Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach

Dong Li[†], Zizhong Chen[⋆], Panruo Wu[⋆], and Jeffrey S. Vetter[†§]

[†]Oak Ridge National Laboratory, [⋆]University of California, Riverside,
[§]Georgia Institute of Technology
lid@ornl.gov, {chen, pwu011}@cs.ucr.edu, vetter@computer.org

ABSTRACT

Algorithm-based fault tolerance (ABFT) is a highly efficient resilience solution for many widely-used scientific computing kernels. However, in the context of the resilience ecosystem, ABFT is completely opaque to any underlying hardware resilience mechanisms. As a result, some data structures are over-protected by ABFT and hardware, which leads to redundant costs in terms of performance and energy. In this paper, we rethink ABFT using an integrated view including both software and hardware with the goal of improving performance and energy efficiency of ABFT-enabled applications. In particular, we study how to coordinate ABFT and error-correcting code (ECC) for main memory, and investigate the impact of this coordination on performance, energy, and resilience for ABFT-enabled applications. Scaling tests and analysis indicate that our approach saves up to 25% for system energy (and up to 40% for dynamic memory energy) with up to 18% performance improvement over traditional approaches of ABFT with ECC.

Keywords

algorithm-based fault tolerance, error-correcting code, adaptive resilience

1. INTRODUCTION

As high-end computing systems scale towards exascale, two trends will bring even more complex challenges and uncertainty in resilience and reliability. First, at the device level, the inherent reliability of hardware may decrease because of the advanced design and manufacturing techniques for power management (e.g., near-threshold voltage). Second, at the whole-system level, the sheer number of components required for an exascale system will certainly scale at least linearly in system error and fault rates compared to a similar, but smaller counterpart.

Algorithm-based fault tolerance (ABFT) is a software-based resilience solution that has attracted considerable research attention recently [7, 8, 10, 13, 14, 15, 26, 39]. By either leveraging redundant information inherent in numerical algorithms, or exploiting invariant relationships between data structures, ABFT can efficiently detect and correct errors occurring in critical application data.

ABFT has attractive characteristics that do not commonly exist in alternative resilience solutions. First, it can reduce or even eliminate the expensive periodic checkpoint/rollback; and, hence, greatly boosting performance and energy efficiency [7, 10, 26]. Second, it usually brings negligible performance loss for many widely used applications, especially when it is deployed at large scale [7, 8, 10, 14, 39]. Finally, it requires no architectural modifications and few system software modifications.

On the other hand, ABFT can only protect algorithm-specific data structures, and cannot protect all program data and instructions, such as those managed by the operating system. Note that, in the context of the resilience ecosystem, ABFT is completely opaque to any underlying hardware resilience mechanisms. Likewise, these hardware resilience mechanisms are also unaware of ABFT, and they implicitly assume that all data and instructions are uniformly vulnerable to contamination even if critical data are already protected by highly efficient ABFT (an assumption that we have recently investigated in [21]). As a result, some data structures are over-protected by both ABFT and hardware, which directly leads to redundant costs in terms of performance and energy efficiency.

In this paper, we rethink ABFT using a cooperative software-hardware approach with the goal of improving performance and energy efficiency of ABFT-enabled applications. In particular, we investigate how to integrate ABFT and hardware-based error-correcting code (ECC) for main memory, and we measure with simulation the impact of this integration on performance, energy, and resilience for ABFT-based applications.

The ECC mechanism for main memory employed by modern HPC systems is rigid. Typically, ECC protection is enforced uniformly on all main memory devices, regardless of vulnerability differences of system and application data they are assigned. Although recent research on flexible ECC enables adaptive ECC based on memory access granularity and redundancy overhead [24, 40, 41, 42], they cannot be applied to ABFT, because there is a significant semantic gap for error detection and location between ECC protection and algorithm-based protection. This gap prohibits easy combination of these two approaches. In addition, the current flexible ECC proposals employ extensive modifications

http://dx.doi.org/10.1145/2503210.2503226.

to various system components (e.g., memory management unit, and additional ECC address translation unit), which make them very difficult to implement. With those ECC proposals, data with different vulnerabilities could also interleave in the same page with a unified ECC protection, which decreases resilience guarantees.

By contrast, we propose an *explicitly-managed ECC* by ABFT, motivated, in part, by the idea of explicitly managed memory hierarchy in current architectures [6, 27]. With limited modifications to the current architecture, we propose customization of memory resilience mechanisms based on algorithm requirements, and thus, provide a method to seamlessly integrate ABFT and ECC. Furthermore, we expose critical architecture features into the OS and runtime, and introduce algorithm semantics into error detection. This change results in new performance improvements in ABFT. With ECC relaxed for most of the execution time, the ABFT-based applications are able to save up to 25% system energy (and up to 40% for dynamic memory energy) with performance improvement up to 18%, compared to the traditional uncooperative ABFT with ECC. Furthermore, this paper demonstrates the necessity and potential benefits of using a co-design and adaptive policy to direct end-to-end, overall resilience for the application and architecture.

The major contributions of the paper are:

- To the best of our knowledge, this is the first exploration that considers ABFT using a holistic view from both software and hardware; it provides the first principled understanding of interactions between ABFT and other resilience solutions.
- We classify error handling scenarios in light of ABFT and ECC integration, and further establish fault models to investigate the implication of this integration for overall system resilience.
- We enhance the memory controller and system software to bridge the semantic gap for error location and detection, and allow co-existence of multiple ECC and runtime ECC transition. Our method provides support for efficient integration of ABFT and ECC.
- We investigate how ABFT can leverage the error notification mechanism from architecture to improve performance. In cooperation with ABFT, we demonstrate better performance and energy efficiency with our ECC than with the state-of-the-art ECC mechanism.

2. BACKGROUND

2.1 Algorithm-Based Fault Tolerance

In this section, we review ABFT and briefly explain the four representative ABFT algorithms that we use in this paper.

General Matrix Multiplication. We use a fault tolerant general matrix multiplication algorithm, which targets fail-continue errors [39] (labeled as *FT-DGEMM* throughout the paper). Fail-continue means that the failed process continues working when a failure occurs. For the matrix multiplication $C = AB$, this ABFT algorithm encodes matrices A, B into a new form with checksums (the following e is a checksum vector to generate checksums):

$$A^r := \begin{bmatrix} A \\ e^T A \end{bmatrix}, B^c := \begin{bmatrix} B & Be \end{bmatrix}$$

To protect the result matrix C, new versions of A and B with checksums are employed. The result C^f includes the desired multiplication result C.

$$A^r B^c = \begin{bmatrix} AB & ABe \\ e^T AB & e^T ABe \end{bmatrix} = \begin{bmatrix} C & Ce \\ e^T C & e^T Ce \end{bmatrix} =: C^f$$

The checksum elements employed by this ABFT algorithm maintain a specific relationship between all entries within the same row or within the same column. In every few iterations of the computation, the algorithm examines checksums to detect and correct errors. With sophisticated checksum vectors, this ABFT algorithm can detect or correct multiple errors in each examining period. We apply relaxed ECC to the matrix C in later discussions.

Cholesky Factorization. We develop a fault tolerant Choleksy factorization algorithm [38] targeting fail-continue errors (labeled as *FT-Cholesky* throughout the paper). To explain FT-Cholesky, we first explain the regular algorithm without fault tolerance. Given an input matrix A ($n \times n$), the regular right looking blocked Cholesky algorithm used in popular LAPACK/ScaLAPACK factors A into a lower triangular matrix L, such that L times L transpose is A. The algorithm iteratively factors small blocks of the input matrix A as follows:

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix}$$

A_{11} is a small square block matrix with size $b \times b$; A_{21} is a column of blocks with size $(n-b) \times b$; A_{22} is a trailing matrix with size $(n-b) \times (n-b)$. The regular Cholesky algorithm follows 4 steps: (1) factor $A_{11} = L_{11} L_{11}^T$ to obtain L_{11}; (2) solve L_{21} from $A_{21} = L_{21} L_{11}^T$ using forward substitution; (3) update the trailing matrix $A_{22} = A_{22} - L_{21} L_{21}^T$; and (4) repeat the above procedure for the trailing matrix A_{22}.

