Babol: A Software-Defined NAND Flash Controller

Kibin Park* Alberto Lerner† Sangjin Lee† Philippe Bonnet‡
Yong Ho Song§ Philippe Cudré-Mauroux† Jungwook Choi*

†*University of Fribourg, Switzerland* **Hanyang University, South Korea*
‡*University of Copenhagen, Denmark* §*Samsung Electronics*

***Abstract*—NAND Flash Storage Controllers are a crucial component of Solid State Drives (SSDs). They provide an abstraction of Flash packages to the SSD firmware by translating high-level operations, such as a Page Program or a Block Erase, into low-level signals. In theory, the Open NAND Flash Interface (ONFI) specification standardizes this interface. In practice, however, every package supports optimized versions of the standard operations as well as non-standard operations. Writing a controller that exploits these optimizations is the only way to obtain competitive performance, but it makes for a highly intricate, error-prone, and non-portable controller development process. Compounding the issue is the fact that new generations of Flash packages are produced yearly, and non-standard optimization techniques are often presented in the literature. Modifying rigid hardware controllers to support these advancements is extremely challenging, making it difficult to rapidly prototype new SSDs and exploit the full potential of Flash memory.**

To address this, we propose Babol, a software-defined Flash controller architecture that provides generic hardware building blocks that can be flexibly combined via software to express complex, package-optimized Flash operations. We implemented two flavors of Babol in an FPGA setting and experimented with several commercial off-the-shelf Flash packages. Our results show that the flexibility that Babol brings far outweighs the marginal amounts of performance and area it requires. We open source our controller, including its unique software programming environment, which we believe can make SSD controller development more productive for seasoned SSD Architects and make prototyping accessible for newcomers who want to join the field.

***Index Terms*—Storage, SSD, Flash Controller.**

I. Introduction

NAND Flash is the *de facto* storage medium for data-intensive applications [3], [6], [19], [28], [53], [58] and fast storage [16], [26], [30], [36]. This is due in no small part to the high number of variations into which Flash memory can be packaged. For instance, the Flash Arrays, which are the fundamental building blocks of Flash storage, can be made of cells that hold one, two, three, or even four bits of information. These arrays can be built into planar or 3D structures providing a variety of address space sizes, and several of those arrays can be present in what is called a multi-plane package. These geometry details, which are nicely explained elsewhere [2], [40], [41], unlock many possibilities on how an SSD can optimize the performance, longevity, and energy consumption of its underlying Flash storage [13], [15], [38], [49].

Ideally, the norms laid out by the ONFI standard [45] make Flash packages interchangeable regardless of their implementation details. The standard specifies the number and voltage of pins a compliant Flash package must have and outlines how different data transfer speeds and modes (synchronous and asynchronous) can be achieved via these pins. From an electrical interoperability perspective, the standard can be considered successful.

Unfortunately, ONFI is less effective when it comes to operations. The issue is not a lack of standard operations; there exists a standard `READ`, `PROGRAM`, and `ERASE`. They each entail using the standard pins to produce a unique waveform, and an SSD architect could develop a controller that issues these waveforms against various packages. However, the standard waveforms lack important optimizations since they abstract away relevant internal Flash Array details.

Take a `READ` operation for instance. The standard indeed supports some variations of this operation, such as a `PAGE READ`, a `READ CACHE` (to interleave reads), and a `CHANGE READ COLUMN` (for partial reads). However, there exist many variations offered by manufacturers on a per-package basis such as `PSEUDO SLC READ`s (to increase the longevity of the device) [14]. Other variations are offered by the academic literature, such as `PARTIAL READ` [20], [33], `READ`s with bounded latency [32], and `READ RETRY` [34], [48]. Similarly, optimizations also exist for a `PROGRAM` [10], [52] as well as an `ERASE` operation [23], [54], to cite a few. The variations are often faster or provide additional interesting functionality (e.g., possibility to suspend long operations).

With such a wide range of alternative optimizations, an SSD Architect often foregoes portability in the name of performance. They develop specialized hardware controllers in what is an extraordinarily laborious and error-prone process, and all the work goes into what is essentially a *one-off* controller. The controller logic may need to be revisited if any changes occur, such as those a new package may bring.

Interestingly, other SSD components, such as the Host-Interface Controller (HIC), the Flash Translation Layer (FTL), and Error Correction Coding (ECC), shown in Figure 1(Left), all have flexible development tools. The HIC can be implemented using available NVMe frameworks [18], [51]; there are numerous options for FTLs [8], [39]; and hardware implementations of ECC are also accessible [7], [12].

