CREECelerator: Composable Hardware Generators for Disaggregated Storage Systems

Bob Zhou, Kyle Kovacs, Vighnesh Iyer, Tan Nguyen
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{bob.linchuan, kylekovacs, vighnesh.iyer, tan.nqd}@berkeley.edu

***Abstract*—Disaggregated datacenter architectures promise efficient resource allocation, ease of maintainability, and fast adoption of new technologies by decoupling compute and storage services. To enable storage disaggregation, hardware accelerators are critical to enable enhanced features, e.g. compression, near the storage device to minimize power, maximize throughput, and improve disk endurance. Whereas most existing accelerators operate in isolation, designing a complete efficient accelerator system requires design space exploration with a hardware generator. This work presents CREECelerator [1], a collection of composable storage accelerator generators written in Chisel [2] with RISC-V Rocket [3] integration.**

I. Introduction

Compression, encryption, and error correction coding (ECC) are common features in storage infrastructures. Data compression can improve disk endurance and throughput; disk encryption is a security requirement in modern datacenters; and parity mitigates bit rot.

Applying these enhancements efficiently introduces computational bottlenecks and increases latency. Dedicated accelerators have been explored to perform these tasks individually. Prior work in hardware accelerated disk compression [4] [5] [6] focused on monolithic approaches that are standalone and OS- and filesystem-invisible. Work on integrating compression with ECC [7] utilized custom algorithms and did not consider hardware acceleration. Existing hardware encryption accelerators either rely on proprietary features [8] or are isolated and application-specific [9].

In this work, we propose CREECelerator: a collection of composable hardware accelerators for compression, encryption, and Reed-Solomon codes (ECC) written in Chisel. The accelerators can be arbitrarily ordered and combined, and can interface with a host system to provide both OS-invisible or filesystem-aware storage acceleration capabilities.

II. CREEC Architecture and Implementation

A. CREECelerator Architecture

The top-level architecture of the CREECelerator consists of a read and write path (Figure 1) which represents two possible combinations of the CREEC blocks. The write path chains the compression → encryption (AES128) → ECC (Reed-Solomon Encoder) blocks, while the read path chains these blocks in the opposite order (and in the opposite modes).

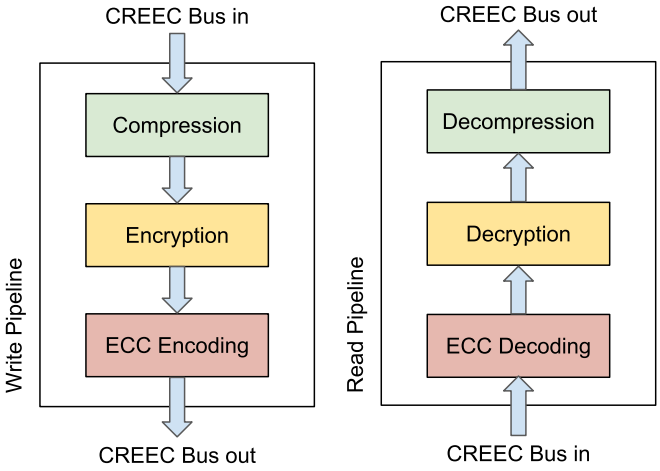

Figure 1: Top-level assembly of the CREECelerator.

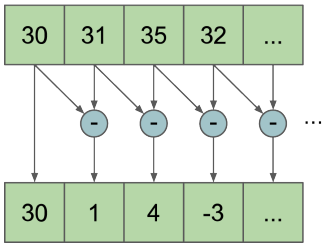

Figure 2: A sequence of bytes can be differential-encoded (or decoded) according to this diagram. The result is an initial value and a series of differences.

B. Compression

The CREEC compressor uses a simple differential + run-length encoding compression scheme. Both differential encoding and run-length encoding have completely reversible analogous operations, meaning the compression is lossless.

1) Differential Coding: This length-preserving transformation is often used as a pre-compression pass over data to prepare it for one or more compression passes. Input sequences are transformed bytewise to an initial value followed by a series of differences. Figure 2 demonstrates the technique.