To make the regular algorithm fault tolerant, FT-Cholesky introduces checksums for each block like A_{11}. These checksums maintain invariant relationships between all entries in A_{11} throughout the above 4 steps. By periodically examining checksums at each step in each iteration of the algorithm, we can detect and correct multiple errors occurring in multiple columns of all relevant blocks. FT-Cholesky protects the matrix A. It protects L as well, because the factorization happens in-place (i.e., the result L gradually overwrites A). We apply relaxed ECC to the matrix A (L as well) later.

Preconditioned Conjugate Gradient. We use a fault tolerant preconditioned conjugate gradient (CG) method with the goal of tolerating fail-continue errors [8] (labeled as *FT-CG* or *FT-Pred-CG* throughout the paper). CG solves the input equation $Ax = b$, where A is a symmetric positive-definite matrix. The regular CG is generally described in Figure 1.

Unlike the above ABFT, FT-CG does not employ checksums. Rather, they detect and correct errors based on the following numerical invariant at every few iterations of the computation.

$$\begin{aligned} p^{(i+1)^T} q^{(i)} &= 0 \\ r^{(i+1)} + Ax^{(i+1)} &= b \end{aligned} \tag{1}$$

Equations (1) are used to detect and solve errors in r, p, q, x and b. They can also be used to detect errors in M and ρ, because errors occurring in M and ρ can be propagated to

```
1 :  Compute r^(0) = b − Ax^(0), z^(0) = M^−1 r^(0), p^(0) = z^(0),
       and ρ_0 = r^(0)T z^(0) for some initial guess x^(0)
2 :  for i = 0, 1, ...
3 :     q^(i) = Ap^(i)
4 :     α_i = ρ_i / p^(i)T q^(i)
5 :     x^(i+1) = x^(i) + α_i p^(i)
6 :     r^(i+1) = r^(i) − α_i q^(i)
7 :     solve Mz^(i+1) = r^(i+1), where M = M^T
8 :     ρ_{i+1} = r^(i+1)T z^(i+1)
9 :     β_i = ρ_{i+1} / ρ_i
10:     p^(i+1) = z^(i+1) + β_i p^(i)
11:     check convergence; continue if necessary
12:  endfor
```

Figure 1: preconditioned conjugate gradient algorithm.

the next iterations and damage the orthogonality relationship established in Equations (1). We apply relaxed ECC to r, p, q, x and b in our later discussions.

High Performance Linpack. We use a fault-tolerant high performance linkpack (HPL) algorithm, targeting fail-stop errors [10] (labeled as *FT-HPL* throughout the paper). Fail-stop refers that the failed process stops working when a failure occurs. HPL is the standard benchmark to test performance and efficiency of supercomputers. It solves a linear equation $Ax = b$ by applying LU decomposition with partial pivoting on the matrix A, such that the equation can be rewritten as $PLUx = b$.

FT-HPL maintains row checksums for the matrix A at each step of the algorithm. Whenever an error occurs in the middle of the computation, the error can be recovered from the row checksum relationship. FT-HPL can protect the matrix A. Because the factorization occurs in-place, it can also protect the matrix U. Later, we apply relaxed ECC to the matrix A (U as well).

2.2 Traditional Main Memory with ECC Protection

Hardware ECC is pervasively employed in high-end servers to protect memory from errors. In general, DIMMs with ECC provide storage for regular data and redundant ECC information, and actual error detection/correction takes place at the memory controller (MC).

The single-bit-error-correcting and double-bit- error-detecting (SECDED) [19] is a widely used ECC. An 8-bit SECDED code protects 64-bit data, creating a 72-bit wide data path. As its name indicates, SECDED can tolerate a 1-bit error, and detect 2-bit errors. The chipkill-correct [12] is another ECC scheme that provides stronger reliability than SECDED and commonly exists in high-end servers [20, 33, 35]. It employs Single Symbol Correct and Double Symbol Detect (SSCDSD), and operates on b-bit symbols (b is the symbol width) rather than individual bits. All errors of all lengths within a single symbol can be corrected. With x4 DRAMs (b is also 4) and 4-check-symbol code, 32 data symbols are protected by 4 ECC symbols, creating a 144-bit data path (128 bits of data with 16 bits ECC). This is typically implemented as two ECC DIMMs, each of which has 18 chips, reading/writing two 64-byte cache lines at a time on a standard DDR3 channel [36]. This x4 chipkill configuration is common in modern architectures [2, 4].

ECC protection comes with overhead, even in fault free operations. First, ECC brings energy overhead, because the system has to fetch more data in memory with ECC than that without ECC. For chipkill, the system activates a larger number of chips than absolutely required, increasing overfetching with DRAM chips, and resulting in substantially increased energy consumption. Second, ECC protection brings performance overhead. When using chipkill, fetching a single cache line invokes multiple memory channels and forces prefetch. If there is not enough locality, the extra bits in all the active DIMMs are wasted. There are also fewer opportunities for rank-level parallelism, potentially losing performance, because a larger number of chips is made busy on every access. Third, ECC protection brings storage overhead, e.g., 12.5% extra storage required for SECDED and for 4-check symbol chipkill (x4 DRAM), 18.75%-37.5% for 3-check symbol chipkill (x8 DRAM) [36].

3. ABFT-DIRECTED FLEXIBLE ECC FOR MAIN MEMORY

We propose a flexible memory protection mechanism that allows ABFT to control the usage of ECC. By coordinating ABFT and ECC with appropriate protection level, we propose an elastic architecture with better energy efficiency and performance. Our general design policy is to evolve the existing ECC mechanisms and avoid drastic modification to the system software stack. We review our design in this section.

3.1 Architecture

Our proposed architecture supports multiple ECC mechanisms simultaneously in a node's main memory system These different ECC mechanisms has different implications for reliability, performance and energy cost. We use x4 DRAM as an example to explain the idea, but our approach easily generalizes to other DRAM chips (e.g., x8 chips). In this design, we evaluate three levels of protection using two ECC mechanisms. (1) Chipkill-correct ECC to provide *strong protection*. The two physical memory channels (typically 72-bit wide) works in lock-step to construct a 144-bit wide logical channel, protecting read and write operations for 64-byte cache lines. (2) SECDED ECC to provide *weak protection*. Each physical memory channel enables 72-bit memory access with 8 bits for ECC. (3)*No ECC protection*, which uses 64-bit wide memory channel for regular 64-byte cache lines.

Each memory channel is physically 72-bit wide for all ECC mechanisms, with 8 bits disabled (or ignored) for no ECC protection. The memory controller (MC) has ECC logic for both SECDED and chipkill. As shown in [23], a MC can include multiple ECC logic mechanisms, and it has marginal increases in power consumption and latency, and has an acceptable increase in area size. Each MC has two physical memory channels, and can control them independently for SECDED and no-ECC or simultaneously for chipkill.

We control ECC protection level at the granularity of page frame. By default, page frames are protected with a strong protection scheme, unless the users specify which page frames can be protected by relaxed ECC because of protection by ABFT (or other software protection schemes). In particular, in a user scenario, a block of page frames are protected by SECDED (or no ECC) plus ABFT, while the

other page frames are protected by chipkill. In another user scenario, a block of page frames are protected by ABFT without ECC, while the other page frames are protected by SECDED (i.e., a relatively strong ECC). Controlling ECC protection at the granularity of page frame minimizes modification to virtual memory management at OS (which will be discussed in the next subsection).

