Software-Based Fault Tolerance for the Maestro Many-Core Processor

John Paul Walters, Robert Kost, Karandeep Singh, Jinwoo Suh, Stephen P. Crago
University of Southern California / Information Sciences Institute
{jwalters, kost, karan, jsuh, crago}@isi.edu

Abstract—The current generation of radiation-hardened general-purpose processors, such as the RAD750, lag far behind their commercial counterparts in terms of performance. To combat this, a new many-core processor was designed that would allow space applications to leverage up to 49 general-purpose processing cores for high performance space applications. The Maestro processor, based on Tilera's TILE64 chip, is the result of this effort. Maestro is rad-hard by design, but there is still the possibility of both hardware and software errors.

In this paper we describe fault tolerance strategies for today's multi-core processors with a focus on the Maestro many-core space processor. Processors such as Maestro represent an attractive target for software-based fault tolerance research as the multiple cores, memory controllers, etc. provide ample resources for replication, checkpoint and rollback, as well as hybrid or application-specific fault tolerance strategies.

We focus specifically on our ongoing work in providing a software fault tolerance layer to augment Maestro's existing radiation tolerance. Our work includes kernel-level checkpoint/rollback, process and thread-level redundancy, and a distributed heartbeat implementation. These strategies can be used independently or cooperatively, depending on an application's requirements. [12]

TABLE OF CONTENTS

1. INTRODUCTION

With current generations of radiation hardened microprocessors lagging far behind their commercial counterparts in performance systems, spacecraft designers have historically been forced to choose between performance and reliability when selecting flight processors. The result is limited on-board processing.

The Maestro processor solves this by providing 49 high-performance processing cores. Maestro is based on Tilera's TILE64 processor [1], but adds support for hardware-assisted floating point and is radiation hardened by design.

Despite radiation hardening, there is still a chance of both software and hardware failures. To help mitigate these potential failures, we have applied common fault tolerance strategies used in high performance computing (HPC) and embedded systems to the Maestro processor.

While traditional clusters have only limited exposure to cosmic rays, they are still subject to increasing failure rates. This is particularly true in the Petaflop era, with systems composed of tens to hundreds of thousands of cores. These systems employ strategies such as checkpoint and rollback, duplication and replication in order to tolerate and recover from failures as necessary.

Maestro, with its 49 tiles has redundant resources, similar to a small computational cluster. The abundance of resources allows us to adapt existing fault tolerance techniques to the Maestro processor in order to provide an additional fault tolerance layer to applications. In this paper we describe our infrastructure that consists of thread-level replication, process replication, heartbeat monitors, and checkpoint/rollback.

These techniques are designed to allow a developer to cope with detected radiation-induced faults within the processor rather than preventing faults from occurring. Faults may still occur at any level of the software stack – the operating system, hypervisor, library code, or the user's application. The strategies described in this paper allow computing to continue despite these upsets.

The techniques do not attempt to detect or correct any errors that do not propagate to the application-level or manifest as an unresponsive processor. Further, the current fault tolerance layer cannot be used to detect and correct errors in external components, for example sensors, networks, etc.

[1] 978-1-4244-7351-9/11/$26.00 ©2011 IEEE
[2] IEEEAC paper #1564, Version 2, Updated January 10, 2011

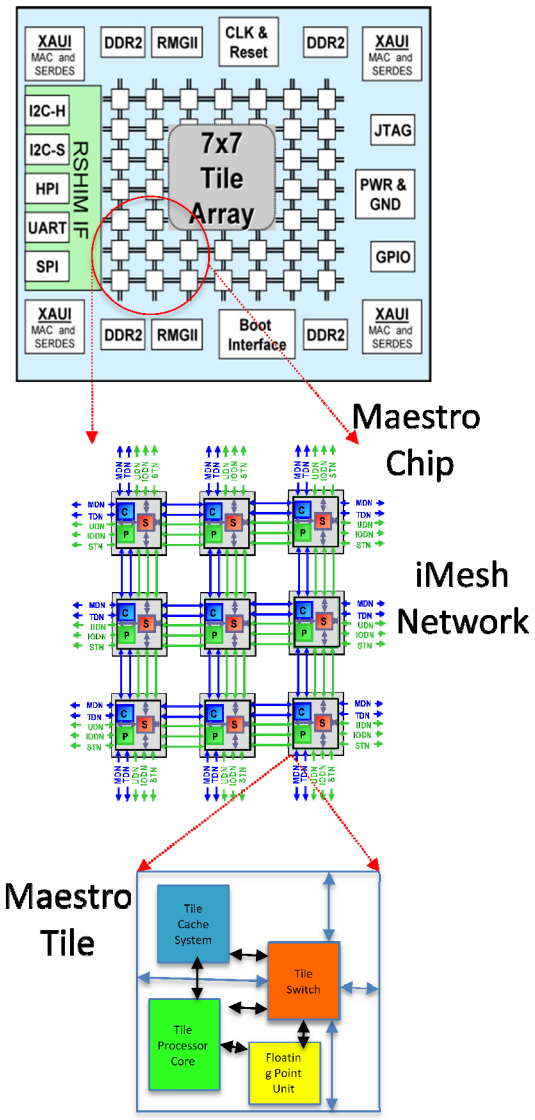

Figure 1—Block diagram of the Maestro architecture.

The remainder of this paper is organized as follows: in Section 2 we describe the Maestro and TILE64 architectures that our solutions target. In Sections 3 - 6 we describe the implementation of our fault detection and mitigation strategies. In Section 7 we describe the related work that has influenced the design of our fault tolerance work. Finally in Section 8 we present our conclusions.

2. MAESTRO ARCHITECTURE

The Maestro processor is a radiation hardened by design processor for space that is based on the TILE64 processor by Tilera (see block diagram in **Error! Unknown switch argument.**). Maestro is composed of a grid of 7x7 general purpose processing cores. Currently each core executes at 300 MHz, though the device is still under test and subject to change.

Each Maestro core is composed of a multi-level cache hierarchy that includes an 8 KB L1 cache and a 64 KB L2 cache. A virtual L3 cache, consisting of each tile's L2 cache, is extended over all of Maestro's 49 tiles.