In addition to potentially reducing the total number of unique symbols required to represent a sequence of bytes,

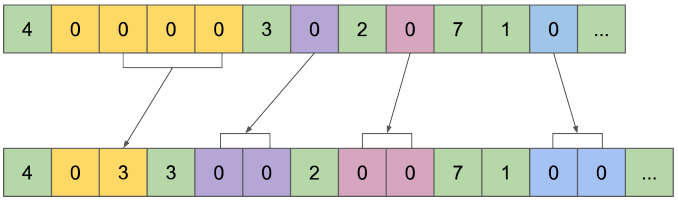

Figure 3: Run-Length encoding. This scheme can occasionally increase the data size. In this diagram, the sequence shrank by 2 bytes from a run of four zeros, but it also grew by 3 bytes due to the three singleton zeros.

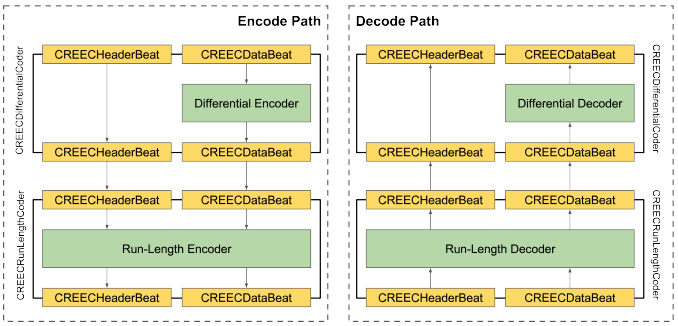

Figure 4: The encode and decode paths of the compressor with CREECBus integration

which can make dictionary or sliding-window compression passes more space efficient, differential encoding turns any sequence of repeated bytes into a sequence of zeros. Since run-length encoding operates on sequences of zeros, differential encoding is necessary to allow run-length encoding to provide good compression.

2) Run-Length Coding: Run-Length encoding compresses "runs" of zeros into a single zero followed by a number of additional zeros. Figure 3 demonstrates the algorithm.

Differential + run-length encoding provides good compression on the specific kinds of data. Any data that has large sequences of repeated values will be compressed very nicely, but random sequences or sequences that repeat at granularities larger than the byte level are not compressible. Furthermore, a bad sequence can sometimes lead to data expansion, rather than compression, as depicted in Figure 3.

In the hardware compression block, differential encoding and decoding is done in chunks according to the size of the data bus. Run-length encoding is done one byte at a time. The two functions are implemented as separate CREEC modules, and the compressor is written as a composition of the two. This allows for more extensibility and flexibility. Figure 4 shows the full layout of the compression section of the CREEC stack. Section II-E explains the CREEC bus in more detail.

C. Encryption

We selected the Advanced Encryption Standard (AES) algorithm for this project for its increasing ubiquity. For simplicity, the block is implemented for the 128-bit key standard and operates in the Electronic Codebook (ECB) mode. While the

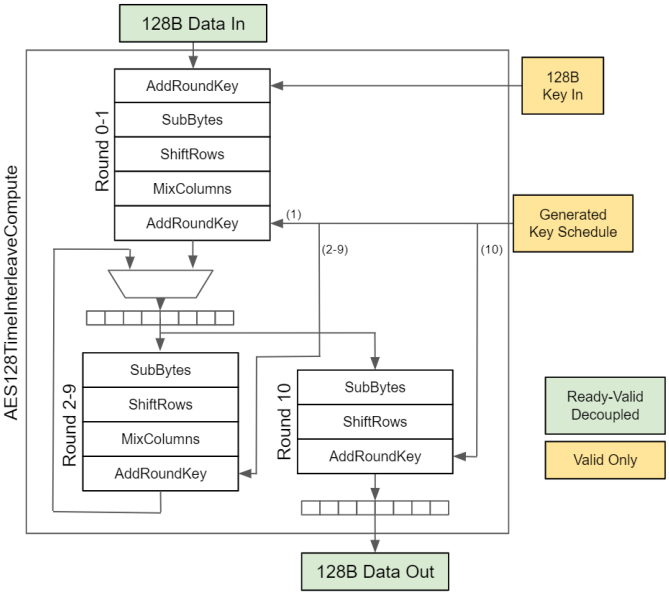

Figure 5: Iterative encryption block diagram

256-bit key mode is more secure, the additional complexity when compared to AES-128 could be easily implemented given this project and so was deemed an irrelevant exercise here. A similar philosophy was adopted toward other operating modes such as cipher block chaining (CBC).