Both chipkill and SECDED have dedicated ECC DRAM chips for ECC code. They have the same storage overhead (i.e., 4 chips out of each 36 chips for chipkill, or 2 chips out of each 18 chips for SECDED). In addition, we use a data layout compatible to both chipkill and SECDED [24, 41]. With the same ECC storage overhead and compatible data layout, switching ECC schemes for a specific block of page frames does not disrupt existing data. In addition, using this data layout means we do not need to change existing DRAM devices.

We introduce a set of configurable registers (named *ECC registers*) into the MC to specify address ranges of each block of page frames protected by ABFT, and to specify which ECC scheme is applied to each frame. The registers can be mapped into memory address space to enable programmability of ECC strategies. This register-based design is feasible, given wide adoption of registers in the modern MC (e.g., timing registers and refresh control registers to configure the MC to meet DDRx specifications). Whenever read/write requests issued from the last level cache, the MC checks if the requested cache line resides in the page frames specified by ECC registers. If yes, the corresponding ECC scheme assigned to the page frames is enforced by the MC.

Besides ECC logic, the MC also manages error detection. In particular, once an ECC-uncorrectable error occurs, MC locates the memory fault site (i.e., finding the chip/row/column), and records it with two registers (called *error registers*). The error registers are mapped into the memory address space to allow OS to read. Locating a memory fault site demands the support from BIOS and the chipset. Based on recent reported work on Google servers and Blue Gene/L [20], this kind of support exists in practice. MC reports errors to the processor by generating an interrupt. Given the rareness of these errors, the performance impact of these costly interrupts has not been a major problem. For those very frequent occurrences of errors because of a hard fault, the critical impact of these interrupts will be obvious to users or system administrator so that they can replace DIMMs or invoke OS to remap data to the spare page frames (i.e., using memory page retire and data migration [34]). It is possible that the new errors happen and flush away old ones before ABFT (or other application-level fault tolerance) is able to correct old ones. To solve this problem, we use n registers ($n = 6$) to record $n/2$ or more error events (multiple error events may happen at the same fault site). The determination of n depends on the possibility of occurrences of $n/2$ errors within an error-examination period of ABFT (typically tens to hundreds of seconds). Given the typical error rates shown in Table 5 and short error-examination period of ABFT, $n = 6$ per MC is a sufficient configuration.

One lingering concern is error propagation. If an error occurs in ABFT protected data, and it is not corrected quickly by ABFT, it may be propagated to data which is outside the scope of ABFT. Fortunately, ABFT guarantees that error propagation is limited within the ABFT error-detectable data and the errors are still correctable by ABFT.

Figure 2 depicts our evaluation architecture. Coupled with system software, this architecture and memory organization provide flexibility to customize ECC protection as guided by the needs of ABFT.

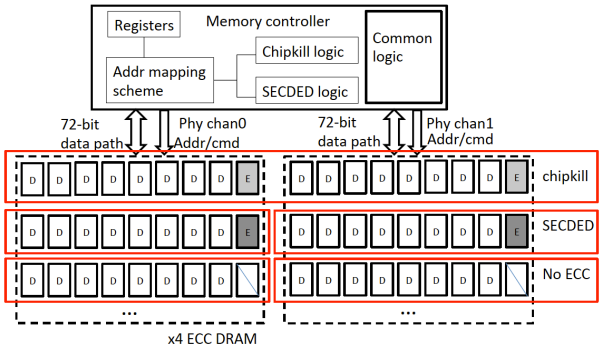


Figure 2: Architecture overview. D represents a byte in a 64B block, and E represents an 8-bit ECC.

3.2 Software

In order to support efficient integration of ABFT and ECC, system software has to be changed. In addition, because of the support from hardware, we can optimize ABFT to improve performance.

3.2.1 System Software Support

OS handles the ECC-error interrupt. In particular, in response to an interrupt, an interrupt handler reads memory-mapped registers for the fault site information and determines if the corrupted data are protected by ABFT. If they are, the virtual address of the corrupted data is exposed to a memory space shared between the kernel and the user spaces (e.g., via sysfs in linux), and left to ABFT to correct. If they are not, the system will go to the panic mode as usual cases. The existing high-end systems always go to the panic mode [17, 29], and maybe log the error, when an ECC-unrecoverable error occurs. This conservative protection strategy is based on the assumption that OS context and the application-critical data may be mangled by errors, such that continuous execution will lead to further corruption. This strategy is blind to where the error happens and whether the application knowledge can be exploited to correct the error. As a result, the system has to resort to checkpoint/restart more frequently than absolutely necessary. In contrast, our method coordinates software and hardware, and makes best efforts to correct errors with ABFT and avoid restart, resulting in improvement of performance and energy efficiency.

To determine if the corrupted data are protected by ABFT, OS must be aware of the address mapping scheme employed in MC (i.e., how to convert a physical address into a specific location in DIMMs), and basic memory organization, such that OS can derive the physical address given fault site information. We can implement the address mapping scheme as a kernel module to enable configurability of address derivation, and expose memory organization as configurable components of the virtual file system to allow user to explicitly inform OS of memory organization. Note that although MC has all information (address mapping and memory organi-

zation) to derive physical addresses, we ask OS to do that to simplify logic in MC.

By default, all memory, including OS kernel memory and user memory, is heavily protected by chipkill or SECDED, as in current high-end servers. However, the users can control which data structures should be protected by which weaker ECC protection schemes. We introduce three ECC control APIs.

- void *malloc_ecc(size_t n, int ecc_type);
- void free_ecc(void *ptr);
- void assign_ecc(void *ptr, int ecc_type);

A call to *malloc_ecc* allocates contiguous physical pages, and asks OS to set the address range of those pages and set assigned ECC schemes into ECC registers in MC. There are limited number of ECC registers in MC (16 in our design for setting 8 address ranges), due to concerns of controlling area size of MC. This indicates that limited number of ABFT protected data structures are recognized in MC. However, multiple data structures may use the same relaxed ECC scheme, and their address ranges may be combined to use the same ECC registers. Furthermore, typically only a few data structures are protected by ABFT (no larger than 5 in the 4 representative ABFT), hence using those limited number of the registers are sufficient. A call to *free_ecc* frees a block of memory previously allocated with *malloc_ecc*. As a result, the ECC address registers are updated to eliminate the freed memory range. A call to *assign_ecc* assigns a specific ECC scheme to a block of pages allocated by *malloc_ecc*. This allows dynamic refinement of ECC protection based on the application needs. To maintain a consistent ECC protection when paging in from auxiliary storage, we also incorporate ECC type in the *page* data structure such that data can be fetched into physical memory devices with desired ECC protection.

To leverage the above system and architecture, ABFT needs to be slightly changed. In particular, at the initial phase, ABFT allocates those data structures protected by ABFT by the above ECC control APIs and records their virtual addresses; at the error correction phase, ABFT identifies which data element should be recovered by recognizing the virtual address of the corrupted data exposed by OS.

3.2.2 ABFT Optimization

While ABFT introduces little overhead in most cases, depending on the input problem size and algorithm, the worst case ABFT overhead (which are often small scale deployment) can be as high as 35% [14]. For those worst case ABFT scenarios, the proposed cooperative software-hardware approach can provide an opportunity to reduce the overhead of ABFT when tolerating fail-continue errors. The overhead of fail-continue ABFT comes from two parts: the increased computation to maintain checksums (if the checksum is employed by ABFT), and the periodical verification to detect/locate errors. Figure 3 displays the overhead breakdown for the three ABFT for fail-continue errors, each of which runs one task with the input problems described in Table 3. We observe that the verification is responsible for a large part of the overhead.

