between the L1 D cache and secondary memory. This buffer is shared by a core pair. As stated above, the core and private L1 cache are contained in the sphere of replication in our fault detection framework. This means that the other components by definition are out of the sphere, such as L2 cache and off-chip memory. These are assumed to be fault free. Each output (i.e. *store*) should be stored to both L1 and comparison buffer first, then compared in comparison buffer before committed

TABLE I
NEW INSTRUCTIONS AND THEIR OPERATIONS

Instruction	Master thread	Redundant thread
allocate/r	Send an allocation message to the place with flag. The master family identifier will be returned to R_{fid}, the redundant family identifier to R_{rfid}.	Set the output register R_{fid} to pending; Send R_{fid}'s index to master thread.
pair	Send a message to the destination place, which will pair master and redundant families.	No-op
rmtwr	Write the redundant family identifier to the redundant thread's R_{fid}, which was pended by instruction *allocate/r* in redundant thread.	No-op

to secondary memory. The operation of data input (i.e. *load*) is the same as before: the data come from secondary memory to L1 D cache. In this paper, we do not address the issue of supporting output comparison in I/O.

As the redundant thread knows its master thread's identifier, which is explained in section IV, the comparison buffer is organized as a number of sets that are indexed by both its core and master thread identifier. Furthermore, the master thread writes data to the set specified by its identifier in the comparison buffer, which is owned by the core it runs on. A store in the redundant thread writes data to the set specified by its master thread. For example, if the master thread (thread identifier is t_i) runs on core i, and redundant thread (thread identifier is t_j) runs on core j, then the stores of these two threads will be sent to set t_i in the comparison buffer specified by core i.

Each set is a FIFO queue, as thread instructions are executed in order in the Microgrid platform. This means that all the stores in one thread will be appended to its dedicated set and compared in order. Each entry in a set has three fields: address, value and flag.

When a set of comparison buffer receives data, the data will be written to the set directly if the set is empty. Otherwise, it will check whether the data and the head of the set come from the same thread. If they are, the data will be appended to the end of the set. If they come from different threads, which mean master and redundant thread, then they will be compared. A fault is detected when they do not match. If they do match, which shows the results are correct, the data will be popped from the set and written to secondary memory.

Any read requests coming from L1 cache, will first search the set indexed by the current thread. Data will be returned if it is available, otherwise the read request will be sent to secondary memory as usual. The comparison buffer does not change the memory protocol, which means it can be used with the Microgrid's various memory interconnects.

VI. EXPERIMENTAL RESULTS

A. Experimental paltform

The thread-level fault detection framework is implemented in the Microgrid. Figure 2 illustrates a Microgrid chip with a

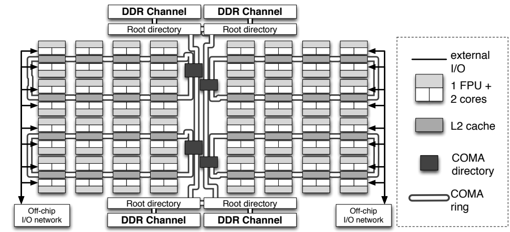

Fig. 2. The Microgrid chip with 128 cores

TABLE II
THE SPECIFICATION OF MICROGRID

Components	Specification
Core	• Alpha ISA with SVP extensions • In-order pipeline of 6 stages • 1024 integer registers and 512 float registers • 1.0 GHz frequency
L1 D Cache	• 16 sets with 4-way set associative • 4KB capacity • Write through
L2 Cache	• 512 sets with 4-way set associative • 128KB capacity • Write update • Shared by 4 cores via a snooping bus
Distributed Cache Structure	• Each sub ring has a directory and 8 L2 caches • 4 root directories each connects to a DDR3-1600 channel mapped to a DRAM bank • Ring directories and evenly distributed root directories from the top ring

configuration of 128 cores, which is the base platform we use for all experiments. All cores on chip are organized in a linear partitionable network for resource allocation and concurrency management within an allocated cluster. It is worth mentioning that there is a custom distributed cache protocol derived from [22], [23]: memory stores are effected at local L2 cache and updates are propagated and merged with other copies. Upon an explicit barriers or bulk creation or synchronization of threads, the update acknowledges must be counted by thread or family to ensure memory consistency for the programming model. The hardware parameters most relevant to this paper are shown in table II, more details can be found in [17]–[19].