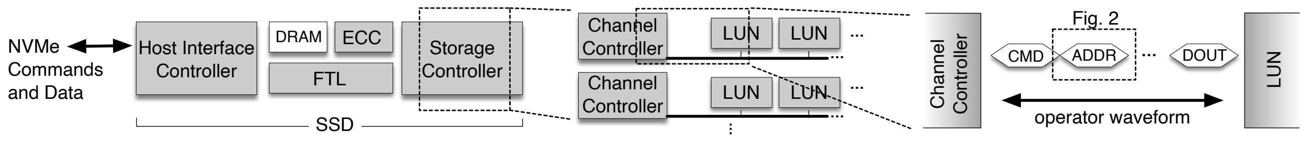

Fig. 1. (Left) A traditional SSD architecture where the HIC interacts with the host, the FTL controls page mapping and management tasks, and the Storage Controller contains Flash packages. (Center) The Storage Controller organizes Flash LUNs into channels. (Right) The waveform summary used in the communication between a channel controller and a package. We show a fragment of a waveform in more detail in Figure 2.

We argue that the only component missing a flexible architecture is the Storage Controller. In this paper, we seize this opportunity and propose a new approach to develop Flash controllers that embrace the variety of Flash packages. We call the resulting controller BABOL.

BABOL's most notable insight is to leverage the significant delays of NAND Flash memory. Compared to DRAM, in which the packages can respond to an operation request in tens of nanoseconds, NAND Flash may require tens of microseconds to perform an operation. While a given operation is being executed in a portion of the Storage Controller, another area may be preparing to issue the next operation. Thanks to the delays, the latter controller portion may be software-supported. To the best of our knowledge, BABOL is the first controller to perform this decoupling while preventing any significant performance degradation.

We describe BABOL's architecture and report on a reference implementation. We discuss how to develop operations in this environment and show that BABOL does not require significantly more area than a traditional controller. We believe that BABOL can become the go-to framework for SSD Architects to build Storage Controllers for existing and potentially future Flash packages. The controller is fully open-sourced.

The structure of the paper follows its contributions:

- We characterize the difficulty of architecting a hardware-based storage controller and explain the need for a software-based one in Section II.
- In Section III, we present BABOL's architecture and highlight its differences compared to classic controllers.
- In Section IV, we present details of BABOL's hardware programmability support.
- In Section V, we discuss how the software portion of BABOL controls and commands the programmable hardware described in the previous section.
- We experiment with BABOL using different actual Flash packages and discuss the results in Section VI.
- We contrast the techniques we introduced in BABOL with other controllers and other software-hardware co-designed systems that have a similar structure in Section VII.

Lastly, we present our conclusions in Section VIII.

II. Background and Motivation

SSD Architecture. As briefly mentioned above, a traditional SSD architecture is usually comprised of the four components shown in Figure 1(Left). The Host Interface Controller is responsible for all communications with the host. In modern SSDs, this controller implements the NVMe protocol [44] and uses a DRAM data buffer to stage the data coming in and out of the device. The HIC requests services from the Flash Translation Layer, a catch-all component responsible for, among other things, locating and allocating the physical addresses of pages and requesting page- or block-level operations that, in turn, are implemented by a Storage Controller component. Another important component is the one dedicated to Error Correction Coding. It addresses the fact that Flash packages are a faulty media [5]. ECC techniques are necessary to identify and fix some of the errors [42].

A conventional Storage Controller, as depicted in Figure 1(Center), exports a continuous Flash memory address range to the FTL. Internally, however, it bundles relatively small and slow Flash packages into a structure called *channel*. The goal is to present a faster interface than each individual package can achieve independently by resorting to parallelism [9]. A typical Flash package carries one or more Logical Units (LUNs), each of which is capable of performing an operation independently. A channel gathers a small number of LUNs together, typically 2 to 16, via a common bus. Because it is shared, the Storage Controller must schedule the bus usage whenever it wishes to communicate with one of its attached LUNs. In other words, while one LUN is busy with, for instance, fetching a page's data as part of a `READ` operation, the Storage Controller can ask another LUN in the same channel to start preparing for, say, a `READ` operation against a different address that the latter holds. This *interleaving* of operations being sent to LUNs through the shared channel is quite common. We will discuss this interleaving in more detail shortly. For now, let us concentrate on one operation at a time.