Because hardware and OS can explicitly locate corrupted data, ABFT can significantly simplify its verification phase when a worst ABFT case occurs. In these bad cases, instead of recomputing checksum and making verification, ABFT can just check error information exposed by OS and hardware. This overhead, which involves reading shared memory and mapping virtual addresses of the corrupted data to the specific elements of the protected data structure, is much smaller than the worst case ABFT overhead. We simplify the verification phases of the three ABFT in their worst case scenarios and rerun them without any ECC relaxing. Table 1 demonstrates that the simplified ABFT achieve 6.0%-12.2% performance improvement.

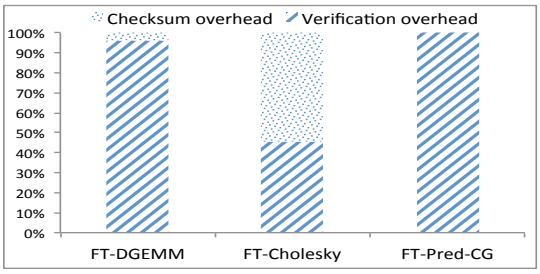


Figure 3: ABFT overhead breakdown

Table 1: ABFT performance improvement with simplified verification

	FT-DGEMM	FT-Cholesky	FT-Pred-CG
Improvement	8.6%	6.0%	12.2%

4. FAULT MODELS

The combination of ABFT with either weak ECC schemes or no ECC protection has complicated implications on system performance, energy consumption and resilience. We discuss those implications in this section. Our discussion is general, and targets on soft/hard errors and single/multi-bit errors. The notations for the discussion are summarized in Table 2.

Table 2: Notations for the fault models

$MTTF$	Mean time to failure
$MTTF_{hetero}$	Mean time to failure for memory with heterogeneous ECC protection
FR	Memory failure rate (The number of failures per time unit per Mbit)
MC_a	Memory capacity per node
N	Number of nodes
A	The average DIMM lifetime
$f(A)$	The age function relating error rates with DIMM lifetime
mc_i, fr_i	The memory capacity and the failure rate of a memory region with ECC i
$f_i(A)$	The age function of a memory region with ECC i
T_0	The native execution time without ECC protection
τ	The performance impact ratio of ECC
N_e	The number of main memory related errors

The introduction of relaxed ECC has impact on the number of errors. In general, MTTF, an indicator of the error rate, is a function of memory capacity, node counts, and memory ages. MTTF without considering ECC is formalized with Equation (2).

$$MTTF = 1/(FR * MC_a * f(A) * N) \quad (2)$$

We assume that all nodes use the same memory organization and the same memory technologies (e.g., 90nm DDR3 SDRAM). This assumption is true for most of high-end computing systems. With ABFT and flexible ECC, a single node can equip main memory with heterogeneous ECC protection against faults. We formalize the above statement using Equation (3).

$$MTTF_{hetero} = 1/(\sum_i (fr_i * mc_i * f_i(A)) * N) \quad (3)$$

We assume that the native execution time of ABFT without any protection of ECC is T_0. Using various ECC strategies can impact T_0 by τ, We can estimate the total number of errors with heterogeneous ECC protection using Equation (4).

$$N_e = T_0 * (1 + \tau)/MTTF_{hetero} \quad (4)$$

With a weak ECC or no ECC, ABFT execution potentially faces more errors than ABFT execution with a strong ECC. Based on the above fault models, we can calculate number of errors, based on which we can derive recovery cost and energy benefit for ABFT plus relaxed ECC (ARE). we use this method to evaluate ARE in large scales and compare it with the traditional setting (i.e., ABFT plus strong ECC (ASE)) in Section 5.2.

With error-free execution, ARE has superior performance and energy saving than ASE because of strong ECC overhead in ASE discussed in Section 2 and shown in Section 5.1. When errors occur, comparing ARE and ASE is challenging because of various error recovery cost and error rate associated with various ECC and ABFT. We classify error scenarios based on whether ABFT and strong ECC can correct the errors, and compare ARE and ASE as follows. We do not consider cases where the errors can be corrected by both weak ECC in ARE and strong ECC in ASE, because ARE and ASE have similar recovery cost for those cases.

Case 1: both strong ECC and ABFT can correct the errors. When this case happens, ARE relies on ABFT to correct the errors, while ASE uses strong ECC to correct them. For ABFT, the error correction cost varies across algorithms: for the checksum-based approaches (FT-DGEMM, FT-Cholesky, and FT-HPL), the correction cost is to restore the corrupted state using checksums, which involves some floating point operations and memory references on the specific matrix rows/columns; for FT-CG that leverages algorithm-inherent invariant to correct errors, the correction cost is comparable to compute a matrix-vector multiplication. On the other hand, with ASE, the error correction by strong ECC typically just takes a few clock cycles [12, 23]. The error correction latency in strong ECC may not even be added into the critical path, because the latency can be hidden by memory parallelism. In terms of energy consumption for error correction, using strong ECC to correct an error takes less than 1 pJ [23], mainly consumed by correction logic in the memory controller; using ABFT to correct an error takes up to hundreds of Joules, depending on the input numerical problem size in ABFT.

In summary, using ABFT to correct errors due to relaxed ECC is much costly than using a strong ECC. Hence, if the error rate is high enough, the correction cost using ABFT can potentially outweigh energy and performance benefit we obtain from relaxed ECC. For those cases with high error rate, we should employ strong ECC throughout all data, even if we have ABFT protection, to avoid potential performance loss and energy cost with ABFT-based error recovery. Note that even though using ABFT is costly to correct errors, for those cases with high error rate it is still beneficial to use ABFT with strong ECC, because ABFT can solve errors uncorrectable by ECC, and reduce or even eliminate expensive checkpoint/restart. Checkpoint/restart is generally much more costly than ABFT.

We formalize the above discussion, and calculate the error rate sweet point ($MTTF_{thr}$) to determine if ARE should be used or not. The performance of using ARE is a multivariable function. Firstly, the performance is related with which specific weak ECC mechanism (or no ECC) is employed on ABFT protected data. Since different ECC mechanism is accompanied with different performance and energy cost, we must know which ECC mechanism is used for comparison purpose to estimate benefit. Second, the performance is related with the number of accesses to memory cell arrays protected with relaxed ECC. Third, the performance is related with memory architecture and memory access pattern of the application, because they have compound impact on memory access parallelism and data locality.

The performance loss of using ARE (T_c) comes from the recovery cost, shown in Equation (5). We assume that the recovery cost per recovery operation is a constant (t_c). We also assume each recovery corrects one error. This is a conservative assumption. Depending on the error patterns and recovery algorithm, a single recovery operation may recover multiple errors. For example, the invariant-based error recovery in FT-CG can correct multiple errors in a vector. Hence, Equation (5) gives the worst performance loss (τ_{are} in the equation quantifies performance impact of ARE):

$$T_c = N_e * t_c = T_0 * (1 + \tau_{are})/MTTF_{hetero} * t_c \quad (5)$$

Based on the above performance analysis, we can calculate the performance benefit of using ARE, shown in Equation (6) (τ_{ase} in the equation quantifies performance impact of ASE):

$$\Delta T = T_0 * (1 + \tau_{ase}) - T_0 * (1 + \tau_{are}) = T_0 * (\tau_{ase} - \tau_{are}) \quad (6)$$

To get benefit from ARE, we must have performance benefit larger than performance loss. Based on this target, we can derive the MTTF threshold for performance benefit ($MTTF_{thr\_t}$), shown in Equation (7).

$$\begin{aligned} & T_c < \Delta T \quad \Rightarrow \\ & \frac{T_0 * (1 + \tau_{are})}{MTTF_{hetero}} * t_c < T_0 * (\tau_{ase} - \tau_{are}) \quad \Rightarrow \\ & MTTF_{thr\_t} = \frac{t_c}{\frac{\tau_{ase}+1}{\tau_{are}+1} - 1} \end{aligned} \quad (7)$$

We can follow the similar process to derive the MTTF threshold for energy benefit ($MTTF_{thr\_en}$). To guarantee no performance loss while having energy benefit, the MTTF threshold should be chosen as Equation (8).

$$MTTF_{thr} = MAX(MTTF_{thr\_t}, MTTF_{thr\_en}) \quad (8)$$

Case 2: ABFT can correct the errors while strong ECC cannot. To give a concrete example for this case, we assume that a FT-DGEMM has multiple errors occurred in

multiple matrix columns. Those errors are still correctable by ABFT. However, those widely dispersed error bits may be caused by multiple memory bank column failures beyond the correction capability of chipkill (e.g., errors distributed on 33 data symbols) and SECDED (e.g., 4-bits errors).

