Micro-checkpointing in Fault Tolerant Runtimes

Pavlos Katsogridakis
Institute of Computer Science
Foundation for Research and Technology – Hellas
katsogr@ics.forth.gr

Polyvios Pratikakis
Institute of Computer Science
Foundation for Research and Technology – Hellas
polyvios@ics.forth.gr

ABSTRACT

Multicore processors are increasingly used in safety-critical applications. On one hand, their increasing chip density causes these processors to be more susceptible to transient faults; on the other hand the existence of many cores offers a straightforward compartmentalization against permanent hardware faults. To tackle the first issue and take advantage of the second, we present FT-BDDT, a fault-tolerant task-parallel runtime system. FT-BDDT extends the BDDT runtime system that implements the OMP-Ss dataflow programming model for spawning and scheduling parallel tasks, in which, similarly to OpenMP 4.0, a dynamic dependence analysis detects conflicting tasks and automatically synchronizes them to avoid data races and non-determinism.

FT-BDDT recovers from both transient and permanent faults. Transient faults during task execution result in simply re-running the task. To handle transient faults in the runtime system, FT-BDDT uses fine-grain micro-checkpointing of the runtime state, so that a recovery is always possible at the level of rerunning a basic block of code on error. Permanent faults are treated in a similar fashion, by having the master core "steal" the task checkpoint or the runtime micro-checkpoint and reschedule the task or recover the runtime state, respectively.

We evaluate FT-BDDT on several benchmarks under various error conditions, while guiding errors to attain maximum coverage of the runtime code. We find a 9.5% average runtime overhead for checkpointing, a constant small space overhead, and a negligible recovery time per error.

Keywords

Task Parallelism, Fault tolerance, Parallel Scheduling, Language Runtime System, Reliability

1. INTRODUCTION

Multicore processors are increasingly used in safety-critical applications [17]. This causes two sets of problems for these systems: First, the high chip density that allows many processing cores to fit in one device, causes these processors to be more susceptible to transient faults or *soft errors* [5, 28], caused by high energy particle strikes, overheating, device aging, etc. [14, 3], or even permanent faults that cause a processor to fail. On the other hand, the existence of many cores offers a straightforward compartmentalization against permanent hardware faults, as the failure of a processing core does not affect the remaining cores in the processor [1].

Second, multicore processor programming is difficult; it requires the programmer to reason about all possible interactions between parallel threads, it introduces non-determinism and implicit communication among processing cores through memory. To address these inherent difficulties with low-level thread parallel programming, task based multicore runtimes such as Cilk [4], OpenMP [6], and Sequoia [24] or task libraries such as TBB [19] and TPL [12], provide a better abstraction to the programmer over threads. Second generation task-parallel programming models such as OpenMP 4.0 [2], OMP-Ss [8], Myrmics [13], and BDDT [26, 27], combine dynamic dataflow, tasks, and automatic synchronization and offer a high-level abstraction that facilitates parallel programming. Moreover, the task abstraction provides useful properties such as well-defined memory footprints for parallel tasks, relaxed yet well-defined points of coherence and communication, ability to optimize locality, discover dependencies, and sophisticated scheduling optimizations.

Due to these properties, tasks provide a suitable abstraction for the management of faults in multicore processors; task boundaries are well-defined points of checkpointing and recovery, allowing the runtime system to checkpoint task data before execution so that the task can be restored and rescheduled on error [7, 29]. Existing runtimes take advantage of this to tolerate faults in application code. However, tolerance of faults in the runtime system or scheduler itself remains an issue, as these runtimes use shared-memory data structures for scheduling and communication and cannot be easily split into well-defined independent computations.

To address the issue of soft errors or permanent faults that occur during the execution of runtime code, this paper presents FT-BDDT, a fault-tolerant task-parallel runtime system. FT-BDDT implements the BDDT dataflow programming model for spawning and scheduling parallel tasks, in which, similarly to OpenMP 4.0, a dynamic dependence analysis detects conflicting tasks and automatically synchronizes them to avoid data races and non-determinism.

FT-BDDT recovers from both transient and permanent faults that may occur during the execution of program tasks,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

but also during the execution of the runtime code. Transient faults during task execution result in simply re-running the task. To handle transient faults in the runtime system, FT-BDDT uses fine-grain micro-checkpointing of the runtime state, so that a recovery is always possible at the level of rerunning a basic block of code on error. Permanent faults are treated in a similar fashion, by having the master core “steal” the task checkpoint or the runtime micro-checkpoint and reschedule the task or recover the runtime state, respectively.

Overall, the contributions of this work are:

- We propose a micro-checkpointing approach for runtime system algorithms and shared data structures, as these parts of the runtime often operate on shared data structures for scheduling and communication among worker threads. Our approach checkpoints the state of basic blocks of code inside the runtime, along with recovery information for locks and other synchronization primitives, so that any part of the runtime can be recovered seamlessly without interrupting global computation.

- We manually transform the default BDDT rutime code, using the aforementioned approach to extend the BDDT runtime with checkpointing, in the FT-BDDT runtime. FT-BDDT assumes that transient or permanent faults can occur in any worker core at any time, even while the scheduler code is running in-between tasks.

- We evaluate FT-BDDT on PARSEC benchmarks under various error conditions, while guiding errors to attain maximum coverage of the runtime code. We find a 9.5% average runtime overhead for checkpointing, a constant small space overhead, and a negligible recovery time per error.

The remainder of this paper is structured as follows. Section 2 motivates and introduces the BDDT programming model and task-parallel execution runtime. Section 3 presents the fault model; the possible faults and the fault detection assumptions made in designing FT-BDDT. Section 4 presents the design of FT-BDDT micro-checkpoints and recovery mechanisms. Section 5 presents the benchmarks and test configurations used to evaluate FT-BDDT and the results of these experiments. Section 6 discusses related work and Section 7 concludes.