1) Algorithm Description: At a high level, the AES algorithm runs iterative obfuscation using various byte shifts, substitution, and key mixing. These transforms utilize a finite field, lending well to bit-wise operations. Mathematically, the algorithm is easily reversible, so encryption and decryption can be implemented with much shared code, including sharing a key schedule generator.

For a full mathematical description of the algorithm, we refer the reader to the official specification [10].

2) Implementation Details: The encryption, decryption, and key schedule generation blocks were all implemented as iterative hardware modules. Figure 5 shows how the encryption block does this. A primary block handles the major cipher rounds (rounds 2-9). A full cipher block is also added to the initial stage to better match stage timing. A final separate output stage sits in parallel to the major cipher block.

Decryption and key generation is implemented in a similar fashion, so we will skip the details here. Unlike encryption, decryption requires all round keys to be generated and valid at the start of operation. This is a major reason to share key schedules between the blocks.

The iterative implementation was chosen for its simplicity. It provides great reuse of blocks and is competitive with pipelined approaches, since multiple of this implementation can be deployed in parallel for time-interleaved execution. For fewer than 10 beats in flight, this implementation saves on area; for greater than 10 beats, this provides little area benefit, but potential design simplifications.

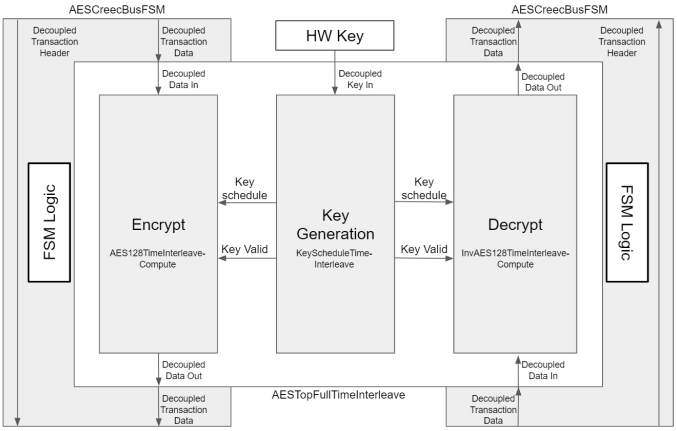

Figure 6: AES Top Level diagram with CREECBus Integration

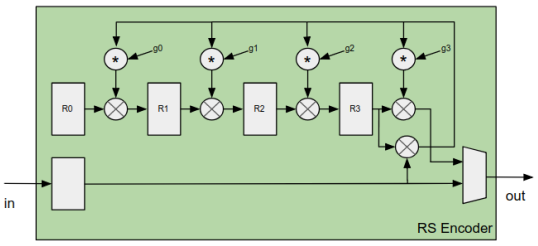

Figure 7: Reed-Solomon Encoder datapath

Figure 6 demonstrates AES top-level integration, including the interfaces with the CREEC bus write and read paths. This design is agnostic to how the key is provided, as the key generator can update during runtime. For our demonstration, our top level has a supplemental Chisel module that provides a built-in hardware key and reset logic.

D. Error-Correcting Code

We adopt the popular Reed-Solomon coding (RS) as the ECC scheme for our CREEC stack. Note that RS is a block-level (symbol) parity scheme.

1) RS Encoder implementation: Figure 7 provides an overview of the RS Encoder datapath. It consists of a linear-feedback shift-register. The number of such registers equals to number of parity symbols, hence each register computes a parity symbol. Each multiplier (in finite field) takes one generator coefficient and the feedback wire as operators. The generator coefficients are generated statically and passed to the block as parameters.

2) RS Decoder implementation: Figure 8 provides an overview of the RS Decoder datapath. The decoding process contains four stages: Syndrome Computation (checks if the input sequence has error), Key Equation Solver (determines the error and magnitude polynomials), Chien Search (locates the erroneous symbols), and Error Correction (fixes the symbols). The decoder is expected to obtain the original, un-encoded symbol sequence.