The simulator currently executes benchmarks in which the redundancy scope is the complete benchmark, however, we only execute small kernels not large applications. Selective redundancy and fault coverage are left to future work. The six benchmarks include image processing kernels, FFT and encryption are used to evaluate the thread-level fault detection technique shown in table III.

B. Results

We use the performance of non-redundant benchmarks run in Microgrid as the baseline and we call the non-redundant benchmark the base benchmark, and benchmark with complete redundancy scope the redundant benchmark. We try to evaluate the performance overhead when redundancy is introduced. The

TABLE III
DESCRIPTION OF BENCHMARKS

Category	Benchmarks	Instructions count	Description
Image processing	convolution	28 million	The size of original image is 800*400. It is zoomed in to 6400*3200, and reduces each pixel from 24 bits to 8 bits in grey conversion.
	zoom in	29 million	
	grey conversion	11 million	
Signal processing	FFT	19 million	Use a 64K phase lookup table and butterfly reduction.
Encryption	rc4	0.05 million per stream per core	The problem size is scaled with number of cores and hardware threads per core, i.e. 1Kbyte stream per hardware thread.
	seal	0.5 million per stream per core	

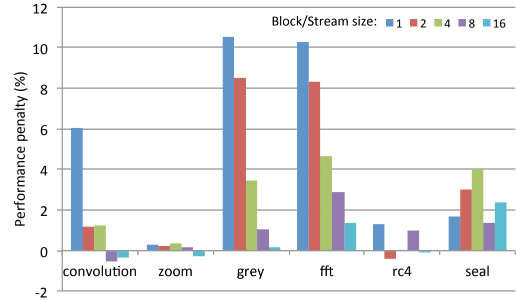

Fig. 3. Performance penalty of single core base benchmarks

performance overhead of a redundant benchmark is also called performance penalty in this paper, which is defined as follows:

$$performance\ penalty = (\frac{t_{redundant}}{t_{base}} - 1) * 100\%$$

And $t_{redundant}$ is the execution time of redundant benchmark, t_{base} is the execution time of base benchmark.

The master and redundant threads are always distributed to different cores that are fix paired. We should distinguish the experimental results of the base benchmark run on single core from many cores. For the single core base benchmarks, another core is used for the redundant execution, which means a redundant hardware resource is added. However, there is no redundant hardware resource added for many-core base benchmarks. For example, if the base benchmark has 2 independent threads in total, and thread 0 and 1 run on core 0 and 1, respectively. Then its redundant benchmark still runs on core 0 and 1 with master thread 0 and redundant thread 1 run on core 0, and master thread 1 and redundant thread 0 run on core 1. It shows that more resource contention occurs in many-core redundant benchmarks.

Single core. In figure 3, the bars from left to right for each benchmark correspond to block size or number of streams (i.e. the number of threads per core). This ranges from 1, 2, 4, 8 to 16, all later figures are organized like this. Figure 3 shows that the performance penalty is less than 11% for all scenarios, which is mainly caused by output comparison. As the number of threads per core increases, the performance penalty reduces because the latency of the output comparison can be tolerated by the hardware multithreading mechanism. It can be seen that in some cases the performance of the redundant benchmark is better than that of the base benchmark. We believe this is because the comparison buffer moderates peak traffic rates onto the memory network. We have observed a degradation of performance at large numbers of threads on these benchmarks due to correlated stores in many threads saturating the memory network and causing a higher latency on synchronization.