Cores within the Maestro are interconnected by a 2D mesh network known as the iMesh™. The iMesh is collectively responsible for routing data from main memory to individual tiles as well as to I/O interfaces and inter-core communication.

All of the solutions described in this paper target both TILE64 and Maestro. However, because of a lack of hardware all of our timings are reported on the TILE64 architecture.

3. PROCESS-LEVEL REPLICATION (PLR)

We begin our discussion with process-level replication. PLR is based on the notion that although faults occur at the microscopic hardware level, the faults that really matter are the ones that affect software correctness. This means that some (perhaps many) single event upsets are benign and will go unnoticed. However, other errors are malignant in that they affect the output of an application. Shifting the focus from hardware correctness to software correctness, changes the domain of fault tolerance to allow the system to ignore benign errors and only concentrate on the errors that affect the correctness of the application. PLR leverages the inherent redundancy of the Maestro architecture to detect data mismatches before they propagate to an application's output.

PLR Implementation

The PLR implementation creates two redundant processes for a single application, similar to TMR. The same application is thus run on three separate tiles. An interposition library intercepts calls to open (e.g., open(), open64(), fopen(), creat(), and creat64()). If the mode for the open call is 'w' or 'wb', the PLR library creates output file(s) with the corresponding redundant pid attached. If the open call is a read, the open call is unchanged and functions as usual. We currently do not support read/write or append modes in our implementation.

After all processes have completed, the output files are compared. If all files match, then the redundant output files are removed and the output file will be stored using its original name. If the files do not match, then a majority rule will apply to determine the correct output as in triple modular redundancy. The temporary redundant output files will be saved so the user can view the differences. In an embedded system, the system would need to have a policy to react to these differences.

This PLR implementation uses shared memory to provide bookkeeping for opening files. All other calls are considered out of scope and are not interposed. It is assumed they will behave as expected. Shared memory, barriers, and

```
FILE *fopen (const char *path, const char *flags)
{
      if (!strcmp (flags, (const char *) "r")) {
        return _real_fopen (path, flags);
      }
      else if (!strcmp (flags, (const char *) "w")) {
        …
        mutex_lock (&filename_info->mutex);
        …
        snprintf (filename_info->  char_array
            [filename_info->counter], MAX_SIZE,
            "%s",  newpath);
        filename_info->counter++;
        …
        mutex_unlock (&filename_info->mutex);
        return _real_fopen (newpath, flags);
      }
      return _real_fopen (path, flags);
}
```

Listing 2—An Interposed fopen.

locking mechanism are allocated and maintained using the Tilera Multicore Components (TMC) Library [2].

An application is compiled by defining preprocessor macros such as, "-Dmain=plr_main and –Dexit=plr_exit." This allows the PLR library to hook into the user's main function to allow the master process to initialize shared memory, barriers, and other bookkeeping structures. Likewise, it will hook into the user's exit function, so a *fini* function can conduct the appropriate output file voting and cleanup. If there is no user exit function, then the *fini* function is called directly. A barrier is called right before the voting occurs to instruct the master process to wait until all the redundant processes have finished the main application and have finished writing any output files before it calls the *fini* function.

If there is a mismatch in any of the output files, then each file is saved so the user can inspect the errors. However, if there are not any mismatches in any of the output files, the temporary output files are copied back, using the original filename.

Function Interposition

Function interposition is used to intercept calls to shared libraries [3]. It is useful for several reasons. First and foremost, interposition allows a shared library to be accessed without having real access to the source code. This is beneficial since many times, the source code is not available or it could be too complex or cumbersome to modify. Second, no recompilation of the shared library or executable is required.

```
void _init (int argc, char **argv) {
      ...
      _real_fopen = dlsym (RTLD_NEXT, "fopen");
      dl_check (dlerror ());
      ...
}
```

Listing 1—Interposition init stub.

Table 1—PLR overhead for image compression.

Phases	Cycles	% of Total Cycles
init	17,939,566	0.42
interposition	13,395,244	0.61
fini	363,815,968	16.63

All of this is possible by instructing the linker to reference a specific function definition at run-time, before referencing the default library. In Linux and hence, on TILE64 and Maestro, the dlsym() function is used. Assigning "RTLD_NEXT" as the first argument, directs dlsym() to return the address of the next occurrence to a specified function. The LD_PRELOAD environment variable works in conjunction with dlsym(). Undefined function calls are referenced first in the shared library that is set to LD_PRELOAD, before the application searches the default library path.

In Listing 1, dlsym is called to find the next occurrence of fopen(). This creates a wrapper to fopen() which can then be manipulated as shown in Listing 2. In the PLR implementation, if the "read" flag is set, then we call the real fopen() without changing any functionality. However, if the "write" flag is set, we create a new file based on the process id, as well as perform other bookkeeping tasks. Finally, the real fopen() is called with a different filename.

Experimental Results

The overhead involved in creating two redundant processes is minimal and is application-independent. There are three main causes for overheard. First, in the *init* phase (called before main), the master process allocates the appropriate shared memory, barriers, and mutexes, as well as forking the redundant processes. Second, since each call to open a file is interposed, there is minimal overhead in locking a mutex and changing the filename. The amount of overhead during the *interposition phase* for a single call to fopen() for the master process is roughly 50,000 cycles. The remaining overhead is incurred during the *fini* phase. This is where the majority of overhead is contained, since this is the phase that performs the voting and cleans up the output files.