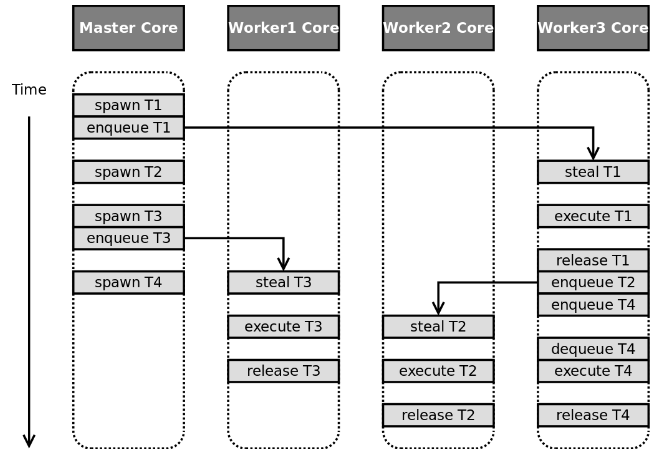
2. BACKGROUND

BDDT is a task-parallel runtime that dynamically discovers and solves dependencies among parallel tasks. When a task is created the programmer states its memory footprint, i.e., the memory locations that the task code will read and write during task execution. The runtime ensures that the task will be ready to run when all its input dependencies are resolved.

Consider the example program shown in Figure 1a, which demonstrates the BDDT [26, 27] task-parallel programming model with automatic synchronization. To create data dependencies among tasks, line 2 declares variables A through F. Lines 4–5 spawn the first parallel task to execute function `T1`. Tasks are spawned by an OpenMP-like syntax, using `#pragma` annotations that define the memory footprint of the spawned task. In this case, the spawned task function

```
1   void main() {
2     int A, B, C, D, E, F;
3
4   #pragma task in(A), in(B), out(C)
5     T1(&A, &B, &C);
6
7   #pragma task in(C), in(A), out(D)
8     T2(&C, &A, &D);
9
10  #pragma task in(A), in(B), out(E)
11    T3(&A, &B, &E);
12
13  #pragma task in(C), in(B), out(F)
14    T4(&C, &B, &F);
15  }
```

(a) A Task-Parallel Program



(b) Execution example

Figure 1: Runtime execution example

`T1` takes three arguments by reference; arguments `A` and `B` are marked `in`, meaning they will only be read by the task, whereas argument `C` is marked `out`, meaning that it will be written by the task. The second task spawned (Lines 7–8) calls function `T2` to read from `C` and `A`, and write to `D`. Although both `T1` and `T2` access `A`, there is no dependency on that access, as `A` is a read-effect `in` for both tasks. On the other hand, variable `C` is written by the first task and read by the second task, creating a read-after-write dependency. The runtime dependence analysis uses the task footprints declared in the `#pragma` annotations to dynamically detect such conflicts. In this example, the runtime will schedule the second task to be executed only after the first task has completed because variable `C` is in the footprint of both, written by the first and read by the second. In the mean time, the program can continue to spawn the third task (Lines 10–11), calling function `T3` that reads variables `A` and `B` and writes `E`. Note that `T3` can start immediately, because it has no conflicting dependencies with the first two tasks. Finally, the program spawns `T4` that reads `C` and `B` and writes `F`. Since `T4` reads C which is written by `T1`, it also has to wait for the first task to finish.

Figure 1b shows a possible execution schedule of the program in Figure 1a, as run by 1 master and 3 worker cores. Figure columns demonstrate the execution trace of each

thread, where time progresses downwards. Each thread has a dequeue of tasks, used to schedule tasks across all workers. The work distribution is done by work stealing. Initially, the Master core spawns task `T1` and immediately places it on its dequeue, as there are no dependencies that may cause task `T1` to block. Next, Worker core 3 steals the task from the masters dequeue and starts executing it, while the Master core spawns task `T2`. Note that since task `T2` is not ready to run as it requires ownership of variable `C`, it is not enqueued for execution but rather waits for task `T1` to finish. Next, the Master core continues to spawn task `T3` which is ready to run and is immediately enqueued for execution, while Worker core 3 executes task `T1`. Next, Worker core 1 steals task `T3` from the queue of the Master core, while the Master core continues to spawn task `T4`. As with task `T2`, task `T4` is not ready to be executed, as it requires access to variable `C` and must wait for task `T1`. When task `T1` is finished, Worker core 3 releases the task's dependencies, causing tasks `T2` and `T4` to be enqueued for execution in the queue of Worker core 3. Worker core 2 can then steal task `T2` from that queue and proceed with executing it, while Worker core 3 continues with executing task `T4` from its own queue.

The task abstraction summarized in this example offers clearly-defined boundaries for checkpointing the application state. Specifically, the programming model guarantees that each task will not communicate with other tasks during its execution and that it will only read and write memory declared in the task footprint. These guarantees greatly simplify checkpointing the application state, since it suffices to checkpoint only the task footprint before running a task and rerunning failed tasks on the same input. Skarlatos et al. [7] have shown task-only checkpointing to scale linearly and perform with minimal overhead, assuming faults only occur during task execution and not while the runtime system is scheduling the next task. In contrast, Section 3 presents our fault model, in which we assume that transient or permanent faults may occur at any point in the execution —even outside application code— while the runtime system performs scheduling, synchronization, memory management, etc.

3. FAULT MODEL

We make the following assumptions regarding possible faults:

- We assume that the memory hierarchy is protected from faults using Error Correcting Code (ECC). Although hardware faults may occur anywhere in the system, in this paper we focus on faults occurring within computing components of the architecture. Specifically, we assume that on fault, only the CPU state of the faulty core (registers, program counter, etc.) is lost.

- We assume two kinds of faults: *Transient faults* cause a computation to give the wrong result once, although a subsequent re-execution of the same operation can compute the correct result. *Permanent faults* cause a processing core to fail and never recover, rendering it unusable for the rest of the execution.