ONFI Operations. The ONFI standard dictates how to interact with a LUN to request each particular operation. Curiously, this communication involves several steps. Figure 1(Right) depicts a sequence of such steps, each of which is represented by an elongated hexagon. An operation's first step is to tell the LUN the desired operation ID.[1] This step is called a *command latch*. A *latch* is the term used to refer to the action of asking the LUN—more precisely, its Flash Array controller—to retain a piece of information for further use. The next step is usually to establish the operation's target address. This step is called an *address latch*. Figure 2 depicts the waveform fragment involved in such latching. Note that several wait times are

[1]The standard refers to operations as commands, but in this paper, we try to distinguish between NVMe commands and ONFI operations whenever possible. We still use terms such as *command latches* that refer to an ONFI operation step, but when we do, we hope that it is clear from the context that we are discussing ONFI aspects rather than NVMe.

involved in constructing the waveform correctly. These are called *timing parameters* and are also specified by ONFI.

In addition to command and address latches, the ONFI standard defines two other types of steps: data in and data out. As the names imply, they each define the waveform fragment that requests a LUN to write or read the data in its Page Register, respectively. ONFI calls each of these operation steps *Basic Timing Cycles* (BTCs). BTCs can be considered a form of vocabulary from which sophisticated sentences (operations) could be built. We will show throughout this paper that the BTC idea is interesting but the vocabulary is incomplete. SSDs often resort to nonstandard operations (e.g., see Algorithm 3 below) for performance reasons that are difficult to express with only standard BTCs. Let us resume our discussion about interleaving steps.

Operation Interleaving and Transaction Scheduling. The multi-step approach to performing an operation is necessary because ONFI uses a limited number of pins for cost and area savings reasons. In other words, there are not enough pins to communicate, for instance, an operation ID and a target address simultaneously. Moreover, the multi-step approach is necessary because there are also wait times across some steps. With rare exceptions, a LUN requires pauses in the interaction with the Storage Controller while it performs actual work. For instance, in a `READ` operation, the LUN needs time to fetch the desired page from the Array and put it in the Page Register, whence it can be transmitted.

Interestingly, while one operation in a given LUN is experiencing a timed wait, thus, relinquishing the channel, another operation targeted at a different LUN can proceed. The Channel Controller is responsible for arbitrating the channel use when several operations are concurrently active. Figure 3 shows how the steps of two `READ` operations can be interleaved. As the figure shows, a scheduler inside the controller groups related steps into atomic units called *transactions*. For instance, a command and address latch form a transaction. A transaction is called this way because it is never descheduled before it completes.

Naturally, deciding on which ongoing operation should issue its next transaction and use the channel is delegated to a scheduler. There are many plausible objectives for such a scheduler. An example could be to maximize the channel throughput [43]. Another one could be to minimize the latency of ongoing NVMe commands [24]. Designing a competitive scheduler is one of the main tasks of the SSD Architect.

Synchronous Channel Controllers. A typical channel controller is depicted in Figure 4 [50]. This type of hardware-based controller is fast because it has dedicated area that implements the ONFI operations. In the figure, they are called `Operation_i` and can issue all the different waveforms necessary to drive a LUN. Each operation is usually implemented as an individual Finite-State Machine (FSM) in the operation module, as the figure also shows. Selecting how to implement each of these FSMs is also the responsibility of the SSD Architect.

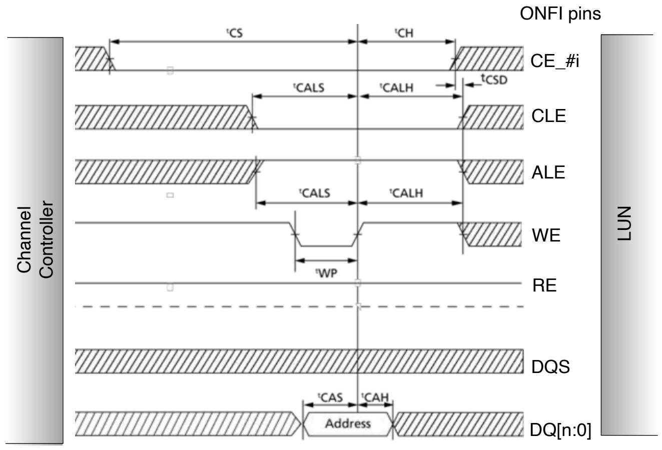

Fig. 2. An address latch step consists of a waveform fragment that the Channel Controller uses to request an operation from a LUN. The waveform uses the pins defined by ONFI, shown here close to the LUN. This fragment is the same for a `READ` or a `PROGRAM` operation.