Table 1 shows the average amount of overhead expected for three total processes (1 master and two slave) while

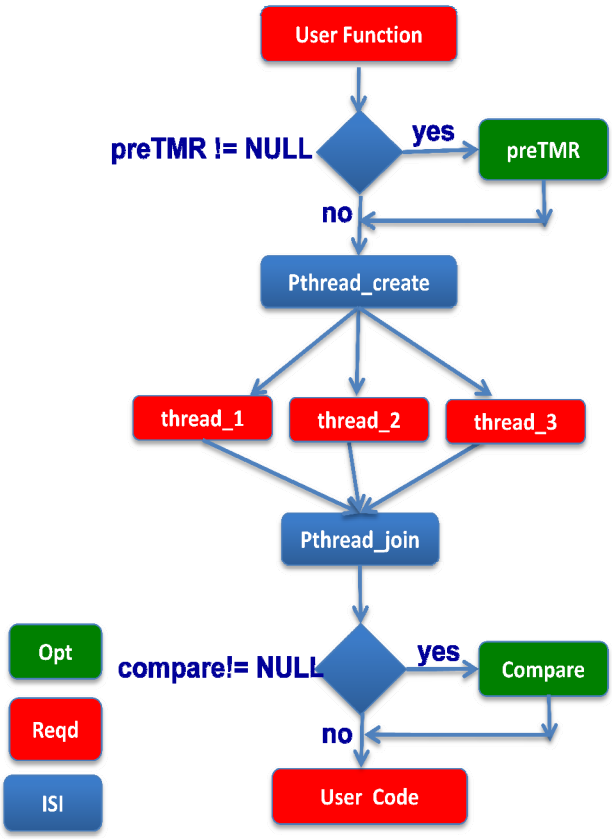

Figure 2—TLR implementation

compressing a 904K image using OpenJPEG2k [4] on Tile64. The amount of overhead in the *fini* phase is directly proportional to the number and size of the output files written; hence the *fini* phase is application-dependent. The total number of cycles for running OpenJPEG2k to compress the image in lossless encoding was 2,187,742,009 cycles, including both the compression and PLR overhead.

4. THREAD-LEVEL REPLICATION (TLR)

In the previous section we described process-level replication, which replicates an entire process in a user-transparent fashion. When the overhead of process-level replication is too high, we also provide a thread-level replication package named TLR. TLR similarly leverages the redundancy of the Maestro architecture, but replaces processes with threads at the cost of decreased transparency. The advantage, however, is that users are provided fine-grained control of the replication process.

The TLR tool is an implementation of software based fault tolerance [5,6] using N-version programming [7] with thread replication. It replicates a user function on three different threads pinned on different cores (tiles) on the Maestro and TILE64 chip. The user function is executed in parallel on these threads. TLR collects the outputs from the three threads and then compares the output values using TMR to determine if any error has occurred and provides the correct output.

TLR Implementation

The framework for TLR is shown in Figure 2. The red boxes show user functions. Blue boxes show the TLR library components that we have implemented, and the green boxes are optional functions that the user may provide. TLR has been implemented in C and is available as a library (libtlr.a) that may be linked directly to user-code.

The user provides a pointer to the function to be replicated using TLR, as well as the inputs and outputs to the function. In this implementation we use the Pthreads library to spawn three threads with copies of the function to be protected. These three threads are pinned to separate tiles and each thread is given its own copy of the inputs. The user may specify the tiles where these threads are placed, or absent input from the user, TLR will pin the threads to tiles linearly. Running the threads on tiles that are physically separated from one another decreases the chances of a single error affecting multiple tiles.

After pinning, the threads are executed in parallel, and TLR waits for their completion. The outputs from these threads are then compared. TLR determines if any errors have been detected in the function's output. If an SEU has occurred, this would likely lead to output from one thread not matching the other two. In such a case corrected outputs are returned back to the calling function, after the threads have been merged. In the case of more than one error, when none of the outputs match, TLR returns an error.

TLR can currently handle user functions with inputs and outputs that are stored contiguously automatically. When inputs and outputs are stored non-contiguously, the user must provide custom pre-TMR and compare functions.

User-supplied pre-TMR and compare functions replace the built-in memcpy() and memcp() comparisons that are used by default. To use the pre-TMR function, the user supplies a function that handles the replication of user inputs. Similarly, to use a custom compare function, users must supply a function that will replace the memcmp() functionality by performing a custom validation step.

TLR API

TLR consists of two functions. The main TLR functionality is implemented by the *threadReplicate* function. A second function, named *insertErrors,* is provided for testing purposes. The parameters for both TLR functions are described in Table 2.

Table 2—TLR API

Function	Parameter	Description
threadReplicate	*userFunction*	Function that the user wants to protect using TLR.
	Inputs	Pointer to the user function's inputs.
	Outputs	Pointer to the user function's outputs.
	input_size	Provides the size of the inputs in bytes.
	output_size	Provides the size of the outputs in bytes.
	input_replicate	Indicates whether input data needs to be duplicated, with each of the 3 threads getting its own copy of the inputs.
	host_cpu	Array that indicates on which tiles to pin the threads. If NULL, then this pinning is done automatically.
	userPreFunction	Optional user supplied function that is called before the user function is replicated, to set it up for TLR.
	userPostFunction	Optional user supplied function that the user wants to call after the three threads have finished computing the user function, to perform output comparisons in complex cases.
insertErrors	*data*	Pointer to the data array where an error is to be inserted.
	thread_num	TLR thread where an error is to be inserted.

The function *threadReplicate* replicates the user function to be protected and insertErrors is used for developer testing.

The function *insertErrors* inserts artificial errors in user data to test functionality and correctness of the implementation. It was also used for implementing the *threadReplicate* function itself. The current implementation was tested by inserting a single error in the input data, in which case correct output was determined using TMR. The second test was performed by inserting multiple errors in the input data, in which case 'threadReplcate' tells the user that correct outputs cannot be determined using TMR for multiple errors.

TLR Performance

Table 3—TLR overheads

Average time for creating 3 threads	4,850,853 cycles
Overall replication and joining costs for an empty function	7,066,721 cycles
Output comparison costs for an empty function	103,696 cycles

Table 3 shows the performance numbers for TLR in terms of the overheads associated with using it. These performance numbers were obtained by implementing TLR for an empty function. The average time for creating 3 pthreads for replication is 4.8 million cycles. This cost is application independent. The overall cost for initializing the 'threadReplicate' function, creating and replicating three threads, pinning them to specific tiles, and then executing and joining the three threads for an empty function is 7 million cycles and for that experiment it takes 103K cycles in the output comparison phase. These two performance numbers are application dependent.

5. HEARTBEATS

While the previous sections have described replication strategies to help mitigate data errors, it is also necessary to know whether a processor is continuing to execute or has failed.