- We assume that both transient and permanent faults are detected early before an erroneous result is written into memory, so that there is no corruption of unrelated memory locations [21]. This is a realistic assumption, as detection can be orthogonally implemented using several haredware or software techniques proposed in the literature. For instance, per-core checks [18] can detect errors in the processor pipeline, DMR or TMR [16] for processor parts (such as the ALU) can detect faulty core components, hardware monitors [22] can detect permanent faults by analyzing the trace of each component in parallel, etc. Upon detection of transient faults, we assume that the processing core interrupts its execution and jumps to the recovery code. On permanent faults, we assume that the processing core becomes stuck and does nothing else. We believe these assumptions are reasonable and agree with existing literature on fault detection.

- We assume that no faults can occur at one of the available cores, which operates as the "master" core for the runtime and spawns all tasks executed by the "worker" cores. This assumption is realistic and can be accomplished using e.g., replication (TMR) to detect and correct faults in the master core, without incurring the same expensive resource overhead to protect the remaining cores.

- We assume that atomic operations remain atomic in the presence of faults.

4. DESIGN

The main operations performed by the original BDDT runtime at each worker core are *Acquiring* a task to execute, *Executing* a task, and *Releasing* a finished task's dependencies. To Acquire a task, the worker core tries to dequeue a task descriptor from the local dequeue; if there are no local pending tasks, the worker tries to steal a task from the dequeue of a different worker or the master. To Execute a task, the worker core invokes the closure of the task as stored in the task descriptor. After the execution of a task, the worker releases all its dependencies. Release is a complex operation because for each output argument of the finished task it must traverse a list of tasks waiting for that memory and remove that dependency. If any of the dependent tasks have no remaining dependencies, then they are free to run and the worker enqueues their task descriptors into its dequeue for later execution or to be stolen by other worker cores.

For instance, in the example of Figure 1, Worker core 3 releases the dependencies of task `T1` once it is complete, which makes tasks `T2` and `T4` ready to execute. They are then enqueued into the dequeue of Worker core 3. Task `T2` is stolen and executed by Worker core 2 and task `T4` is dequeued and executed locally by Worker core 3. Figure 2 shows the code for function **`g_dequeue`**, called by Worker core 2 to steal task `T2` and by Worker core 3 to remove task `T4`, both from the local dequeue of Worker core 3. Note that the dequeue is implemented using an array and circular modulo indexing. To synchronize the two potentially racy calls to **`g_dequeue`** by Worker core 2 and Worker core 3, the function uses a lock per queue, acquired in line 4. The function then computes the list size (line 5) and if there is no task to take, it releases the list lock and returns NULL (lines 11–13). Otherwise, it takes a task from the list (line 8), updates the top index (line 9), releases the lock and returns (lines 11-13).

According to the fault model described in Section 3, any Worker core may fault at any of these points: (i) during

its top-level loop that acquires, runs and releases tasks; (ii) during the acquiring of a task from the local dequeue or the stealing of a task from a remote dequeue; (iii) during the execution of a task; and (iv) during the releasing of a task's dependencies. The third case has been addressed by Skarlatos et al. [7] using the declaration of the task memory footprint: before running a task, the runtime checkpoints all writable memory declared in its footprint; if an error occurs during the execution of the task, the runtime restores the contents of memory and reruns the task locally, for transient errors, or allows the task to be stolen for permanent errors.

Unfortunately, the runtime system code cannot be treated the same way, because it uses shared data structures among all worker cores to schedule tasks and to perform the dependence analysis. Thus, for the other three possible points of failure, namely (i), (ii), and (iv), we propose a micro-checkpointing technique that allows the runtime to seamlessly recover from both transient and permanent faults.

4.1 Micro-checkpointing of the Runtime Code

To enable the runtime system to checkpoint every point during execution, we perform the following changes on the runtime system code.

1. Allocate a `state` structure per thread, to hold all the information necessary to reconstruct the execution from any point of failure.
2. Divide every function in the runtime code into phases corresponding to basic blocks, so that every phase contains at most a single write to thread-shared memory, or a single function call.
3. Insert code in every function so that on entering every phase, the runtime writes the current phase identifier in the thread state.
4. Insert code in the previous phase immediately before every write to shared memory, to store the previous value into the thread state, e.g., the previous value of a counter being incremented.
5. Insert code before entering a new phase, to store the contents of any stack-allocated local variables that are used from one phase to the next.

We also change the code so that it is possible to restore the execution to the last micro-checkpoint upon failure. To do that, we perform the following changes:

1. Extend every function with an additional argument denoting the phase at which execution should resume after the fault.
2. Rewrite the function code so that control flow jumps to the correct phase of execution; this is often straightforward using a `switch` statement to jump to the correct point, similarly to the Cilk [4] source-to-source transformation that enables executing the continuation of a point in the function code.

For example, consider the code in Figure 2. Function `g_dequeue` can be divided into four phases, or basic blocks: (i) lines 4–7 acquire the lock and the basic block ends with control-flow via the `if` statement; (ii) line 8 reads from shared memory via the `Q` pointer and writes to a local variable; (iii) line 9 writes to shared memory by incrementing

```
1   task_t* g_dequeue(g_Queue Q) {
2     task_t*  steal_task  = NULL;
3
4     lock(Q->lock);
5     if ((Q->bottom - Q->top) <= 0) {
6       goto exit;
7     }
8     steal_task  = Q->qEntry[Q->top % MAX_ENTRIES];
9     Q->top++;
10
11  exit:
12    unlock(Q->lock);
13    return  steal_task;
14  }
```

Figure 2: Original dequeue function

```
1   task_t* g_dequeue_chkpt(g_Queue Q, phase_t phase) {
2     task_t*  steal_task =NULL;
3
4     switch(phase) {
5       case(PHASE_ONE):
6         set_shared_deqstate (PHASE_ONE);
7         lock(Q->lock);
8         if ((Q->bottom - Q->top) <= 0) {
9           goto exit;
10        }
11      case(PHASE_TWO):
12        set_shared_deqstate (PHASE_TWO);
13        steal_task  = Q->qEntry[top % MAX_ENTRIES];
14        set_sqentry ( steal_task );
15        set_backup_qtop(Q->top);
16      case(PHASE_THREE):
17        set_shared_deqstate (PHASE_THREE);
18        Q->top++;
19      case(PHASE_FOUR):
20      exit:
21        set_shared_deqstate (PHASE_FOUR);
22        unlock(Q->lock);
23        return  steal_task;
24    }
25  }
```

Figure 3: Checkpointing dequeue function

a counter; and (iv) lines 11–13 release the lock and return. Note that even though only the local variable `steal_task` is written in phase (ii), phase (iii) needs to be a separate phase. This occurs because `steal_task` is also read in phase (iv) and therefore needs to be checkpointed inside phase (ii). At the same time, the write to `Q->top` writes to shared memory and therefore needs to checkpoint its previous value at the end of the previous phase. Thus, were phases (ii) and (iii) to be merged, the two checkpointings would happen with a different order than in the original program.