When this case happens, ASE may crash the system, if the errors are not exposed to the application level and ABFT has no opportunity to correct them. As a result, the system has to restart from the last checkpoint. If the errors are exposed to the application in ASE, then the errors are detected and corrected by ABFT, similar to in ARE. For the former scenario (i.e., system crashes), ASE has to pay much higher recovery cost than ARE; for the latter scenario, the recovery cost is about the same in both ARE and ASE, but ARE wins over ASE in general, due to benefits in error-free execution.

Case 3: strong ECC can correct the errors while ABFT cannot. When this case happens, ASE can continue execution without falling back to the previous checkpoint; ARE, on the other hand, cannot correct the errors and has to restart from the last checkpoint. The performance loss and energy cost in ARE are those in association with the application restart. Depending on when the last checkpoint and the errors occur, ARE can be significantly inferior than ASE, in terms of both performance and energy consumption.

The occurrences of this case depends on the error pattern, i.e., how the error bits are distributed within the ABFT protected data structure and how they are distributed between memory devices. This case may be rare because of the following reasons. ABFT cannot correct the errors when they frequently and collectively happen within specific data bits of a data structure (e.g., a specific matrix row). However, the frequent occurrences of such coincident errors within a short time (i.e., the time period for ABFT examining errors, typically a few tens of computing iterations) is rare. Also, using weak ECC in ARE can further reduces the occurrent opportunities of those errors. In addition, even if this type of burst errors happens, it is highly possible that ECC cannot fully correct them either, because of the limitation inherent in ECC (e.g., the number of correctable bits).

Case 4: neither strong ECC nor ABFT can correct the errors. This case happens when the errors occur beyond the correction capabilities of both strong ECC and ABFT (e.g., errors highly disperse across memory devices, and happen frequently). Both ARE and ASE have to rely on the checkpoint/restart scheme. Hence, both of them have the same recovery cost. However, they differ during error-free execution. As discussed before, ARE can save energy and improve performance over ASE for error-free execution.

Discussion: In general, given the rareness of errors, ARE wins over ASE in terms of performance and energy for most of cases. As shown in our scaling tests in Section 5.2, energy benefit of ARE is usually much larger than recovery cost, even if in very large scales with frequent occurrence of errors. However, if the error rates are extremely high or the errors frequently and coincidentally happen within specific data bits, ARE loses to ASE because of high recovery cost, which is rare in real cases.

5. EVALUATION

We evaluate coordination of ABFT and ECC in this section. We use McSim [28], a Pin [30] based manycore simulation infrastructure. McSim provides event-driven timing simulation and models cores, caches, directories, on-chip networks, and memory channels. We modify McSim to integrate DRAMSim2, a memory system simulator [31], to account for the impact of the proposed system on memory power, bandwidth and performance. We also modify DRAMSim2 to implement the proposed memory controller and memory organization. We estimate processor power consumption using an IPC-based linear scaling of the maximum power consumption of a 45nm Intel Xeon, similar to [3, 40]. We estimate DRAM power consumption using a power model developed by Micron Corporation [1] and embedded within DRAMSim2. To control the occurrences of memory errors for resilience study, we combine McSim with BIFIT [21], a Pin-based fault injection infrastructure, to enable fault injection at specific time and data location. Figure 4 generally depicts this evaluation platform. Table 3 summarizes important system parameters for simulation. During the simulation, we skip the initialization phases of each algorithm and simulate a few iterations or representative computation phases. Different ECC strategies have different implications on performance, energy consumption and resilience. We study error-free scenarios in Section 5.1, and study error scenarios with scaling tests and analytical models in Section 5.2. We compare our work with the state-of-the-art ECC in Section 5.3.

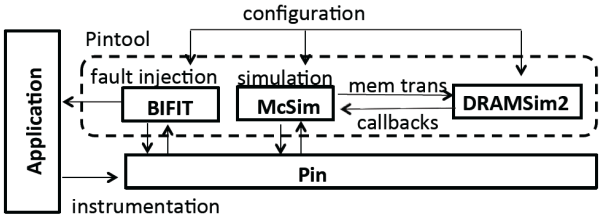


Figure 4: Simulation framework to evaluate coordination of ABFT and ECC

Table 3: System parameters for simulation

Parameter	Value
Processor	4 in-order cores and 4 threads per core
L1 cache	split I/D caches, each 16KB, 4way, 64B block, private cache
L2 cache	a unified 8MB cache, 16way, 64B block, shared cache
Clock rate	2 GHz
DRAM device	DDR3-667Mhz, x4, 1.5v
Memory organization	4 memory channels, 2DIMMs per channel, 4ranks per DIMM, 8 banks per rank
Capacity	8GB
Row buffer policy	open
Chipkill	128b data+16b ECC, 2 channels
SECDED	64b data+8b ECC, 1 channel
FT-DGEMM	matrix dim per task: 3000 × 3000, dp
FT-Cholesky	matrix dim per task: 3000 × 3000, dp
FT-Pred-CG	matrix dim per task: 3000 × 3000, dp
FT-HPL	matrix dim per task: 8192 × 8192, dp

5.1 Basic Tests

We investigate ABFT in combination with various ECC strategies. We perform the simulation with the smallest-scale deployment for each ABFT. In particular, for FT-HPL, we run each simulation with 4 MPI tasks (a 2x2 process

grid); for FT-DGEMM, FT-Cholesky and FT-CG, we run each simulation with one task. For each ABFT, we perform simulation with 6 ECC strategies: (1) ABFT without ECC (labeled as *No ECC*); (2) ABFT with chipkill applied to all data (labeled as *W_CK*); (3) ABFT without ECC for those ABFT protected data and with chipkill applied to other data (labeled as *P_CK+No_ECC*); (4) ABFT with SECDED applied to all data (labeled as *W_SD*); (5) ABFT without ECC for those ABFT protected data and with SECDED applied to other data (labeled as *P_SD+No_ECC*); (6) ABFT with chipkill applied to those data without ABFT protection and with SECDED applied to those data with ABFT protection (labeled as *P_CK+P_SD*). We call the tests 2 and 4 *whole ECC*, and call the tests 3, 5, 6 *partial ECC* in the later discussion.

Figure 5 presents memory energy data for the 6 ECC strategies. The results are normalized to the ones for *No ECC* tests (the baseline cases). We notice that chipkill greatly increases energy consumption. For FT-CG, the most memory intensive ABFT in our tests, there is 68% increase in memory energy. This large energy consumption is because chipkill must activate twice the number of DIMMs to access a single cacheline, which consumes more power and bandwidth. In addition, without enough locality overfetching data can block parallel accesses to various DIMMs, which hurts performance and further causes extra energy consumption. With relaxed chipkill protection on ABFT protected data, we have significant energy benefit. In particular, comparing with the tests 2 (i.e., whole chipkill), the tests 3 and 6 (i.e., partial chipkill) result in 49% and 48% energy saving for FT-DGEMM, and 38% and 33% energy saving for FT-CG. The tests 6 consume slightly more energy than the tests 3, because of the application of the second ECC mechanism, SECDED. However the tests 6 can be more energy efficient than the tests 3 when considering error recovery cost in large scales, shown in Section 5.2. SECDED is not as costly as chipkill, shown by the tests 4 (whole SECDED), but it still results in about 12% more energy consumption in average than those cases without ECC. With partial ECC (the tests 5), we reduce energy consumption for SECDED by 11% for FT-DGEMM. We also notice that dynamic memory energy is more sensitive to ECC strategies than standby memory energy, because ECC overhead happens when MC actively fetch/write back data from/to the memory cell arrays.