Heartbeats are a mechanism that allows each CPU to track the status of any other CPU participating in the heartbeat network. Heartbeats allow a CPU to check whether another CPU is working properly or has failed. We implemented the heartbeat mechanism on Tilera's TILE64 and Maestro.

Heartbeat Model

In our implementation, a single master process creates multiple child processes. The child process can create one or more grand-child processes. Our implementation imposes no limits on the levels of the process hierarchy, except that the total number of processes should be fixed and known at initialization time.

The implemented library provides a scorekeeping mechanism by assigning a unique ID to a process and updates the scoreboard at a user-defined interval. It also provides a mechanism to reply to a scoreboard tracker.

Heartbeats are valid after the first process completes its initialization function, i.e., if the first process dies before its initialization is complete, the heartbeat checks are not guaranteed to return a correct status.

Table 4—Heartbeat events by cycle.

Cycles	CPU	Events
0	cpu1	init start
2,380,316	cpu1	init end
2,475,563 (A)	cpu1	not started
3,850,043	cpu2	init start
5,875,461 (B)	cpu0	init start
7,124,093	cpu3	init start
9,956,705	cpu2	init end
10,088,917 (C)	cpu2	alive
10,414,746 (D)	cpu0	init end
10,774,425	cpu0	alive
101,804,275	cpu3	init end
101,927,858	cpu3	alive
702,640,140	cpu1	alive
710,238,263	cpu2	alive
9,104,765,302 (E)	cpu0	kill
9,104,846,269	cpu1	alive
9,111,987,861	cpu2	alive
9,112,901,312	cpu0	alive
9,204,126,277	cpu3	alive
9,806,407,306 (F)	cpu1	dead
9,812,153,605	cpu2	dead
9,813,059,881	cpu0	dead
9,904,898,959	cpu3	dead
15,408,024,831	cpu1	resurrect
15,408,093,412 (G)	cpu1	alive
15,413,333,100	cpu2	alive
15,414,136,278	cpu0	alive
15,506,578,033	cpu3	alive

Implementation Details

Our heartbeat implementation uses a scoreboard in shared memory. The shared memory is allocated using Linux shmget(). The shared memory is allocated by a "root" process and initialized, while other processes share the allocated space. In case the shared memory is not protected by hardware, other protection mechanisms such as TMR may be used in order to safeguard the scoreboard.

The shared memory scoreboard has an entry for each process. The entry is used by each process to store its latest time stamp and is in turn used by all other processes to check the health of the process.

When a process is spawned, it first either allocates shared memory space (root) or obtains the allocated shared memory space information (non-root). Then, it creates a daemon thread that is responsible for heartbeat updates. The daemon is pinned to the same CPU as its parent process using Tilera's TMC library (tmc_cpus_set_my_cpu()). The daemon wakes up periodically and stores the current time stamp to its corresponding entry in the scoreboard.

When a CPU wants to check another CPU's health, it reads the data in the scoreboard that corresponds to the target CPU using a provided API. If the time stamp data in the entry is not updated within the timeout window, it knows that the process is dead. If the content in the scoreboard is updated recently, it knows that the corresponding process is alive.

There is predefined slack time before a CPU is proclaimed to be dead to avoid false alarm due to delay of update operations. For example, if there is an interrupt, it can cause delay in the update operation. The slack time prevents these normal delays from marking a CPU as failed. The slack time must be larger than the sum of the scoreboard update period and interrupt service times that can happen during one scoreboard update period.

In addition to these functions, we implemented two auxiliary functions that allow a programmer to test application code. The first function simulates the failure of one or more CPUs by stopping the update process for the affected CPUs. The second function resumes the scoreboard updates for the effected CPUs.

Heartbeat API

int hb_init(float heart_beat_interval, int num_processes):

This is used to initialize the heartbeat function after fork. The parameter num_processes is the total number of processes that will share the heartbeat mechanism. The variable heart_beat_interval is the interval between scoreboard updates for a CPU and the unit is second. It returns hb_id that is in rage of 0 to (num_processes - 1).

int hb_check(int hb_id):

This function checks if a process is alive. It returns 0 if alive, -1 if dead, or 1 if the heartbeat daemon has not started yet.

int hb_check_all():

This function checks if all CPUs are alive. It returns 0 if all alive, -1 if one or more are dead, or 1 if one or more have not started yet.

void hb_aux_kill(int hb_id) and
void hb_aux_resurrect(int hb_id):

These functions are not to be used in a real application. These are used for testing of user applications. They force the process identified by hb_id to stop or resume updating the scoreboard.

Experimental Results

In our experiments, four CPUs are used, and the time interval between heartbeat daemon update is set to one second. In application code, after initialization, our tests repeatedly check the heartbeat of cpu0.

Since it is not practical to make a CPU faulty artificially, we used the auxiliary functions to simulate the failure of cpu0. Hb_aux_kill() is used to stop updating the scoreboard, and hb_aux_resurrect() is used to resume the scoreboard updating.

Table 4 shows the important events in the execution to show the operation of heartbeat. As the results show, each CPU first initializes before checking the heartbeat status of cpu0. Initially cpu0 returns "not started" to cpu1 meaning that the daemon has not yet started. Note that cpu0 is checked first at cycle 2,475,563 (A) and the initialization starts at cycle 5,875,461 (B).

However, at cycle 10,088,917 (C), when cpu2 checks the status of cpu0, it shows that cpu0 is alive. Note that although the end of initialization is shown at cycle 10,414,746 (D), the actual update starts earlier than the return of the initialization function and that is why the cpu0 shows the status as alive even before the init end event.

At cycle 9,104,765,302 (E) we simulate the failure of cpu0. The failure is discovered at cycle 9,806,407,306 (F). This is due to the slack time that is given to each CPU in order to avoid false positives.

At cycle 15,408,093,412 (G), the cpu0 is resurrected by using the hb_aux_resurrect() function. After that, when cpu1 checks the status of the cpu0, it shows the status as alive.

We measured the number of cycles of checking the status of a CPU and updating scoreboard. The average number of cycles was 4,195 cycles for checking and 15,353 cycles for updating scoreboard.