Many cores. The performance penalty varies in the many-core benchmarks (figure 4). Compared to the base benchmark, the performance penalty of the redundant benchmark is less than 100% even though it executes double the number of instructions using the same single-issue, in-order core resource. This is mainly attributable to the hardware multithreading and its ability to tolerate latency.

The block size in figure 4 is based on the base benchmark. For example, a scenario of N cores with a block size of M indicates that the base benchmark is executed on M threads in each of N cores. However, the redundant benchmark is executed in N cores with 2*M threads. The additional M threads are used by the redundant copy. The performance penalty of 1 thread per core is always smaller than other situations. It is less than 44% even in the computationally intensive *convolution* program. It also relatively stable for different numbers of cores. Going from 1 to 2 threads per core gives a significant speedup and it seems that a speedup of between 1.5 to 2 can be achieved for all the benchmarks here.

The general trend we can find in figure 4 is: the more cores are used, the less the performance penalty. Because for a given base benchmark, the efficiency becomes lower when more resources are used. The relation between block size and performance penalty is not so clear, as the distributed cache system is involved. Usually, the larger the block size, the greater the performance penalty, because of that, the efficiency of the base benchmark is quite high when a bigger block size is set. We also see some unexplained results with a block size of 16 in *zoom* and *rc4*. We believe the reason is again as described above in the case of the single core benchmark. The more cores used the greater impact of the correlated stores on synchronization costs. So it appears that the comparison buffer mitigates this problem. Note that in any case the redundant benchmark has twice the computational complexity (operations per byte stored) as the base benchmark, as only one store to the L2 cache occurs for two writes from each core.

We take *rc4* using 64 cores as an example and runs some diagnostics to try to explain this anomaly in more detail. We found that the messages in the cache ring decreased by 55% (read-related by 75% and write-related by 44%) in the redundant benchmark compared to the baseline, even though the total number of read requests has doubled in the redundant benchmark (the number of write requests is same). We believe that because the comparison buffer delays each write it will