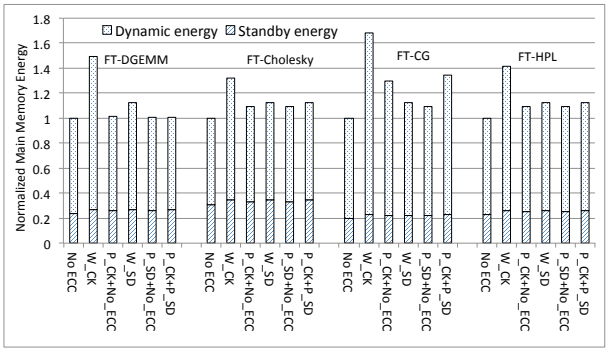


Figure 5: Memory energy for ABFT with different ECC strategies

To further understand energy consumption data, we profile last level cache misses and separately count accesses to blocks with ABFT protection and without ABFT protection. The results are shown in Table 4. FT-DGEMM has the largest ratio of cacheline references with ABFT protection to cacheline references without ABFT. This explains why the three partial ECC strategies have the similar energy consumption in FT-GDEMM, shown in Figure 5. In particular, the small number of references to data without ABFT protection make FT-DGEMM not sensitive to what kind of ECC protection is applied to those data without ABFT protection. This is in contrast with FT-CG, which has the smallest ratio. We also notice that the dynamic memory energy saving in percentage shown in Figure 5 is generally much smaller than the possible energy saving estimated based on the reference ratio shown in the table. We attribute it to row buffers in memory devices. The row buffer serves as a fast cache. If cache blocks can be fetched from row buffers, then memory arrays are not accessed, which saves some ECC energy overhead. Hence, if access locality is good (i.e., the row buffer hit rate is high), then the dynamic energy saving is limited when we apply partial ECC.

Table 4: Classification of cacheline accesses based on their ABFT protection

ABFT	#Ref to blocks w/t ABFT	#Ref to blocks w/o ABFT	Ratio
FT-DGEMM	158667478	242534	654
FT-Cholesky	4589419	335321	14
FT-CG	413087770	158081435	3
FT-HPL	90397449	4548042	20

Figure 6 displays system energy normalized to the ones for the tests 1 (i.e., *No ECC*). We report processor and memory energy, because they are the major energy consumption components within the system. We observe that processor energy varies with ECC strategies. This is especially true for the memory intensive FT-CG, because ECC impacts memory parallelism and bandwidth which in turn affects instruction issuing and scheduling in processors. In addition, we notice system energy saving because of using relaxed ECC. Comparing with the whole chipkill, partial chipkill results in up to 22%, 8%, 25% and 10% energy saving for FT-DGEMM, FT-Cholesky, FT-CG and FT-HPL respectively. Comparing with whole SECDED, partial SECDED results in up to 5% energy saving (the FT-DGEMM case).

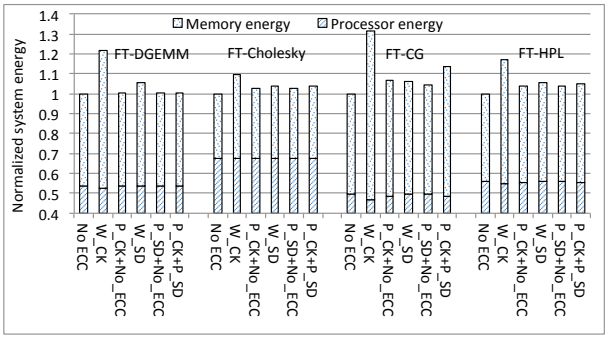


Figure 6: System energy for ABFT with different ECC strategies

Figure 7 displays performance (instruction per cycle or IPC) normalized to the ones for the tests 1 (i.e., *No ECC*). We notice that selectively applying ECC because of ABFT can make performance comparable to the ones without ap-

plying ECC. This is especially true for FT-DGEMM and FT-Cholesky (i.e., the performance of tests 3 is pretty close to that of tests 1). We also notice that performance variance across ECC strategies is generally smaller than energy variance, because memory parallelism can partially hide memory access latency due to ECC protection.

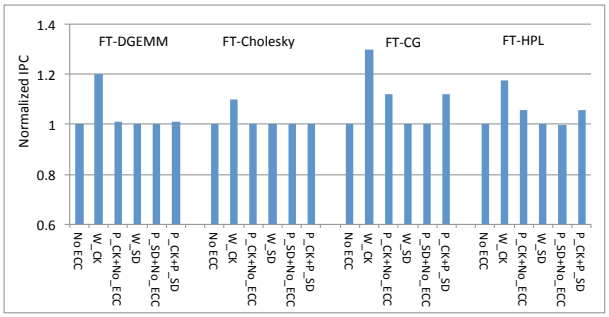


Figure 7: Performance for ABFT with different ECC strategies

5.2 Scaling Tests with Fault Modeling

We discuss how the system scale and error rate impact energy efficiency and resilience of our proposed schemes in this section. In general, large scale systems suffer from higher error rate than smaller systems. Given the relatively large recovery cost of ABFT, we want to investigate whether it is beneficial to use ABFT in combination with various ECC strategies when the number of errors is large and varies across scales. We compare energy benefit and recovery cost. Energy benefit refers to system energy saving because of applying relaxed ECC on those ABFT protected data. For partial chipkill (*P_CK+No_ECC* and *P_CK+P_SD*), the energy baseline for calculating energy benefit is the energy consumption when *W_CK* is applied; for partial SECDED (*P_SD+No_ECC*), the baseline is the energy consumption when *W_SD* is applied. Recovery cost refers to system energy consumption used to correct errors with ABFT instead of with ECC. We study FT-CG in this section, because its recovery cost is higher than other ABFT, and hence represents the worst cases when using ABFT to recover errors.

We assume that bit errors are uniformly distributed across time and space, and independent of each other; the memory aging effect is reflected in Table 5. Hence, given application execution time and memory footprint, we can analytically estimate number of errors for ABFT deployed in large scales, based on the error rates shown in Table 5 and Equation 4. In addition, we consider the case 1 errors (see Section 4) which are correctable by both ABFT and ECC, because they are most likely to happen in reality, and it is arguable whether ABFT can bring energy advantage for those cases. In summary, we inject the errors into ABFT protected data for the case 1 errors, and the number of injected errors across scales is based on the analytical models.

When calculating energy benefit, we must have performance, processor core energy, and memory energy data. For performance data, we factor in the parallel efficiency when scaling from a single process to large scales based on workload characterization and performance projection [5, 37]. To calculate processor core energy, we must know processor core power. We assume that processor core power remains constant as we scale the system. For strong scaling tests, this assumption may not be valid, but the processor energy variance across different schemes is minor, based on the results shown in Figure 6. Hence this assumption does not significantly impact energy benefit calculation (i.e., processor energy is mostly zero out when calculating energy benefit). To calculate standby memory energy, we first measure it with the simulator for a single ABFT process and then scale down the memory energy for strong scaling results or directly use it for weak scaling results for each ABFT process. To calculate dynamic memory energy, we first profile last level cache misses on a single process for each scaling test like Section 5.1, and then use dynamic memory energy from Section 5.1 as baseline cases and scale it down in proportion to the cache miss ratio of the calculation cases and the base line cases. To quantify recovery cost, we use BIFIT to inject errors and measure energy consumption for the recovery phase. We also assume one recovery operation recovers one error, which conservatively maximizes recovery cost.