6. CHECKPOINT/ROLLBACK

While the previous sections focused primarily on the detection of errors in a running processor, checkpointing and rollback focuses primarily on the mitigation of detected errors, whether those errors are detected through a heartbeat mechanism, replication/duplication strategy, or by any other means.

Simply put, checkpointing is the act of capturing the state of a running computation such that it may be restarted at the point of capture at a later time. Checkpointing is commonly implemented at one of three levels:

(1) Application-level – a precompiler is used to transform a user's source code into a self-checkpointing application; or, state-saving code is manually written by the programmer in an application-specific manner.

(2) User-level – a checkpoint-enabled library is linked or interposed that allows an arbitrary application to save its state either through a function call or an interrupt.

(3) Kernel-level – the operating system is directly modified to allow the state of a process to be saved and reloaded, typically using a system call. Alternatively, a loadable kernel module is used.

There are advantages to each of the approaches listed above. However, for our purpose, the kernel-level scheme was the most sensible approach. Checkpointing at the kernel-level avoids library and language issues that are present in checkpointing at the application and user levels. Further, checkpointing at the kernel level may be used in a manner that is transparent to the running application. That is, no recompilation is needed, nor should the running application initialize an interrupt or make a function call.

The disadvantage to a kernel-level approach is that the solution is highly kernel-specific and architecture-specific. The remainder of this section will describe our implementation and will provide a brief performance analysis.

Implementation

In our implementation we chose to leverage the existing Linux-CR v15 [8] work that is publicly available and actively maintained by a small community of Linux developers. The Linux-CR work consists of several components: a user-space process hierarchy rebuilding tool, a set of architecture-independent kernel patches, and several architecture-specific functions that are responsible for capturing the unique state of a particular architecture. We will begin with a description of the simplest case of checkpointing – where a single process checkpoints and later restores itself. We will then extend the discussion to describe the parallel case, where a process hierarchy is captured and restored. We refer to these cases as "self" checkpointing and "external" checkpointing, respectively.

Self Checkpoints—We ported the architecture-independent Linux kernel patches to Maestro's production kernel, based on the 2.6.26 Linux kernel. These patches are responsible for checkpointing a process' memory segment (VMAs), libraries, process binary, file state, etc. They are also responsible for coordinating the restart of checkpoints.

Most of the porting effort was directed towards adding support for the TMPFS RAM disk file system used by the Maestro's Linux kernel, and handling the vDSO (virtual dynamic shared object) VMAs that are automatically

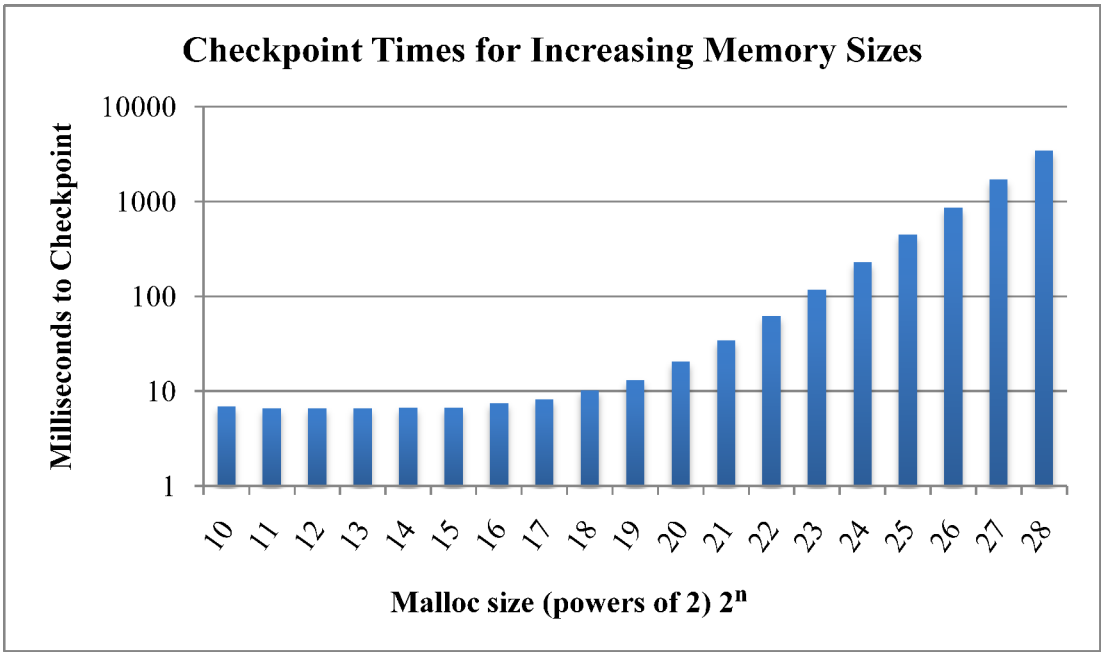

Figure 3—checkpoint times for increasing memory sizes.