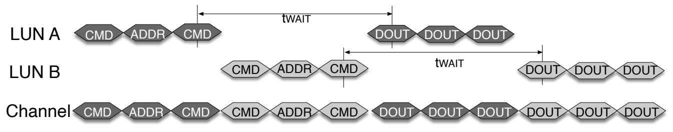

Fig. 3. Two independent operations being interleaved. (Top, Center) Timed view from each operation's perspective. (Bottom) Timed view from the channel perspective.

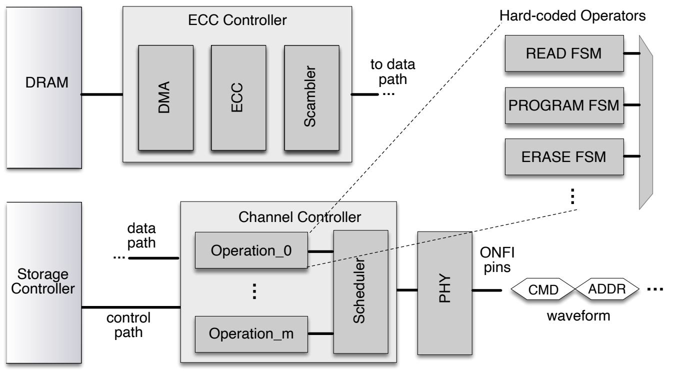

Fig. 4. A typical hardware-based Channel Controller. Because the operations waveforms contain both data and instructions, the control and data paths of the SSD merge at the Channel Controller.

There are as many operation modules as LUNs in the channel. This way, the controller can manage all the LUNs in the channel concurrently. Once the controller receives an operation request against a LUN, the corresponding operation module is configured to execute that operation. Note, however, that the operation modules do not proceed if they do not have access to the channel. This is the reason such a controller design is deemed synchronous.

A Scheduler module manages access to the channel, i.e., it acts as an Arbiter. Whenever the channel is free, the Scheduler grants the latter's use to one of the operation modules. The winning operation module then produces however many transactions it can, which may involve DMA-ing data from DRAM

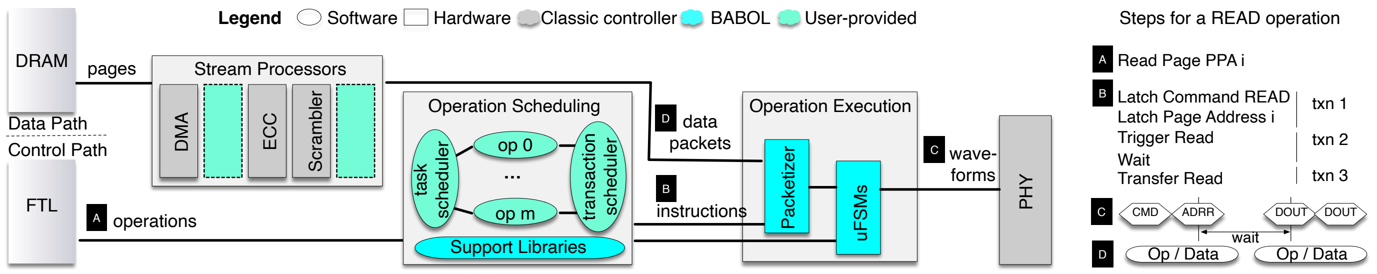

Fig. 5. (Left) The BABOL controller comprises three components: stream processors, operation scheduling, and operation execution. Execution has strict time constraints and is implemented in hardware. The implementation, however, does not use hard-coded waveforms. It allows programmatically building them through μFSMs, which are software-configurable waveform segments emitters. Operations such as `READ`, `PROGRAM`, and `ERASE` are written in software and, with the help of the schedulers, drive the μFSM. (Right) Examples of steps an operation goes through as it is implemented in BABOL.

and generating parity information. When the operation releases the channel, for instance during a mandatory wait time, the Scheduler takes over, and the cycle repeats.