Table 5: Error rate with ECC in place (FIT=failures per billion hours)

ECC Protection	Error Rate (FIT/Mbit)
No ECC	5000 [23, 25]
Chipkill correct	0.02 [25, 34]
SECDED	1300 [25, 36]

Figure 8 displays the weak scaling results with matrix dimension 3000×3000 on a single FT-CG process. For the weak scaling, the memory footprint increases as the system scales up, hence the number of errors increases. As a result, the recovery cost increases quickly shown in the right figure. However, the energy benefit increases as well. The increasing of energy benefit and recovery cost is almost in proportion to the increasing of system scales. The energy benefit is also much larger than the recovery cost in general. We also notice that the scheme *P_CK+P_SD* has the similar energy benefit to *P_CK+No_ECC* while the recover cost is much smaller than other schemes in large scales. This is because of using SECDED to protect those ABFT protected data. *P_CK+P_SD* pays limited energy cost for using SECDED while reducing the possibility of using expensive ABFT to recover, hence it is more energy efficient than other strategies.

Figure 9 presents the strong scaling results. In order to increase memory footprint and memory error counts, we use a mixture of strong and weak scaling, similar to a deployment suggested by [37]. In [37], they use strong scaling as far as possible, and then increase the problem size with weak scaling, in order to reach the concurrencies required in large scales. Following their methodology, in our cases we deploy weak scaling using 100 FT-CG processes (each with matrix dimension 12K×12K), and then use strong scaling on them to collect strong scaling results. We first notice that the energy benefit increases as system scales up and then decreases afterwards. This is because of the contradicting effects of system scaling on ABFT and ECC. On one hand, scaling the application involves more processes, each of which brings energy benefit because of ECC relaxation. On the other hand, as the system scales up, the numerical problem per FT-CG becomes smaller, and hence the main memory accesses become less because of caching (ECC relaxation becomes less efficient for energy saving). Therefore, there is a sweet point for energy benefit for using relaxed

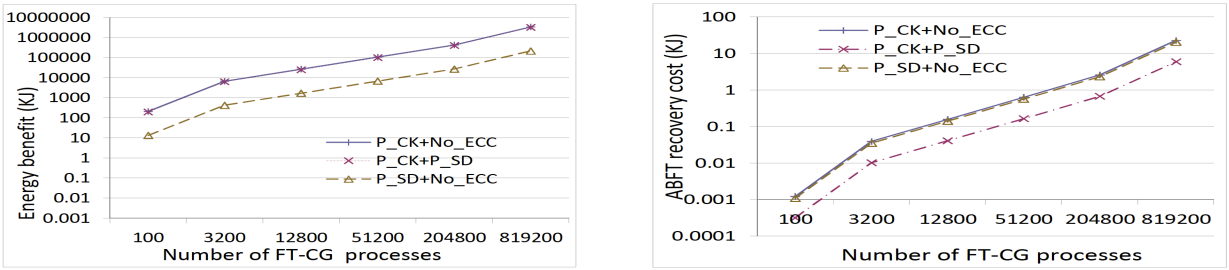


Figure 8: Weak scaling tests to compare energy benefit and ABFT recovery cost with fault modeling

ECC for strong scaling cases. We further notice that the recovery cost becomes smaller as the system scales up. This is because the numerical problem per FT-CG process becomes smaller and the recovery operation becomes cheaper as system strong scaling. In general, the energy benefit is larger than the recovery cost in our cases. We also notice that *P_CK+P_SD* is more energy efficient than other schemes, consistent with what we have observed in the weak scaling.

5.3 Comparison with State-of-the-Art ECC

Dynamic Granularity Memory System (DGMS) is a hardware-based flexible ECC mechanism [42]. Like other flexible ECC work, this state-of-the-art mechanism cannot be applied to ABFT, due to the lack of support for error detection and location for ABFT (see discussions in Sections 1 and 6). However we compare DGMS with our work for error-free cases. DGMS employs spatial pattern prediction to select which ECC should be chosen based on memory access granularities. To use DGMS, we choose chipkill (64 bytes access granularity, the same as before) for strong protection and SECDED (16 bytes access granularity [42], different from the previous SECDED) for relaxed ECC protection. To make comparison fair, we also use the same chipkill and SECDED in our scheme for strong and weak protection respectively. We implement prediction controller, spatial pattern predictor and use sub-ranked DRAM employed in DGMS.

Figure 10 display the results for FT-DGEMM (a representative ABFT with high spatial locality), and FT-Pred-CG (a representative ABFT with relatively low spatial locality). These results are normalized to those without ECC. We found that the performance and energy for DGEMM with DGMS is similar to those with *W_CK*. Further investigation reveals that all memory accesses are attributed with coarse-grained chipkill protection, because FT-DGEMM has high spatial locality and DGMS associates high locality accesses with chipkill. Our method, however, enables 18% performance improvement and 49% memory energy saving over DGMS, because ABFT and ECC cooperation enables weak ECC protection for ABFT protected data. For CG, performance with DGMS is close to that with our method, but memory energy consumption is still about 24% more. Further investigation reveals that DGMS employs chipkill for some memory accesses to p and q, while our method employs cheaper SECDED. Without ABFT knowledge, DGMS simply bases its ECC decision on memory access tracing, which results in costly ECC assignment. Note that the energy consumption of new hardware components (e.g., predictors and register/demux) in DGMS is not considered. These components will add further energy consumption in DGMS.

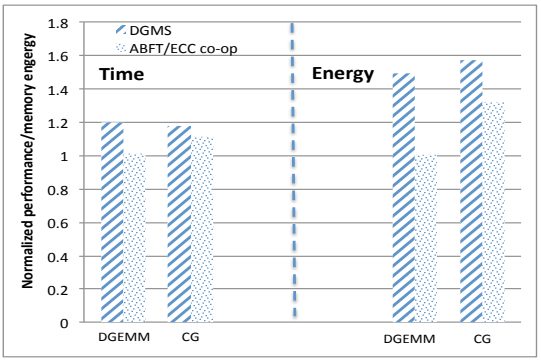


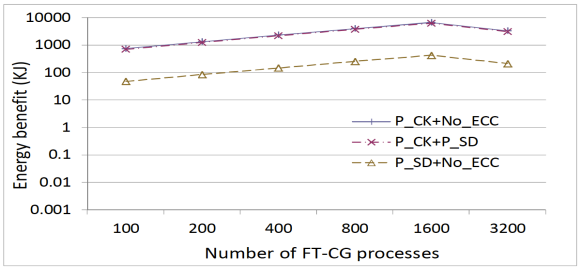
Figure 10: Performance and energy comparison between the state-of-the-art ECC (DGMS) and our work

6. RELATED WORK

Algorithm-Based Fault Tolerance: ABFT has been actively researched for a handful of popular algorithms, including general matrix multiplication [39], iterative methods for solving linear equations [8], factorization (Cholesky [38], LU [9, 10, 14] and QR [14]). However, previous work focuses on the algorithms themselves and never uses a holistic view of the entire resilience ecosystem. Our work is the first exploration that considers ABFT using a collaborative view from both software and hardware.