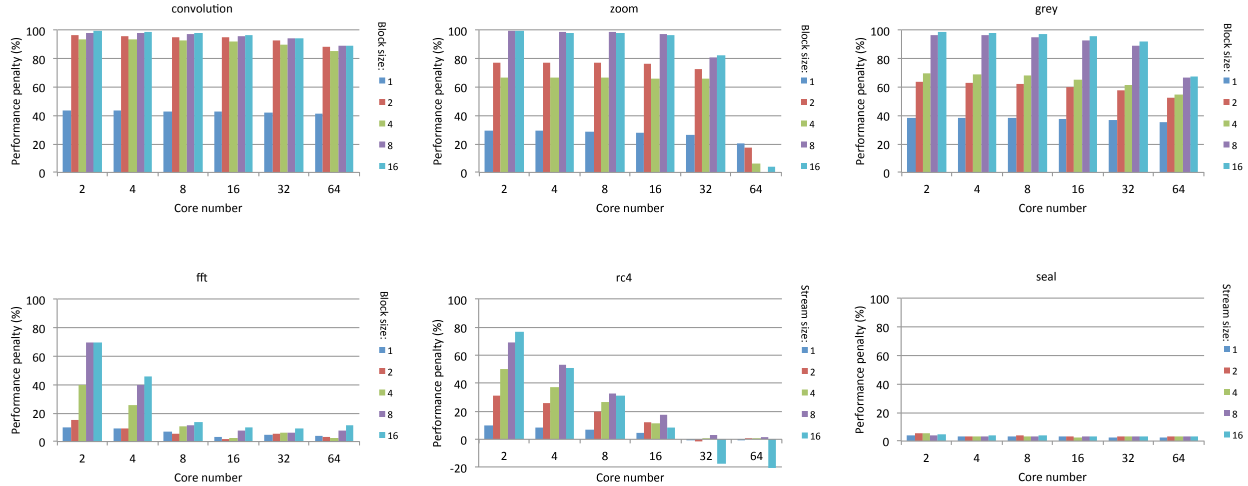

Fig. 4. Performance penalty of many cores base benchmarks

relieve the congestion of cache ring. Remember that the cache policy on L1 is non-allocating write through with a write update policy at L2. So the comparison buffer acts like a write combining buffer delaying and aggregating writes. In effect it allows more writes to be achieved without generating an update message in the ring. Similarly as the combining buffer delays the eviction of the cache line it can also increase the hit rate to existing lines. Note that the encryption benchmarks have little locality between threads, as each is an independent stream. Thus the redundant benchmark is able to improve both the write and read hit rate and this is why the performance penalty goes lower with the increasing of block/stream size in 64 cores of *zoom* and *rc4*. Moreover, the *rc4* redundant benchmark can have better performance than baseline even though it has double number of threads and twice the work to do. It is in effect highlighting the inefficiency of the distributed cache system, which is sensitive to the quantity and distribution of messages at high load.

The three image processing redundant benchmarks have a higher performance penalty compared to the other benchmarks, as they are highly parallel benchmarks with exceptional locality and hence have high efficiency. Although FFT is also high parallel, its communication is non-local, which means that its efficiency is constrained by contention in the cache ring. So again, its redundant benchmark has much smaller penalty than image processing ones. The encryption benchmarks have sufficient parallelism (one stream per hardware thread) but lack locality as each stream is independent. Thus frequent evictions of data cause the efficiency of the base benchmark to be low and the overhead of the redundant benchmark to be correspondingly low.

VII. Related work

The research on thread-level redundancy (TLR) became popular following the introduction of SMT [24], as it can benefit from the higher resource utilization when master and redundant threads can co-exist within the processor. AR-SMT [8] proposed executing two copies of the same program in an SMT environment first. Then, SRT [9] improved on the performance of AR-SMT using slack fetch and branch outcome queue based on speculation and cache locality. SRTR [10] extends SRT to implement fault recovery.

Due to heavy overhead of full fault coverage, some extensions of redundant multithreading (RMT) have explored the ability to only partially replicate the master thread, such as Slipstream [25], SlicK [26], and the opportunistic transient fault detection in [27]. Meanwhile, with the emergence of multi-core technology, DCC [1], CRT [12], CRTR [13], and Reunion [14] apply RMT to CMPs. They found that it is fewer overheads than performing RMT in single SMT core, as CMPs mitigate the resource contention in single core. Also most RMTs in CMPs provide both hard and soft fault detection.

However, all of above techniques are aimed on single-thread applications, or single thread/core for the CMPs, which is inefficiency in multi-core environment. In order to make full use of multi-core systems, especially with multi-threading to achieve latency tolerance, the program model must be shifted to a concurrent programming methodology. We propose on-demand thread-level fault detection framework within concurrent programming environment, which gives us the ability to define the redundancy scope required by users or algorithm designers. It is a flexible and efficiency fault detection mechanism, as it is configurable on the basis of demand. [28] also targets parallel programs but only focuses on micro-architecture support for output comparison under the condition that a large number of threads existed at same time. It is not concerned with thread duplication in a hierarchical concurrent environment, which is the main contribution of this paper.

VIII. Conclusion

Fault tolerance will be inevitable as multi-core systems become mainstream. There is no doubt that concurrent program-

ming is also an important opportunity to improve the efficiency of multi-core processors. In this context, this paper presents and implements an on-demand fault detection framework that has been added to an existing multi-threaded, many-core chip emulation.

By necessity the framework has been co-designed across multiple layers, including programming model, compiler, ISA and micro-architecture. This cross-layer cooperation makes the fault detection mechanism much more flexible and efficient. It also makes on-demand redundancy possible. In addition, in order to target a hierarchical concurrency model, this paper proposes an intelligent redundant thread creation and synchronization technique. This technique allows the master thread to create both master and redundant child families, and connects the redundant parent and child family. It not only avoids redundant threads explosion, but also keeps the synchronization channel between redundant parent and child.