Overall, we transform the original `g_dequeue` function to its checkpointing equivalent, function `g_dequeue_chkpt`, shown in Figure 3. Lines 6–10 correspond to phase (i): at the start of each phase (line 6) the phase identifier is checkpointed into the state of the running thread. In this case, the checkpointing uses function `set_shared_deqstate` meaning that if a fault occurs before the next micro-checkpoint, the program counter should be restored at the first phase of function `g_dequeue`. Lines 12–15 correspond to phase (ii): in addition to setting the phase identifier in line 12, line 14 checkpoints the local variable `steal_task` because its value escapes the current phase and may be read in phase (iv). Moreover, line 15 checkpoints the value of `Q->top` because

```
1  void restore_dequeue(int cpuid) {
2    deq_state_t next_phase;
3    switch( scheduler_state [cpuid].dequeue_state) {
4      case(PHASE_ONE):
5        if (lock_isowned(Q−>lock, cpuid))
6          next_phase = PHASE_FOUR;
7        else return;
8        break;
9      case(PHASE_TWO):
10       next_phase = PHASE_TWO;
11       break;
12     case(PHASE_THREE):
13       if (Q−>top == scheduler_state[cpuid].qtop) {
14         next_phase = PHASE_THREE;
15       } else {
16         assert (Q−>top == scheduler_state[cpuid].qtop+1);
17         next_phase = PHASE_FOUR;
18       }
19       break;
20     case(PHASE_FOUR):
21       if (lock_isowned(Q−>lock, cpuid))
22         next_phase = PHASE_FOUR;
23       else return;
24       break;
25   }
26   g_dequeue_chkpt(Q, next_phase);
27 }
```

Figure 4: Recovery code for the dequeue function

it will be written in the next phase. Lines 17–18 correspond to phase (iii) that increases `Q->top`. Finally, Lines 20–23 correspond to phase (iv).

The remaining changes shown in Figure 3, namely the addition of a second function argument (line 1) and its use to drive control flow to the start of the appropriate phase (lines 4–5, 11, 16, 19) are used in the event of a fault, so that the same thread (for transient faults) or a different thread (for permanent faults) can resume execution at the last micro-checkpoint.

4.2 Recovery Code

In addition to the above changes to all runtime code, we extend the runtime system with recovery code. For each function in the runtime system we add a new recovery function. Recovery functions are called on error by the same core (for transient faults) or a different core (for permanent faults) and drive execution to the appropriate point in the runtime code depending on the micro-checkpoint available. Recovery functions are pure, i.e., they have no visible side-effects on shared memory or the state of the runtime system on other cores. This facilitates the recovery process: in the case of a second fault during the recovery process, recovery can simply restart.

As described in Section 3, we assume that when a transient error is detected, the faulty core resets its execution and jumps to the main recovery function. Each recovery function f_r reads the checkpoint state of the core where the fault occurred and calls the corresponding runtime function f to continue execution from the appropriate phase. If the fault occurred in another function g called by f, then the recovery function f_r will call the recovery function g_r, and continue the execution of f only after g has recovered properly. This way, the stack at the time of the fault is restored backwards, effectively executing the continuation of the fault.

For example, Figure 4 shows the corresponding recovery function `restore_dequeue` for the function `g_dequeue_chkpt` shown in Figure 3. The only argument to the recovery function is the identifier of the core where the fault occurred, as all other information can be found in the checkpoint state of that core. Local variable `next_phase` (line 2) is used to call function `g_dequeue_chkpt` and drive execution to the appropriate phase. Initially, the recovery function looks up the last phase that `g_dequeue_chkpt` checkpointed before the fault (line 3). Recall that the first phase locks the dequeue and checks its size, as shown in Figure 3 (lines 6–10). Therefore, any error during that phase need only restore the state of the lock, if necessary, as it is always safe for a steal or local-dequeue operation to fail to dequeue a task. So, the recovery function checks whether the lock is acquired by the core where the fault occurred (line 5) and calls the runtime function `g_dequeue_chkpt` to continue at the last phase and release the lock (line 26), or simply ends recovery if the lock was not acquired before the fault (line 7). Lines 9–11 of the recovery function simply check whether the fault occurred anywhere inside the second phase (lines 12–15 of Figure 3), as it is always safe to execute the second phase again. The third phase of function `g_dequeue_chkpt` increments a shared counter (line 18 of Figure 3), thus the recovery code for the third phase (lines 12–19) checks whether the write to shared memory occurred before the fault and so the phase does not need to be repeated, otherwise it sets the phase to be executed. Note that no other thread can write the counter since the core that faulted still holds the lock and so it is always safe to compare the current value of the counter with the checkpointed value to decide whether it was incremented or not (line 13). Finally, if the fault occurred during the fourth phase (lines 22–23 of Figure 3), the recovery function checks the ownership and state of the dequeue lock, as the fourth phase needs to be executed only if the lock was not released before the fault and is still owned by the core that faulted.

4.3 Stack Recovery

According to the fault model described in Section 3, the stack pointer of the faulty core, along with all other registers, may be corrupted by a transient fault or inaccessible due to a permanent fault. The recovery process must, then, recreate the contents of the stack at the checkpointed state in order to run the continuation of the code at the point where the fault occurred. To do that, we take advantage of the fact that the runtime code is not recursive. Figure 5 shows the main parts of the call tree of the runtime system. Since there is a recovery function for every function in the runtime code, the call tree of the recovery functions is similar; in addition, each recovery function also calls the corresponding runtime function.