The Polynomial evaluation logic is used in Syndrome Computation, Chien Search, and Error Correction. To promote

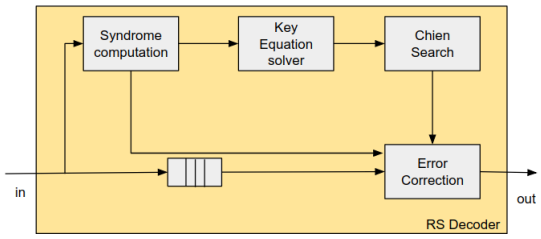

Figure 8: Reed-Solomon Decoder datapath

reusability, we design a polynomial evaluation hardware block of 1D systolic array to effectively compute a polynomial at multiple evaluation points in parallel.

The Key Equation Solver uses the Modified Euclidean Algorithm (M.E.A) to perform polynomial division. Since division is expensive in finite field, we use the cross multiplication technique to reduce the hardware complexity.

3) CREECBus-wrapper Modules: Similar to other CREEC blocks, the RS Encoder and Decoder are also wrapped inside top-level modules with CREECBus interface. This enables the blocks to communicate with the rest of the stack.

E. Interconnect

The `CREECBus` is a simple unidirectional bus with decoupled (ready/valid) header and data channels. All the CREEC blocks use a custom `CREECBus` to interface with each other. Each block has an input and output `CREECBus` as its top-level IO. `CREECBus` was designed to easily bridge to a memory mapped bus (such as AXI4 or TileLink) or a streaming bus (like AXI4-Stream). The header channel carries a payload with a transaction `id` and `length`. The data channel carries `data` with an `id` corresponding to a header beat. This enables write interleaving on the `CREECBus` if the master supports it.

III. Design Methodology

A. Verification

1) Transaction-Level Modeling (TLM): In the pursuit of a unified SW testing and RTL stimulus/checking infrastructure, we created a transaction-level modeling framework in Scala. The abstract notion of a `Transaction` is a chunk of data and control that a block can process or produce.

We defined 2 levels of `Transactions` specific to the `CREECBus`. High-level transactions are plain sequences of bytes and are agnostic to the bus parameterization (data width, maximum burst length), while `CREECHeaderBeats` and `CREECDataBeatss` are low-level and are specialized for a concrete bus instance. TLM enables a unified stimulus to drive both hardware RTL blocks and software models. High-level transactions can be converted to low-level ones given a specific bus parameterization, and vice versa.

2) Software Modeling: Each hardware CREEC block has a corresponding software model which describes the behavior of the block. Software models serve as a golden reference that the RTL can be compared against. The models are checked against hand-verified stimulus and expected outputs. All of the CREEC software models operate on high-level transactions.

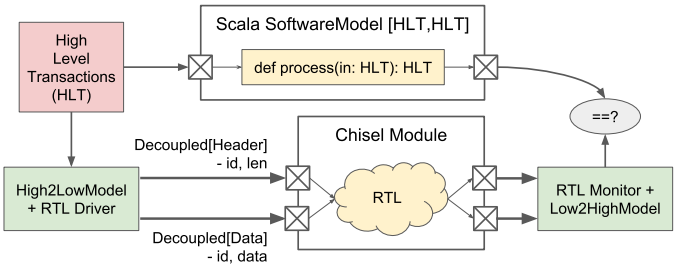

Figure 9: Modeling and working with the CREECBus in software and RTL

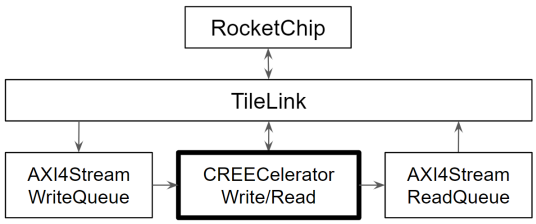

Figure 10: CREECBus integration with Rocket-chip

3) RTL Simulation: The same high-level stimulus that we use to test the software model of a block can be used to drive the RTL implementation of that same block. The software model can also be used to generate golden output for new random stimulus. We use the testers2 [11] Chisel library to create a RTL-level `CREECBus` driver and monitor for low-level transactions.

B. Rocket Chip Integration

Rocket Chip [3] is a popular SoC generator built around the Rocket RISC-V processor. Figure 10 demonstrates how we integrate `CREECelerator` with Rocket using their project template [12]. Rocket chip's TileLink crossbar allows for simple integration of AXI4-Stream devices. We first select either the write-path or read-path CREECelerator to integrate. We create write and read queues wrapped in AXI4-Stream to feed data into CREECelerator. Configuration information for the accelerator is set using TileLink MMIO.