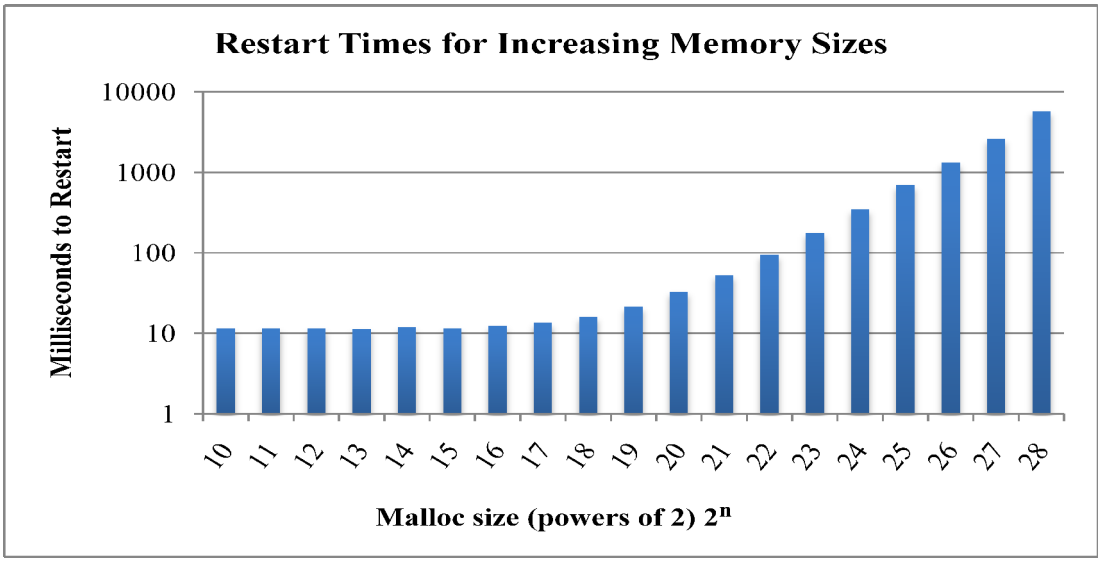

Figure 4—Restart times for increasing memory sizes.

mapped into the address space of every process. The challenging in handling the TMPFS RAM disk is that files appearing in the RAM disk otherwise appeared to the Linux-CR system as a shared memory segment instead of a set of files. This was particularly troublesome when trying to checkpoint the state of libraries. A simple flag attached to the file system memory segments allowed Linux-CR to differentiate between file system and program memory segments.

Tilera's vDSO is used to provide each process with an efficient signal return path. Checkpointing processes with a vDSO required selectively checkpointing all VMAs below the vDSO, which Tilera's kernel reliably maps to the same address in every process. Because the content of the vDSO doesn't change we can safely ignore it while checkpointing and leave the default vDSO in place during the process' restart.

While capturing the memory segments is relatively architecture-independent, capturing the processor-specific elements is not. Specifically, the architecture must provide functions for capturing and restoring the state of all CPU registers. In the Maestro processor, this means capturing both the general-purpose register file as well as the FPU registers. Restoring the register state is nearly the reverse of saving

the register state. Once the process hierarchy has been rebuilt, registers are copied back into their Linux data structures, and computation resumes.

Linux-CR separates the restore process into both a user-space segment and kernel segment. The kernel segment is described above: memory segments and registers are restored from the saved process image. However, before they may be restored we must first rebuild the process hierarchy. In the case of a single process in "self" checkpointing mode, this is straightforward. We simply call the restart system call (sys_restart) from an arbitrary process, the checkpoint is read into the kernel, and the caller's process image is replaced by the checkpointed image before resuming computation.

External Checkpoints—The self-checkpointing work described above is effective for sequential applications because computation is, by definition, paused during the execution of a system call. Within the context of a single process checkpoint, there was no danger of memory or file system inconsistencies. Parallel applications, however, are different. If multiple independently executing processes are paused and checkpointed separately, inconsistencies arise, and computations cannot be reliably restored.

To solve this, Linux-CR adds an additional quiescing stage, where each process within the parallel hierarchy is frozen before checkpointing begins. The freezing is accomplished through the use of the Linux freezer subsystem that was backported from the public Linux-CR project to the Maestro Linux kernel.

Once an application has been frozen, checkpointing is simply a matter of iterating over all processes in the parallel hierarchy and capturing each process' individual state. Global state, such as shared memory is captured once, marked as checkpointed, and skipped over for the remaining processes.

This process is called "external" checkpointing, because unlike the self-checkpointing described earlier, external checkpoints are initiated outside of the processes that will be checkpointed. This means that a process need not be checkpoint-aware, recompiled, or modified in any way.

Restoring a parallel process is more complex than the sequential case. This is due to Linux-CR's design decision to rebuild process hierarchies in user space before invoking the restart system call. Unlike the parallel checkpointing case, where a single call to sys_checkpoint() would iterate over the entire process hierarchy, sys_restart() should be called by **each** process in the restored process hierarchy.

This is accomplished using a dedicated restart application. Where the self-checkpointing case only required an application to invoke sys_restart, the external checkpointing case requires the application to rebuild the entire process hierarchy by first cloning each process, and then calling sys_restart() from within each clone. We were able to directly leverage Linux-CR's restart application for this task.

Performance

In this section we describe the performance of our checkpoint-enabled kernel. We use two benchmarks to characterize our implementation. The first uses a single process in self-checkpointing mode with increasing memory image sizes, from 1024 bytes through 256 MB. We then measure the time to perform the checkpoint as well as the time to restart the checkpoint. These times are obtained by measuring the time spent in the system calls sys_checkpoint() and sys_restart(), respectively.

In the second test we use a parallel matrix multiply to measure the external checkpoint and rollback times. In this case, checkpoint time is measured by the time to freeze all running processes, execute the sys_checkpoint() system call and return to user space. The restart time is measured from the time the first process enters sys_restart() until the last process exits.

In Figure 3, we present the results of checkpointing a single process application with increasing memory sizes. The memory footprint of the application is controlled by calling malloc() to dynamically allocate various memory block sizes. As we show, initially there is little difference in checkpointing time between memory sizes of 1KB and 32KB. This reflects the typical layout of a process in Linux. A default heap is initially created for a process. When the dynamically allocated memory fits within that preallocated space, there is little change in size of the application. As the size of the heap increases, so does the time to checkpoint the process, shown in **Error! Unknown switch argument.**. At 256MB, for example, it takes 3.4 seconds to capture the process image.

We see a similar effect in restarts; however, as we show in Figure 4, restarts are consistently longer than checkpoints in self-checkpointing. This is because during the restart, memories must also be allocated in addition to being copied from disk. As the process is being rebuilt during the restart, VMAs (virtual memory areas) must be allocated for the process' restored memories. This is reflected in the restart times shown in Figure 4, where 256MB takes 5.7 seconds to restart.

In Figure 5 and Figure 6 we present the checkpoint and restart timings for a parallel matrix multiplication application. The size of the matrix was fixed at 1024x1024, and the number of tiles varied from 1 – 32 tiles, resulting in checkpoint sizes of 12.5–19.5MB. The checkpoint image increases due to the increasing number of processes/tiles participating in the computation. Each process has a base checkpoint that is independent of the shared state, and the increased checkpoint image size reflects that.