To demonstrate the recovery of the execution stack after a fault, assume the stack indicated by the bold arrows in Figure 5. Namely, the main loop of a Worker core calls the Release function that removes the dependencies of a finished task. In turn, function Release calls function PushTasks to find any tasks without remaining dependencies, which calls function Enqueue to insert each such task to the dequeue for execution. Assume that a transient fault occurs during the execution of the Enqueue function. According to the fault model, the Worker core resets and starts executing the recovery code. Recovery always starts with the recovery function corresponding to the top-level of the worker runtime,

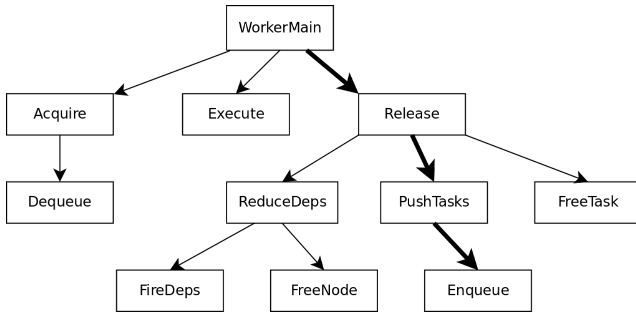


Figure 5: Call tree of the runtime system

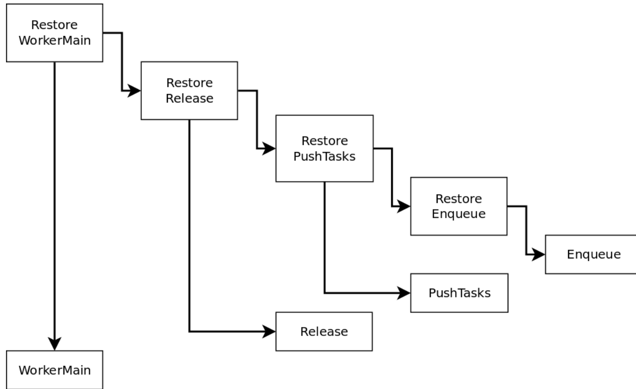


Figure 6: Restore call trace

namely function WorkerMain. Figure 6 depicts the call trace of the recovery, starting with the call to the WorkerMain recovery function. The figure shows the stack increasing to the right as time advances downwards. The WorkerMain recovery function examines the last checkpointed state of the core for the WorkerMain function and calls the recovery function for Release, shown as "Restore Release" in the Figure. Function Restore Release inspects the last checkpointed state for function Release, infers that Release had invoked function PushTasks at the point of fault and itself calls the recovery function for PushTasks, "Restore Pushtasks", which similarly calls "Restore Enqueue". After the recovery of the function Enqueue as explained in the example of the previous section, Enqueue returns to "Restore Enqueue", which returns to "Restore PushTasks". That recovery function can then call function PushTasks at the appropriate phase, to finish its execution now that Enqueue has been recovered and executed. Similarly, PushTasks returns to "Restore PushTasks" which returns to "Restore Release". Function Release can then be called to continue its execution at the phase after it calls PushTasks. Once the recovery function "Restore Release" returns to the top-level recovery function "Restore WorkerMain", that can call (or longjmp) to function WorkerMain, to continue with the worker's top-level loop as all computations interrupted by the fault have finished.

4.4 Recoverable Locks

In order to implement the above recovery mechanism, FT-BDDT requires the recovery code to be able to discern whether a lock was successfully acquired by the core that faulted before the fault. In general, atomic operations pose

```
1  void spinlock_acquire (int32_t* lock) {
2    int cpuid = runtime_get_worker_id();
3    int i, delay = MIN_DELAY;
4    while(!CAS(lock, UNLOCKED, ((cpuid)<<16) | 0x1)) {
5      do{
6              delay <<= 1;
7              for(i = 0; i < delay; i++){ pause(); }
8      } while( spinlock_islocked(*lock));
9    }
10 }
```

(a) Lock acquiring

```
1  int spinlock_isowned(int32_t* lock, uint32_t cpuid) {
2    uint32_t var = (LOCKED | (cpuid << 16));
3    return (lock == var);
4  }
```

(b) Lock ownership

Figure 7: Lock implementation

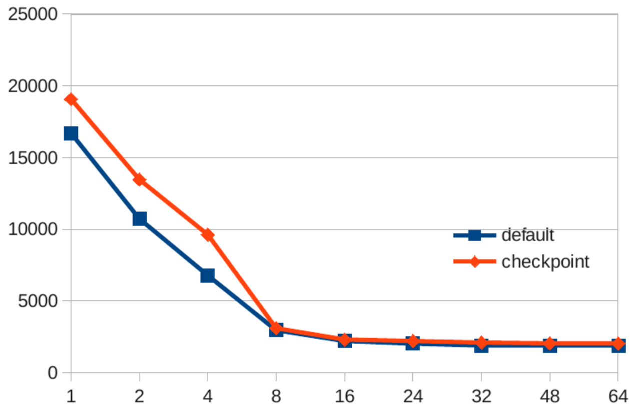
a difficulty in checkpointing as it is not always possible to checkpoint the result of an atomic operation or differentiate the effects of the faulty core with the effects of concurrent code. For example, synchronization via atomic increment of shared counters is not possible to checkpoint and recover, as there is no way for the recovery code to know if an atomic increment instruction was executed before the fault by comparing with the previous known value of the counter.

To address this issue, we have implemented a custom, recoverable lock primitive and used it to replace other ways of synchronization used in the original BDDT code. By default, BDDT uses atomic primitives when possible and TAS locks to synchronize concurrent accesses to memory not accessible by atomic primitives. We replaced all these synchronization operations with recoverable locks in FT-BDDT, so that the recovery code can safely query the status and ownership of all locks to recreate lock state information after a fault. Figure 7 presents the implementation of recoverable locks. We implement each lock using 32-bit word; the least significant bit represents the state of the lock, while the 16 most significant bits store the core identifier of the owner core. Recoverable locks are acquired and released using the compare-and-swap (CAS) atomic primitive.