Software and Hardware Cooperation for System Resilience: Software and hardware cooperation has been investigated to solve resilience challenges. Ho et al. [18] describes a language-level mechanism that asks programmers to explicitly point out vulnerability of code regions, while hardware provides adaptive dual-modular redundancy. The SWAT system [22, 32] relies on software to detect errors by watching anomalous software behavior and likely program invariants. They use hardware checkpoint to recovery. Erez et al. [16] use software-based fault detection (instruction replication and kernel re-execution) and use hardware checkpointing. Kruijf et al. [11] introduce an ISA extension for compilers and programmers to mark code regions for software recovery, based on which hardware can relax reliability constraints.

Our work differs from the above approaches. First, we do not rely on specific program constructs to mark vulnerable code regions. Instead, we exploit resilience implications embedded at the algorithm level to identify code vulnerability. Second, we expose important architectural features to OS, and tightly couple OS, runtime, ABFT and hardware to detect and identify errors. Our method results in the simplification of algorithm and hardware designs.



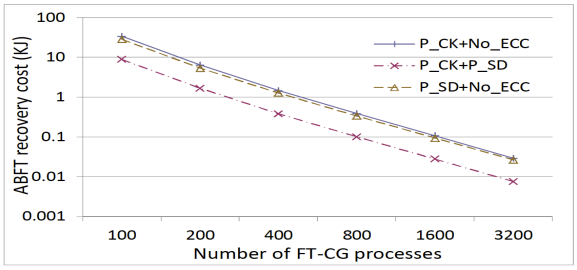


Figure 9: Strong scaling tests to compare energy benefit and ABFT recovery cost with fault modeling

Flexible ECC: Yoon and Erez [40] enables DRAM namespace sharing between error correction information and data to provide strong chipkill and double-chipkill ECC. Yoon et. al [42] use a pure hardware based approach that adjusts ECC based on memory access granularity prediction; Yoon et. al [41] leverage OS to track memory access granularity to adjust ECC.

Previous works cannot be combined with ABFT because of a semantic gap for error detection and location between ECC and algorithm-based protection. In addition, they target on avoiding data overfetching from main memory, and may interleave data with different vulnerability in the same page with a unified ECC protection. For a data structure protected by ABFT, it may be distributed into different page frames based on its access pattern and hence has various ECC protection for the same data structure. Previous works also make extensive modifications to hardware. These changes make building systems costly.

7. CONCLUSIONS

Resiliency continues to be one of the major design goals for high-end computing systems. ABFT, as a software-based resilience solution, provides protection for some critical application data structures, and effectively reduce or even eliminate costly checkpoint/restart. In this paper, we rethink ABFT with a software-hardware collaborative approach. Given data protection from ABFT, we selectively relax ECC protection for those ABFT protected data structures on main memory to improve performance and system energy efficiency. We introduce an explicitly-managed ECC mechanism to allow tailoring error detection and recovery based on ABFT needs. We classify error handling scenarios in light of ABFT and ECC integration, and establish fault models to investigate the impact of the coordination for system resilience. The main implication of this work is that exposing architecture information to the user and enabling holistic reliability management provides tangible benefits.

Acknowledgments: The paper has been authored by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is partly supported by US National Science Foundation under grants #CNS-1304969, #CCF-1305622, and #OCI-1305624.

8. REFERENCES

[1] Calculating Memory System Power for DDR3, Technical Report TN-41-01. Technical report, Micron Technology, 2007.

[2] OpenSPARC T2 System-On-Chip (SOC) Microarchitecture Specification. Technical report, Sun Microsystems Inc., 2008.

[3] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Future Scaling of Processor-Memory Interfaces. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[4] AMD. BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors, 2007.

[5] A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, M. Schulz, and U. M. Yang. Scaling Algebraic Multigrid Solvers: On the Road to Exascale. In *International Conf. on Competence in HPC*, 2011.

[6] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM Journal of Research and Development*, 2007.

[7] Z. Chen. Algorithm-Based Recovery for Iterative Methods without Checkpointing. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2011.

[8] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2013.

[9] T. Davies and Z. Chen. Correcting Soft Errors Online in LU Factorization. *Symposium on High-Performance Parallel and Distributed Computing*, 2013.

[10] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing. In *International Conference on Supercomputing*, 2011.

[11] M. de Kruijif, S. Nomura, and K. Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[12] T. Dell. A White Paper On The Benefits Of Chipkill-Correct ECC for PC Server Main Memory. Technical report, IBM Microelectronics Division, 1997.

[13] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen. Matrix Multiplication on GPUs with On-Line Fault Tolerance. In *International Symposium on Parallel and Distributed Processing with Applications*, 2011.

[14] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and

J. Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[15] P. Du, P. Luszczek, S. Tomovand, and J. Dongarra. High Performance Dense Linear System Solver with Soft Error Resilience. In *IEEE Cluster*, 2011.

[16] M. Erez, N. Jayasena, T. J. Knight, and W. J. Dally. Fault Tolerance Techniques for the Merrimac Streaming Supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2005.

[17] K. Ferreira, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[18] C.-H. Ho, M. de Kruijif, K. Sankaralingam, B. Rountree, M. Schulz, and B. R. de Supinski. Mechanisms and Evaluation of Cross-Layer Fault-Tolerance for Supercomputing. In *International Conference on Parallel Processing (ICPP)*, 2012.

[19] M. Y. Hsiao. A Class of Optimal Minimum Odd-Weight-Column SECDED Codes. *IBM Journal of Research and Development*, 1970.

[20] A. A. Hwang, I. Stefanovici, and B. Schroeder. Cosmic Rays Do not Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[21] D. Li, J. S. Vetter, and W. Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[22] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[23] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi. System Implications of Memory Reliability in Exascale Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[24] S. Li, D. H. Yoon, K. Chen, J. Zhao, J. H. Ahn, J. B. Brockman, Y. Xie, and N. P. Jouppi. MAGE: Adaptive Granularity and ECC for Resilient and Power Efficient Memory Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[25] X. Li, M. C. Huang, K. Shen, and L. Chu. A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In *USENIX ATC*, 2010.

[26] H. Liu, T. Davies, C. Ding, C. Karlsson, and Z. Chen. Algorithm-Based Recovery for Newton's Method without Checkpointing. In *Workshop on Dependable Par., Distributed and Network-Centric Systems*, 2011.

[27] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: the Programmer's View. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[28] Mcsim: A manycore simulation infrastructure. http://scale.snu.ac.kr/mcsim.

[29] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[30] V. Reddi, A. Settle, D. Connors, and R. Cohn. Pin: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the 2004 workshop on Computer architecture education*, 2004.

[31] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[32] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V.Adve, V. S. Adve, and Y. Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *International Conf. on Dependable Systems and Networks*, 2008.

[33] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.

[34] C. Slayman. Impact of Error Correction Code and Dynamic Memory Reconfiguration on High-Reliability/Low-Cost Server Memory. In *Integrated Reliability Workshop*, 2006.

[35] V. Sridharan and D. Liberty. A Study of DRAM Failures in the Field. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[36] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi. LOT-ECC: Localized and Tiered Reliability Mechanisms for Commodity Memory Systems. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[37] N. J. Wright, W. Pfeiffer, and A. Snavely. Characterizing Parallel Scaling of Scientific Applications using IPM. In *International Conference on HPC*, 2009.

[38] P. Wu, L. Chen, L. Tan, and Z. Chen. Online Soft Error Correction in Cholesky Decomposition. *UC, Riverside, Technical Report UCR-CS-13-002*, 2013.

[39] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen. Fault Tolerant Matrix-Matrix Multiplication: Correcting Soft Errors On-line. In *Workshop on Scalable Algorithms for Large-Scale Systems*, 2011.

[40] D. H. Yoon and M. Erez. Virtualized and Flexible ECC for Main Memory. In *ASPLOS*, 2010.

[41] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive Granularity Memory Systems: A Tradeoff between Storage Effciency and Throughput. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[42] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. The Dynamic Granularity Memory System. In *International Symp. on Computer Architecture*, 2012.