Finally the paper presents some results on the overhead of executing the redundant thread across a range of kernel benchmarks with a range of concurrency resources (threads per core and numbers of cores). The results show that for a single core benchmark, where additional resources are brought into play, the performance penalty for redundancy is less than 11%. For the the many-core situation, the penalty is never larger than 100% even though the the redundant benchmark has twice the dynamic instruction count. This is attributed to the latency tolerance of hardware multithreading, especially in situations where the efficiency is poor on the baseline. In some cases we even show that the redundant benchmark has a better performance than the baseline and confirm that it places a smaller load on the shared resources due to a modified scheduling of writes to L2 cache.

REFERENCES

[1] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 317–326.

[2] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, "Impact of CMOS process scaling and SOI on the soft error rates of logic processes," in *IEEE Symposium on VLSI Technology*, 2001, pp. 73–74.

[3] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *C. E. Shannon and J. McCarthy, editors, Automata Studies*, 1956, pp. 43–98.

[4] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 microprocessor design," *Micro, IEEE*, vol. 19, no. 2, pp. 12–23, mar/apr 1999.

[5] D. Jewett, "Integrity S2: a fault-tolerant Unix platform," in *21st International Symposium on Fault-Tolerant Computing (FTCS)*, jun 1991, pp. 512–519.

[6] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 12–21.

[7] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *Proceedings of the 1996 IEEE Aerospace Applications Conference*, 1996, pp. 293–307.

[8] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *29th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, june 1999, pp. 84–91.

[9] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000, pp. 25–36.

[10] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002, pp. 87–98.

[11] E. Schuchman and T. Vijaykumar, "BlackJack: hard error detection with redundant threads on smt," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, june 2007, pp. 327–337.

[12] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002, pp. 99–110.

[13] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 98–109.

[14] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: complexity-effective multicore redundancy," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 223–234.

[15] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010, pp. 497–508.

[16] C. Jesshope, "The SVP model," University of Amsterdam, Tech. Rep., March 2011. [Online]. Available: https://notes.svp-home.org/svp38.html

[17] T. Bernard, K. Bousias, L. Guang, C. Jesshope, M. Lankamp, M. van Tol, and L. Zhang, "A general model of concurrency and its implementation as many-core dynamic risc processors," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, july 2008, pp. 1–9.

[18] Q. Yang, C. Jesshope, and J. Fu, "A micro-threading based concurrency model for parallel computing," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, may 2011, pp. 1668–1674.

[19] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, and C. Jesshope, "Apple-CORE: Microgrids of SVP cores," in *Proc. 15th Euromicro Conference on Digital System Design (DSD)*, September 2012.

[20] R. Poss, "SL—a "quick and dirty" but working intermediate language for SVP systems," University of Amsterdam, Tech. Rep., August 2012. [Online]. Available: http://arxiv.org/abs/1208.4572

[21] M. Lankamp and R. Poss, "SVP extensions to the Alpha ISA," University of Amsterdam, Tech. Rep., March 2011. [Online]. Available: https://notes.svp-home.org/mgsim2.html

[22] L. Zhang and C. Jesshope, "On-chip COMA cache-coherence protocol for microgrids of microthreaded cores," in *Proceedings of the 2007 Conference on Parallel Processing (Euro-Par)*, 2008, pp. 38–48.

[23] T. D. Vu, L. Zhang, and C. Jesshope, "The verification of the on-chip COMA cache coherence protocol," in *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, 2008, pp. 413–429.

[24] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, june 1995, pp. 392–403.

[25] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream processors: improving both performance and fault tolerance," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000, pp. 257–268.

[26] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi, "SlicK: slice-based locality exploitation for efficient redundant multithreading," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 95–105.

[27] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic transient-fault detection," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005, pp. 172–183.

[28] M. Rashid and M. Huang, "Supporting highly-decoupled thread-level redundancy for parallel programs," in *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 393–404.