This integrated design is tested by compiling an RTL simulator using Verilator. This executable can then consume C code compiled for RISC-V.

IV. Results and Discussion

To judge performance, we stimulated each block with a 512-byte test vector based on ASCII text from Plato's *Republic*. We report cycle performance numbers in Table I. Our full pipeline tests place compression first, then encryption, follows by ECC.

For synthesis, we targeted a Xilinx VC707 FPGA using HAMMER, a synthesis and place & route tool in active development at UC Berkeley [13]. We report our area results in Table II and timing results in Table III.

	Full Pipeline		**Standalone Blocks**	
	Encode	Decode	Encode	Decode
Compression	1764	1707	1764	1707
Encryption	474	474	460	460
ECC	1258	*7022	1220	2308
Total	3496	9203	3444	4475

Table I: Cycle counts for each top level block in the full pipeline are shown on the left. Data was passed through each block in order. The right column data comes from each block running given the initial sample data, outside the influence of the rest of the pipeline. **The full pipeline test was run with random noise introduced in order to measure average performance of the ECC block. This number represents an average over ten trials.*

Module	**LUTs**	**FFs**
Compressor	29161	21403
Decompressor	29035	21431
ECCEncoderTop	160	362
ECCDecoderTop	3622	1374
AES128TimeInterleaveCompute	3238	274
InvAES128TimeInterleaveCompute	3453	263
KeyScheduleTimeInterleave	368	1421
AESCREECBusFSM	29	321
Total	69800	48757

Table II: Reported Area usage for unoptimized synthesis

1) Compression: Due to the fact that the CREEC bus requires headers to be received before data, CREEC blocks cannot send any data until they know all the appropriate header information for a given transaction. In the case of the compression block, the length of the output is not known until all the data has been received and processed. This means that the compressor must wait for all data to arrive before sending anything. Additionally, since run-length encoding operates on a byte-by-byte basis, it takes many cycles to process the data.

In terms of area, the run-length encoder and decoder use large vectors of registers to store all the data before it is passed on. This is because of the way the CREEC bus is structured, as discussed earlier.

2) Encryption: The performance results are as expected. We expect the iterative compute block to take 10 cycles per 128-byte beat. 10 initial cycles are expected from key generation. In the performance test, we also account for 4 extra cycles per beat from the extra state machines to interface with CREECBus, which could be reduced.

Area usage was kept reasonable thanks to the iterative implementation. Sharing the key generation module was useful,

Top Module	**Critical Path (ns)**
Compressor	3.51
Decompressor	5.13
ECCEncoderTop	3.04
ECCDecoderTop	4.72
AESTopCREECBus	4.35

Table III: Critical Paths for each major block (maximum delay).

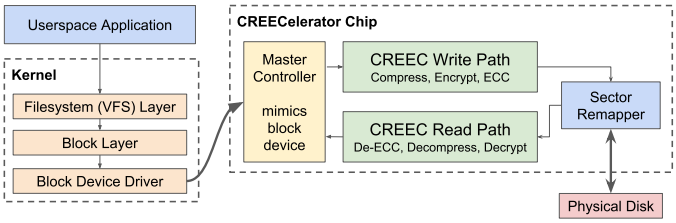

Figure 11: CREEC blocks as an OS-invisible accelerator

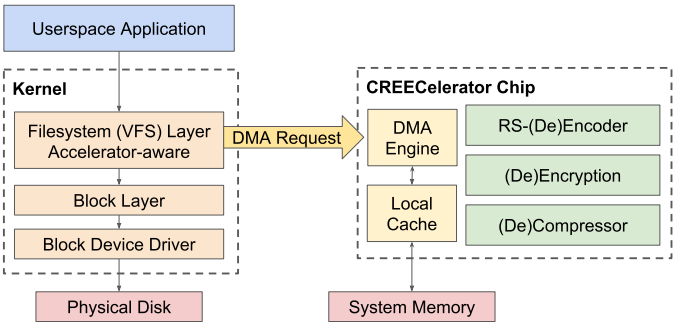

Figure 12: CREEC blocks as a filesystem-aware accelerator

since its size is roughly half of either cipher block, largely from key storage. The primary area contribution stems from the repeated usage of SBox lookup tables. Each SubByte stage had its own lookup table, resulting in roughly 100 table instances. A potential design exploration would involve sharing SBox tables between blocks.