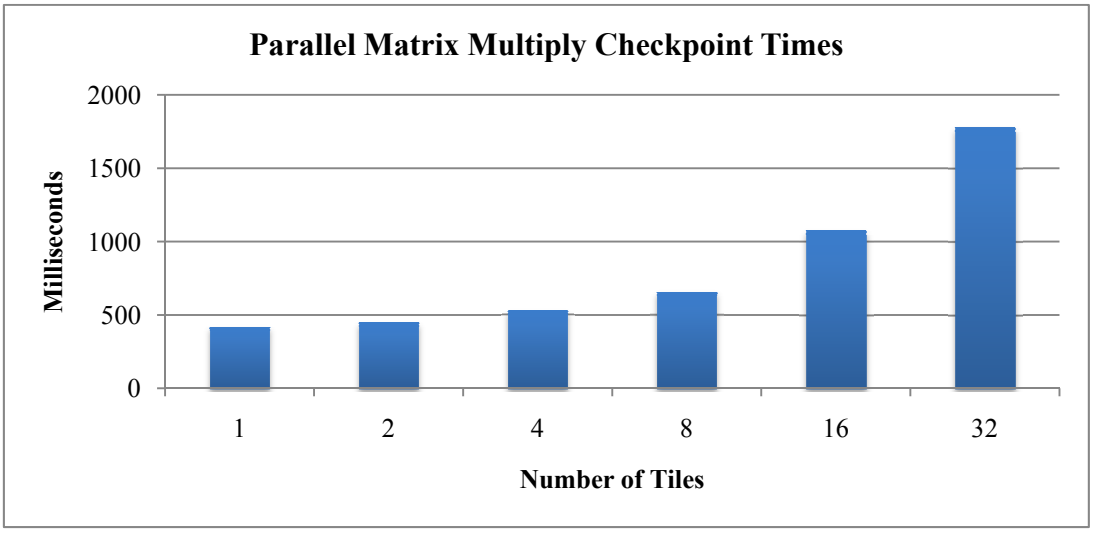

Figure 5—The time to checkpoint a parallel matrix multiply for increasing numbers of tiles.

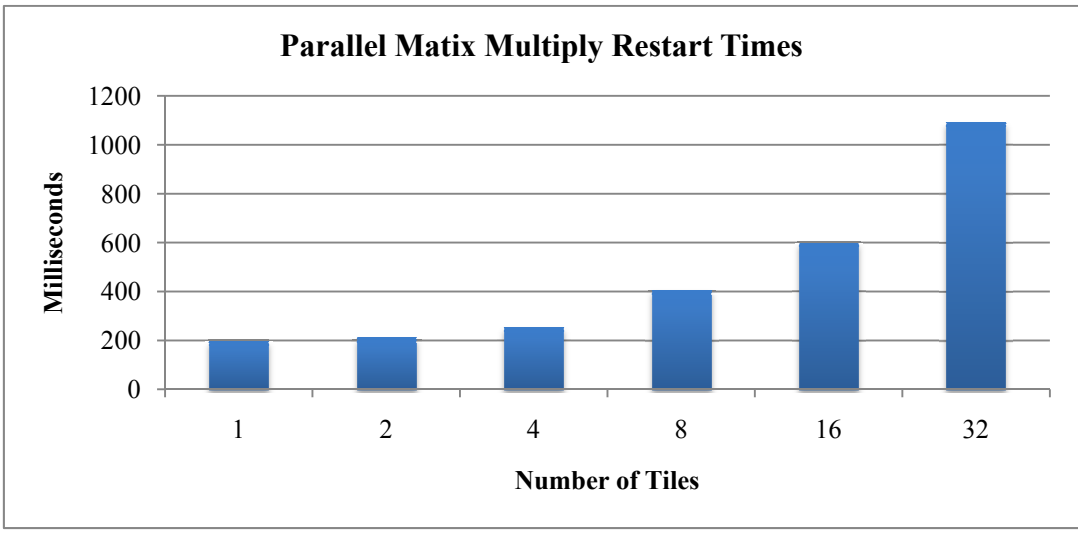

Figure 6—The time to restart a parallel matrix multiply with increasing numbers of tiles.

In Figure 5 we present the external checkpointing results. The increase in checkpointing time is a consequence of the process freezing step, which costs nearly 300 ms. Restart times, as shown in Figure 6, are comparable to those in Figure 4 (1 tile in Figure 6 corresponds roughly to a heap size of 2^{23} bytes in Figure 4). As the number of tiles participating in the computation increases, so too does the restore time due to the associated costs of restoring each process' process-specific data. Unlike the self-checkpointing case, restart times are consistently lower than checkpoint times in the case of external checkpoints. This, as described previously, is due to the added cost of freezing the processes.

7. Related Work

Fault tolerance, particularly in the areas of high performance computing has been well studied in the literature. Checkpointing is widely used and several public checkpoint/rollback implementations are available. BLCR [9] is perhaps the most well known implementation. BLCR is a kernel module, allowing it to be loaded and unloaded. The publicly available Linux-CR [8], and our port to the TILE64 and Maestro architectures, are kernel patches.

Checkpointing is also commonly used in embedded systems. In previous work we have demonstrated a working user-level checkpointing library for PowerPC 405 processors embedded within Xilinx V4FX60 FPGAs [10].

The Linux-HA project [11] is an open source project that maintains a set of building blocks for high availability cluster systems. It includes not only the heartbeat mechanism, but also other high level blocks such as cluster resource manager and cluster messaging layer.

An open source software package for heartbeat is SGI's FailSafe [12], which does not require the user's application to be rewritten or recompiled. It supports up to 16 nodes in a cluster.

The PLR prototype is a different implementation of PLR from previous work completed at the University of Colorado [13]. The conceptual ideas are the same, in that PLR focuses its replication scheme at the user address space level, but the implementation is different since they used a binary translator tool, PIN, for their PLR implementation. In this implementation, interposition is used to catch files open calls. Thus, syscalls are not virtualized nor are any other libc functions interposed besides file open calls.