Discussion. This type of scheduler may sound intuitive, but a naive implementation creates several drawbacks. Consider how the Scheduler in Figure 4 selects the next operation/transaction to be executed and dispatches it to the LUN. Because a channel may be fast, this scheduler needs to be built in hardware and should react to the channel vacancies promptly. Developing this arbiter in hardware is less than flexible because it forces the SSD Architect to stipulate optimization criteria that will stay with the SSD until it is decommissioned. As we mentioned above, different workloads may require different criteria [24], [43].

Another drawback is that any active operation can be scheduled next at any time. Once again, a naive implementation must, therefore, generate the next transaction—its waveform segment—-promptly when selected, which indicates a somewhat time-constrained hardware implementation. Developing and debugging an artifact to issue such a precise waveform is laborious. One may argue that this difficulty is immaterial, given that once a `READ` FSM exists, it can be used by any Flash package that adheres to ONFI. The sad fact about ONFI, as mentioned above, is that manufacturers invariably provide proprietary advanced commands.

III. BABOL ARCHITECTURE

We propose a new architecture that can be as responsive as the synchronous one described above but that has none of its disadvantages. The architecture, shown in Figure 5, is based on two new principles.

Separation of Scheduling and Executing of a Waveform. Unlike the architecture depicted in Figure 4, BABOL does not decide on which waveform or transaction thereof to issue next as a response to the channel becoming available.[2] *Instead, a description of the desired segment is produced prior to the opportunity to execute it.* This separation is reflected by the existence of two distinct modules, as shown in Figure 5. The module that describes a segment to be executed in the future is called `Operation Scheduling`. The module that produces that segment once the execution is possible is called `Operation Execution`. We call this architecture *asynchronous* because it separates the description of what a next segment to issue should be from its actual execution.

[2]For simplicity, we refer to both a waveform and a transaction as a segment.

To make the communication between these modules possible, BABOL creates an abstraction to encapsulates a segment's description. The abstraction is akin to a sort of waveform instruction set. Its instructions are amenable to queuing.

There are two flavors of instructions, as Figure 5 implies. One describes the source or the destination of data being moved into and out of the Flash packages, and the other one describes the segment itself. BABOL implements one hardware unit for each kind of instruction: the μ`FSM` unit handles control and the `Packetizer` unit handles data. We discuss these in more detail in Section IV.

Software-based Programming Model. The second aspect that our new architecture challenges is implementing the entire controller in hardware. BABOL implements operation scheduling entirely in software. This is possible because of a combination of factors. As mentioned above, LUNs are often busy performing internal data movements (to/from the Array and Page Register) that can take tens of microseconds. While a single LUN is busy in a given operation, there is enough time to schedule the following operation in software. Moreover, several LUNs share a channel, which is unavailable during data transfers between the controller and LUN. Similarly, while a data transfer is ongoing, there is enough time to decide in software on the next task to give a particular LUN.

Thanks to these two new principles, BABOL allows an SSD Architect to encode standard and nonstandard operations easily. It also allows them to define and implement different scheduling strategies. The challenge in designing BABOL was determining which abstractions to expose or hide from the SSD Architect. We discuss these abstractions in more detail in Section V.

IV. PROGRAMMABLE HARDWARE

The hardware components responsible for communicating with Flash packages, the μFSMs, sit at the bottom of the BABOL architecture. We have shown in Section II that the

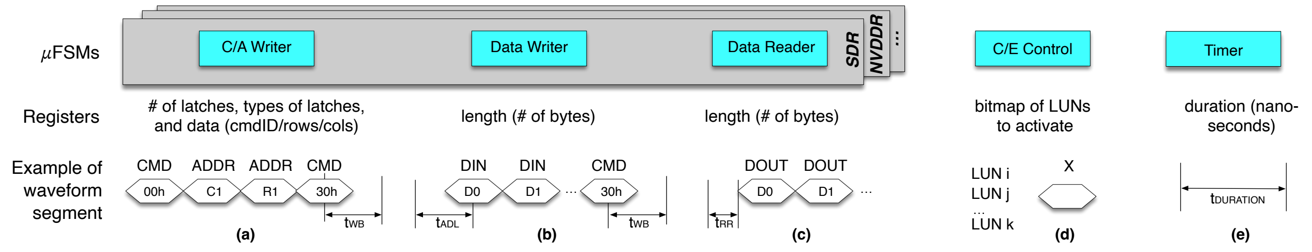

Fig. 6. BABOL's μFSMs and their parameterization. (Top) The leftmost three μFSMs come from the ONFI standard and produce waveform segments useful for issuing command preambles, data writes, and data reads, respectively. Each of these μFSMs needs to be re-implemented for different data modes (e.g., SDR, NVDDR, etc.), but their interface is the same across all modes. The rightmost two μFSM can be considered punctuation (`Timer`) and a modifier (`C/E Control`), but the expressiveness they bring is crucial for programmability. (Bottom) Examples of waveforms that each μFSM can emit.