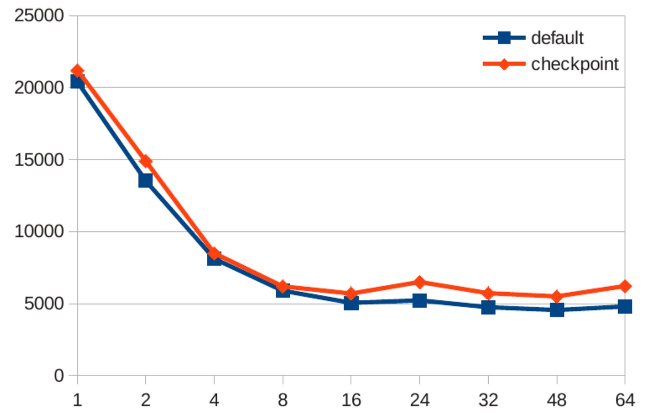
Figure 7a shows how the lock is acquired. Line 2 looks up the core identifier using the runtime API. We assume that the runtime system is able to initialize itself correctly, so that each worker core has a unique identifier. Line 4 attempts to acquire the lock using a CAS atomic operation to write both the state of the lock and the identifier of the owner core, if the lock is released. Lines 5–8 implement a simple exponential back-off to reduce contention among competing cores. Note that a fault at any point in the process does not cause recovery to suffer, as the CAS operation is atomic; it is easy to decide if the recovering core succeeded in acquiring the lock during recovery, as shown in Figure 7b. Here, we assume that the load instruction of a 32-bit word that runs before the comparison at line 3 is atomic.

5. EXPERIMENTS

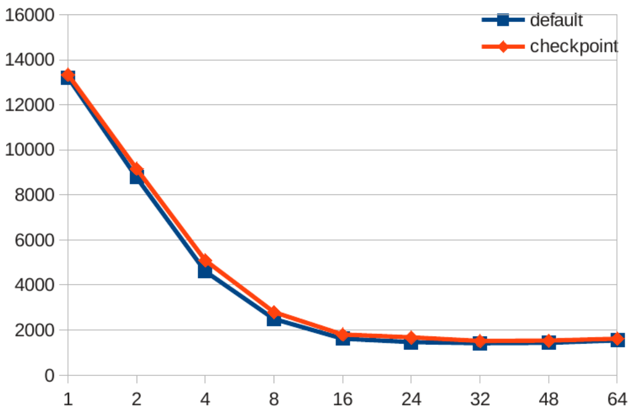
We evaluate the cost of micro-checkpointing using six representative benchmarks: HPL, Multisort, FFT, Jacobi, Black-



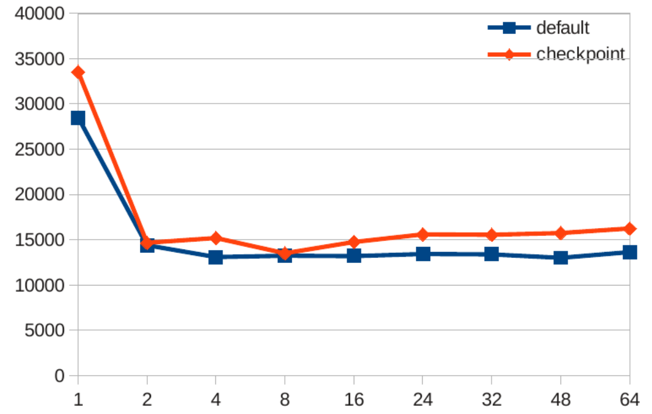
(a) HPL



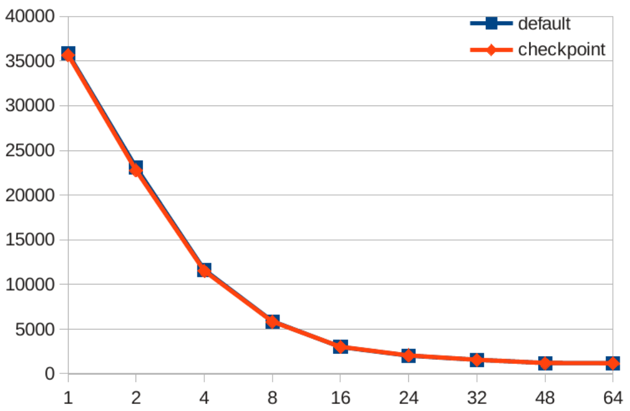
(b) Multisort



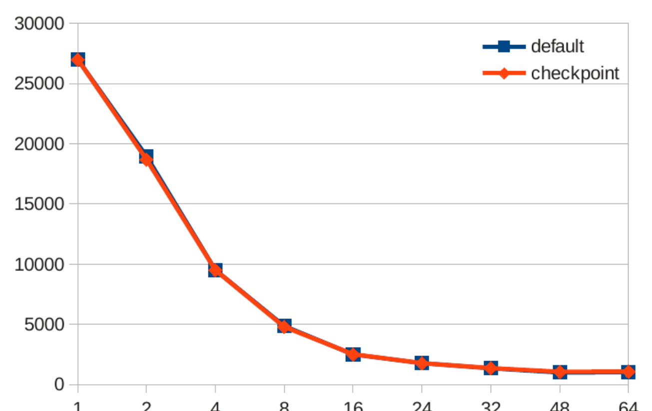
(c) FFT



(d) Jacobi



(e) Black-Scholes



(f) Cholesky

Figure 8: Comparison between FT-BDDT and default BDDT

Scholes and Cholesky. We ran all experiments on a NUMA-cc 4-CPU machine with 64 AMD Opteron(TM) Processor 6272 cores and 256GB total RAM, running Linux 2.6.32. All benchmarks were compiled using GCC 4.6.3 with optimization level -O3. All reported running times are the average of 3 executions. All plots omit variance or deviation for clarity, as the execution times showed minimal variance of less than 10 milliseconds for all benchmarks. The y axis shows the time in milliseconds, the x axis the number of cores.