The critical path of this block is expectedly the path from the registered key schedule through an entire decrypt stage. It is possible to break this path into multiple stages, but the total computation latency would not change.

3) Error-Correcting Code: The encoder is relatively inexpensive since it only requires an LSFR, hence justifies the small consumption of LUTs and FFs. On the other hand, the decoder is fairly complex as it involves many computing stages, In addition, an inversion operation is required and it lies on the critical path. At the moment, the inversion is implemented using a look-up table.

V. Future work

A. Block Improvements

The implemented differential + run-length compression block has design inefficiencies that could be explored and improved. Future work could implement a more advanced algorithm, such as Snappy [14] or Zstandard [15].

The AES block could explore Sbox implementations, which is under active research in the community [9]. Additional exploration is also needed in other AES modes (192- and 256-bit keys, CBC and CFB, etc.) and time-interleave performance.

The Reed-Solomon decoder could be pipelined using a systolic array architecture to improve the throughput and the critical path as shown in prior work [16].

B. System-Level Testing

We had planned to perform system-level testing where a virtual block device is used by unmodified userspace applications and would proxy the `bio structs` to the CREEC pipeline running as a RTL simulation. A sector remapper is used to pack compressed, encrypted, and parity-added sectors (which usually are not 512 bytes long) into regular disk sectors, which is the minimum granularity of access. Exploring sector remapping would help balance the metrics of memory usage and disk latency/throughput.

C. CREEC Applications

The CREEC blocks are designed to be reusable to implement the following applications:

- An OS-invisible storage accelerator which sits in between the block device visible to the OS and the physical disk controller (Figure 11). This pipeline can improve SSD endurance with the compression functionality, add within-sector redundancy with the ECC functionality, or provide transparent full disk encryption facilities. Our top-level architecture (Figure 1) was designed to function in this manner.
- An accelerator for filesystem-level implementations of compression/encryption/ECC (Figure 12). For instance, ZFS supports all three capabilities to some extent and dedicated hardware accelerators can provide a considerable performance boost, while also save CPU cycles.

References

[1] "Creecelerator," https://github.com/ucberkeley-ee290c/fa18-smartnic, accessed: 2018-12-4.

[2] "Chisel 3," https://github.com/freechipsproject/chisel3, accessed: 2018-12-4.

[3] "Rocket chip generator," https://github.com/freechipsproject/rocket-chip, accessed: 2018-12-4.

[4] K. F. A. S. Lee, J. Park and J. Kim, "Improving performance and lifetime of solid-state drives using hardware-accelerated compression," *IEEE Transactions on Consumer Electronics*, vol. 57, no. 4, pp. 1732–1739, 2011.

[5] *Pensieve: A Machine Learning Assisted SSD Layer for Extending the Lifetime*. IEEE International Conference on Computer Design, 2018.

[6] M. M. M. D. F. A. B. Yannis Klonatos, Thanos Makatos, "Transparent online storage compression at the block-level," *ACM Transactions on Storage*, vol. 8, no. 5, 2012.

[7] *Adding aggressive error correction to a high-performance compressing flash file system*. Proceedings of the seventh ACM international conference on Embedded software, 2009.

[8] "Introduction to intel aes-ni and intel secure key instructions," https://software.intel.com/en-us/node/256280, accessed: 2018-12-4.

[9] *An ASIC implementation of low area AES encryption core for wireless networks*. Communications Management and Telecommunications (ComManTel) 2015 International Conference, 2015.

[10] N. I. of Standards and Technology, "Announcing the advanced encryption standard," Nov 2001.

[11] "Chisel testers 2," https://github.com/ucb-bar/chisel-testers2, accessed: 2018-12-4.

[12] "Risc-v project template," https://github.com/ucb-bar/project-template, accessed: 2018-12-4.

[13] "Hammer: Highly agile masks made effortlessly from rtl," https://github.com/ucb-bar/hammer, accessed: 2018-12-4.

[14] "Snappy," https://google.github.io/snappy/, accessed: 2018-12-4.

[15] "Zstandard," https://facebook.github.io/zstd/, accessed: 2018-12-4.

[16] R. P. Brent and H. T. Kung, "Systolic vlsi arrays for polynomial gcd computation," *IEEE Transactions on Computers*, vol. C-33, no. 8, pp. 731–736, Aug 1984.