The original PLR authors utilize a watchdog timer and a figurehead process to monitor the status of the redundant slave processes. System calls, input replication and output replication are all emulated. The authors also created a fault injector using the Intel binary instrumentation tool, PIN[14], to measure the efficacy of their PLR implementation.

The software techniques described in this paper are designed to allow an application to tolerate a radiation event and mitigate its effect. Other strategies, such as radiation hardening, seek to prevent the radiation-induced upset altogether. The RAD750 is a radiation hardened single board computer [15]. It is designed to be nearly immune to SEUs, however its performance lags far behind today's commercial non-radiation hardened processors. The techniques described in this paper attempt to bridge the gap between "soft" processors and radiation hardened processors

8. Conclusions and Future Work

In this paper we have described our fault tolerance implementation for the TILE64 and Maestro processors. Our next step is to improve the functionality of each component and to integrate them into a fully interoperable suite of tools.

Our PLR work currently targets single process applications. We intend to extend this work to multi-threaded applications and to characterize the performance impact of multi-threaded applications executing within the PLR environment. Further, we intend to add support to allocating redundant processes to specific locations within the tile grid. This will reduce the likelihood of an upset affecting multiple tiles. Such an approach may also improve memory and I/O performance by reducing contention to specific memory shims. Finally, we intend to extend our work to virtualize all system calls and to add support for a single figurehead process that will allow PLR to maintain POSIX compliance.

We intend to extend our TLR work to handle arbitrary inputs to the *threadReplicate* function. Currently *threadReplicate* requires all inputs to exist in a single contiguous memory address range. The same restriction is in place for outputs. A more robust I/O mechanism would allow TLR to be more easily integrated into existing applications, reducing the barrier to its use.

Finally, we intend to extend our checkpoint-enabled kernel to track the development of the Linux-CR project. This will provide a number of added features, including multi-threaded application support. Further, we hope to extend the Linux-CR work to allow checkpointing of the unique features of TILE64 and Maestro, for example portions of the iMesh as well as the DMA engine.

REFERENCES

[1] Tilera, *Tile Processor User Architecture Manual*, Nov. 2009.

[2] Tilera, *Application Libraries Reference Guide*, 2010.

[3] T. W. Curry. *Profiling and tracing dynamic library usage via interposition*. In USENIX Summer 1994 Technical Conference, pages 267–278, Boston, MA, June 1994.

[4] http://www.openjpeg.org/, 2010.

[5] T. Wilfredo, *Software Fault Tolerance: A Tutorial*, NASA Langley Technical Report Server, 2000.

[6] R. Guerraoui and A. Schipper, *Software-Based Replication for Fault Tolerance*, IEEE Computer, April 1997, pp. 68 - 74.

[7] A. Avizienis, *The N-Version Approach to Fault-Tolerant Software*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 290 - 300.

[8] Linux-CR, http://ckpt.wiki.kernel.org, 2010.

[9] J. Duell, P. Hargrove, and E. Roman, *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*. Berkeley Lab Technical Report (publication LBNL-54941), December 2002.

[10] J.P. Walters, M. Bucciero, M. French, *Radiation Hardening by Software for the Embedded PowerPC, Preliminary Findings* (abstract), ReSpace/MAPLD 2010.

[11] Linux-Ha, http://www.linux-ha.org, 2010.

[12] SGI, http://oss.sgi.com/projects/failsafe/, 2010.

[13] Shye, et. al. *PLR: A Software Approach to Transient Fault Tolerance for Multi-Core Architectures*. IEEE Transactions on Dependable and Secure Computing (TDSC). April-June 2009 (vol. 6 no. 2) pp. 135-148.

[14] C.-K. Luk, et al. *Pin: Building customized program anal- ysis tools with dynamic instrumentation*. In *PLDI* 2005.

[15] L. Burcin, *Rad750 Experience: The Challenge of SEE Hardening a High Performance Commercial Processor.* In Microelectronics Reliability & Qualification Workshop (MRQW), 2002.

BIOGRAPHY

John Paul Walters *is a Computer Scientist at the University of Southern California's Information Sciences Institute, located in Arlington, VA. He received his PhD in computer science from Wayne State University in 2007, and his BA in computer science from Albion College in 2002. His research interests include fault tolerance, high performance computing, cloud computing, parallel processing, and many-core architectures.*

Robert Kost *is employed at the University of Southern California/Information Sciences Institute (USC/ISI) in Arlington, Virginia. He is also a PhD candidate at the University of Colorado at Boulder. He received his M.S. degree in eletectrial engineering at the University of Colorado at Boulder in 2006, and a B.S. degree from Old Dominion University in 2003. He was also an Army officer prior to his employment with USC/ISI. His research interests include both the system and software domain: fault tolerance, optimization, parallel programming, cloud computing, high performance computing, embedded system design, and computer architecture. He is a member of the IEEE and IEEE Computer Society.*

Karandeep Singh *is a Research Programmer at the University of Southern California's Information Sciences Institute. His research interests include high performance computing, embedded systems and cloud computing. He has a MS in Electrical and Computer Engineering from the University of Maryland, College Park.*

Jinwoo Suh *is a Computer Scientist at the University of Southern California's Information Sciences Institute since 1999. He received a Ph. D. degree in electrical engineering from the University of Southern California in 1999, an MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea in 1990, and a BS degree in electronic engineering from Dongguk University, Seoul, Korea in 1988. He was an engineer at the research center of the DAEWOO Electronics Co., Ltd, Seoul, Korea from 1990 to 1994. His research interests include cloud computing, fault tolerance, parallel processing, algorithm design, performance optimization, real-time processing, and computer architecture. He is a senior member of the IEEE and IEEE Computer Society.*

Stephen Crago *is an Assistant Director at the University of Southern California's Information Sciences Institute, where he currently leads a group focused on multi-core software. He has led development of software and hardware for data-intensive, power-aware, polymorphous, cognitive information, and space-based processing systems. He has been at USC/ISI's Arlington facility since 1997. He has a BSCEE and MSEE from Purdue University and a Ph.D. in EE from the University of Southern California.*