5.1 Overhead of Micro-Checkpointing

To measure the cost of micro-checkpointing in the runtime system we have compared FT-BDDT where both task checkpointing and runtime checkpointing are enabled, with

the original non-fault-tolerant BDDT runtime. Figure 8 presents the results for all benchmarks.

HPL.

HPL solves a random dense linear system in double precision arithmetic. It is part of the High-Performance Linpack Benchmark. Figure 8a shows the execution times in milliseconds of BDDT and FT-BDDT for a grid size of 8MB and block size of 64 doubles. Checkpointing in FT-BDDT incurs 13.7% overhead for 1 thread, which drops to 11% for 32 threads and 8% for 64 threads.

Multisort.

The Multisort kernel is a parallel version of the Mergesort algorithm, originating from the Cilk distribution. Multisort is a divide and conquer algorithm with two phases. The first phase divides the data into chunks and sorts each, and the second phase merges them. Figure 8b shows the resulting execution times for an array of 256M elements, using a threshold of 128K elements for stopping recursive subdivision. The FT-BDDT checkpointing in Multisort results into the largest overhead over BDDT, namely 1% for 1 core, 20% for 32 and 30% for 64 cores. This overhead is caused by two factors; (i) the large size of the task footprints increases the copying overhead of task checkpoints and (ii) the large number of task dependencies causes the runtime code to execute more often and perform more computations, resulting in a larger number of micro-checkpoints.

FFT.

The FFT benchmark is a task-parallel implementation of a 2-dimensional Fast Fourier Transform algorithm. This FFT implementation is part of the SMP-Ss distribution [25], and consists of five parallel loops that alternate in transposing the input array and performing 1-dimensional FFT on each row. Each task created in the FFT calculation loop operates on an entire row of the array, while transposition phases break the array into tiles and create a task to transpose a group of tiles. Figure 8c shows the execution times for FT-BDDT and BDDT, using a 2-D array of 64M elements and 32×32 transpose tile size. The overhead of checkpointing the runtime state is 1% for 1 thread and increases to 6% for 32 and 64 threads.

Jacobi.

The Jacobi kernel is a 5 point stencil computation used to solve linear equations. We use the Jacobi kernel implementation from the SMP-Ss distribution which uses row-major array layout. Each task in Jacobi works on a tile of the array. Figure 8d shows the execution times of FT-BDDT and BDDT for an array of 8192×8192 elements using a task tile size of 128×128. The kernel is communication bound and memory intensive so in Jacobi the state-keeping overhead is large: FT-BDDT incurs 17% overhead for 1 thread, 14% for 32 threads and 16% for 64 threads. This is also caused by the large number of task dependencies, as in the case of Multisort. BDDT and FT-BDDTalso do not scale, as the maximum speedup is 2×.

Black-Scholes.

The Black-Scholes application is a parallel implementation of a mathematical model for price variations in financial markets with derivative investment instruments. It decomposes and processes the data in rows. Black-Scholes was tested for 30000 options, split into chunks of 128. The overhead of FT-BDDT over BDDT is negligible for all numbers of cores, as shown in Figure 8e.

Cholesky.

The Cholesky factorization kernel is used for LU decomposition in symmetric positive definite arrays. Figure 8f shows the performance for a 4096×4096 double precision matrix, with 64×64 tiles. The difference between the execution time of BDDT and FT-BDDT is in the noise. Both versions achieve a top speedup of 25× on 64 cores.

5.2 Correctness Testing via Fault Injection

We tested the correctness and reliability of FT-BDDT in the presence of transient and permanent faults using both systematic and randomized fault-injection.

To test FT-BDDT in the presence of transient faults, we inserted fault emulation code before and after every load and store of a shared memory location. The emulation code causes a transient fault according to a uniform probability model and immediately jumps the “faulty” core to the recovery code. We tested FT-BDDT for all benchmarks and various probabilities of error of up to 5%, resulting in 1 up to 65536 faults for each worker. We found that recovery from transient faults in runtime code did not affect the total execution time in a statistically significant way, i.e., the effect of fault recovery is always in the noise of the execution time variation, which is always less than 10 milliseconds of total execution time. The zero effect on total execution time occurs because the computation that is redone is negligible and the caches are already warm. In addition to random transient faults, we performed a systematic test of all emulated transient error points failing exactly once.

To emulate permanent faults we use `pthread_kill` to send signals from the Master thread to random Worker threads. To emulate permanent fault detection, the Master thread periodically checks the state of all Worker threads and calls recovery code for any Workers found not running. We tested all benchmarks running on 64 cores with one permanent fault during the first second of execution and found negligible variation in total execution time. We believe that fault emulation using kill signals is a reliable method of emulating permanent faults, as the signals can be delivered between the execution of any two instructions of the receiving thread.

6. RELATED WORK

Task Parallel Programming.

Task parallel languages and runtimes such as Cilk [4], OpenMP [6], and Sequoia [24] or task libraries such as Intel TBB [19] and Microsoft TPL [12], provide a better abstraction to the programmer over threads. Second generation task-parallel programming models such as OpenMP 4.0 [2], OMP-Ss [8], Myrmics [13], and BDDT [26, 27], combine dynamic dataflow, tasks, and automatic synchronization and offer a high-level abstraction that facilitates parallel programming.

BDDT [26, 27] is a task parallel runtime that implements the OpenMP-Ss task parallel language. BDDT uses a custom memory allocator to view memory in terms of fixed-

sized memory blocks and detects task dependencies in terms of the memory blocks included in task footprints. This enables BDDT to support arbitrary memory references and pointer arithmetic, tiling and re-tiling of arrays into arbitrary tasks without re-allocation, and changing access patterns during the many phases of an application.

Fault Detection and Recovery.

There are several methods of fault detection that fit the FT-BDDTfault model. RAFT [28] is a fault detection and recovery system for single-threaded programs. RAFT spawns a second instance of the running process and wraps all system calls in order to check that argument values match. To avoid synchronization RAFT speculates the return values of system calls, avoids synchronization barriers and only verifies values that escape the user space. The reported overhead for RAFT is 2.83% on average. Reinhardt et al. [20] propose an architectural method for fault detection by using hardware multithreading to run the same instance of the program simultaneously using lockstepping and compare the store instructions of each execution. The authors measure the overhead of the multithreaded version versus the single threaded using simulation and report it to be less than 2%. An alternative approach is to verify each stage in the hardware pipeline, correcting faults that have propagated in subsequent stages [18]. Similar approaches can be used to detect transient faults and implement the fault model we assume in FT-BDDT, forcing the processor to a reliable state in the recovery code as soon as a fault is detected in the core.

SWIFT [21] is a compiler-based way of detecting and recovering from transient faults in single-theaded applications. The compiler inserts extra assembly instructions that duplicate the ALU operations and one compare instruction to check whether the two results match, or create 3 instances and vote to get the correct result. Shoestring [9] uses a compiler analysis to detect the error prone code and inserts instruction duplication in a more efficient way than SWIFT, reducing unecessary duplication. Although these systems target single processor applications, the same fault-detection methods can be used to detect faults for FT-BDDT. Our generic recovery solution can then take advantage of the fault detection result and recover the state of the application, without further need for compiler support or constraint to single-threaded applications.

Napper et al. [15] model the JVM as a state machine and replicate some states to make it fault resistant. In order to handle multithreaded applications, the authors suggest a mechanism to keep a log of the lock acquisitions and replicate them. However, this mechanism works only for uniprossessor systems and does not directly translate to concurrent JVM executions.

The idea of checkpointing the data and rollback after a failure also occurs in Transactional Memory [11]. Software transactional memory runtimes often use checkpointing or similar techniques to save and restore the state of transactional variables to consistent points in the program execution. However, Transactional Memory checkpointing targets well-defined transactions of the application and does not consider the concurrency of the TM system itself. FT-BDDT on the other hand transfers these ideas to the fault-tolerance of the runtime system, not just the application executed.

Stodghill et al. [10] have developed a way to checkpoint OpenMP applications by saving the stack of every thread and the heap to a safe location. This approach requires from the programmer to set checkpoint calls in the programm, which might not be straightforward to do. Moreover, checkpoint calls may not always be atomic actions with respect to faults or interleaving of threads. Finaly, the representation and alignment of stack, or the value of the stack pointer may itself be a point of failure, or not easy to restore to its original representation on some architectures. FT-BDDT uses a portable method of reconstructing the stack that does not rely on the previous contents of the stack or a saved value of the stack pointer.

Similar techniques can be applied in distributed systems without shared memory. FTC-Charm++ [29] stores remote, coarse-grain checkpoints in a distributed system, using peer nodes, while Schulz et al. [23] construct a global checkpoint that includes messages and reconstruct program state from the global checkpoint by replaying messages as necessary.

7. CONCLUSIONS

This paper presents FT-BDDT, a fault-tolerant execution runtime system for task-parallel programs. FT-BDDT implements micro-checkpointing that enables the runtime system to use shared memory data structures seamlessly to schedule tasks to threads, and recover from transient and permanent faults, without the need for global checkpointing. This method incurs little overhead for checkpointing as all checkpoints are small and can be done in parallel. We have tested FT-BDDT by emulating faults using various methods and found that it is able to recover seamlessly. We found the cost of recovery for transient and permanent faults to be negligible.

Acknowledgements

This work has been supported in part by the Seventh Framework Programme of the European Commission under the DeSyRe Project (`http://www.desyre.eu`), grant agreement N° 287611.

8. REFERENCES

[1] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: Building high availability systems with commodity multi-core processors. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[3] Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 2001.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the ACM symposium on Principles and Practice of Parallel Programming*, 1995.

[5] Shekhar Borkar et al. Microarchitecture and design challenges for gigascale integration. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2004.

[6] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5, January 1998.

[7] Skarlatos Dimitrios, Pratikakis Polyvios, and Pnevmatikatos Dionisios. Towards reliable task parallel programs. In *HiPEAC Workshop on Design for Reliability*, 2013.

[8] Alejandro Duran, Eduard Ayguade, Rosa M Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[9] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] Paul Stodghill Greg Bronevetsky, Keshav Pingali. Application-level checkpointing for openmp programs. In *International Conference on Supercomputing*, 2006.

[11] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, 1993.

[12] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[13] Spyros Lyberis. *Myrmics: A Scalable Runtime System for Global Address Spaces*. PhD thesis, University of Crete, August 2013.

[14] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in Los Alamos National Labratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5:329–335, 2005.

[15] Jeff Napper, Lorenzo Alvisi, and Harrick Vin. A fault-tolerant java virtual machine. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2002.

[16] S. Nomura, M.D. Sinclair, Chen-Han Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + DMR: Practical and low-overhead permanent fault detection. In *Proceedings of the International Symposium on Computer Architecture*, 2011.

[17] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *European Dependable Computing Conference (EDCC)*, pages 132–143, 2012.

[18] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2001.

[19] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[20] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the International Symposium on Computer Architecture*, 2000.

[21] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[22] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2009.

[23] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pengali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *SC*, 2004.

[24] The sequoia programming language. `http://http://sequoia.stanford.edu`.

[25] *SMP Superscalar (SMPSs) v2.3 User's Manual*, 2010.

[26] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos. BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism. In *Proceedings of the ACM symposium on Principles and Practice of Parallel Programming*, 2012. Poster paper.

[27] George Tzenakis, Angelos Papatriantafyllou, Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S. Nikolopoulos. BDDT: Block-level dynamic dependence analysis for task-based parallelism. In *Advanced Parallel Processing Technologies*, 2013.

[28] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime asynchronous fault tolerance via speculation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2012.

[29] Gengbin Zheng, Lixia Shi, and L.V. Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.