basic dialog unit between a controller and the Flash packages is what the ONFI standard calls a Basic Timing Cycle (BTC) (cf. Figure 2). Simply put, a BTC is a waveform fragment that establishes one piece of information (e.g., what command to execute, or what address to target, etc.) between a controller and a package. Expressing a full command requires a concatenation of BTCs in a pre-established order that may be unique to each Flash package.

BABOL expands the notion of the BTC by replacing them with μFSMs, a more powerful way to generate waveform segments. Every μFSM is parameterized and can issue many variations of the waveform segment. Some μFSMs are a combination of more than one ONFI BTC, while others find no similar BTC in the standard. *The idea of parameterizing a μFSM may sound simple, but ultimately, describing segments as patterns rather than constant waveforms is what gives our scheme the expressive power to encode basic and advanced operations we found in the literature.* Putting it differently, BABOL's μFSMs are an instruction set to generate ONFI-like waveforms.

We describe the μFSMs next and will comment on how to program them, including showing examples, when we discuss BABOL's software environment.

A. An Expressive Set of μFSMs

The list of μFSMs is depicted in Figure 6. We discuss each of them in turn, emphasizing how they can be parameterized to generate a wide variation of waveform segments.

Command/Address Writer. This μFSM produces the waveform segment containing a command and possibly an address that the command would target. The `C/A Writer`, for short, can be parameterized via three operands: the number of latches ($length$) one wishes to issue, a vector with $length$ elements with the type of each latch (e.g., command or address latch), and a vector of $length$ latch values. Figure 6(a) shows an example of a command with an address width that requires only two latches.

Data Writer. This μFSM emits a waveform segment and its effect is to transfer data into the LUN's Page Register. To obtain the data, the `Data Writer` works closely with the `Packetizer` (see Figure 5), a specialized DMA unit that can read data from the DRAM area of the SSD and deliver it in packets of the same width as a package's `DQ` bus, the ONFI pins dedicated to data transfer. The `Data Writer` is programmed in tandem with the `Packetizer`: The former takes the number of bytes it needs to request, and the latter takes the address from which these bytes would be read.

Once again, as simple as it may sound, the `Data Writer` frees the SSD Architect from all the details of the data transfer, such as driving a strobe pulse (`DQS` in some ONFI Data Interface modes) at a required period to correctly time the data transfer. It also shields the SSD Architect from changes, such as bus width, when they want to use swap packages on an existing controller. Figure 6(b) shows an example of a waveform segment generated by the `Data Writer`.

Data Reader. The `Data Reader` is functionally the inverse of the Data Writer. It interacts similarly with the `Packetizer` to deliver data read from the Page Register into DRAM. It is also responsible for driving `DQS` at the desired period to synchronize the transfer. Figure 6(c) shows an example of a waveform segment generated by this μFSM.

Chip Control. This μFSM changes how other μFSM emit theirs. The `Chip Control` directs a waveform segment to a set of designated "chips" (LUN). This μFSM takes as an argument a bitmap with one bit per package in the channel. Figure 6(d) shows the result of modifying an arbitrary μFSM ('X') via a `Chip Control`.

In practice, the `Chip Control` can be used to *gang schedule* a particular operation or a part thereof. For instance, in RAIL, the authors suggest that data can be replicated inside an SSD so that, whenever read, latency variance could be diminished by trying reads from several different positions [32]. If these positions are within the same channel, these reads and writes could be gang-scheduled using the Chip Enable pin.

Timer. This μFSM produces a pause of at least a given amount of time, according to the *duration* parameter. Figure 6(e) shows the result of introducing a `Timer` into an operation. The `Timer` μFSM is helpful in situations where the SSD Architect needs to give the Flash Array controller a chance to work on the requests that an operation has done. For example, ONFI supports an operation called `SET FEATURE` that can modify several behaviors inside the Flash Array, such as the voltage at which the Flash Array reads Flash cells. This change

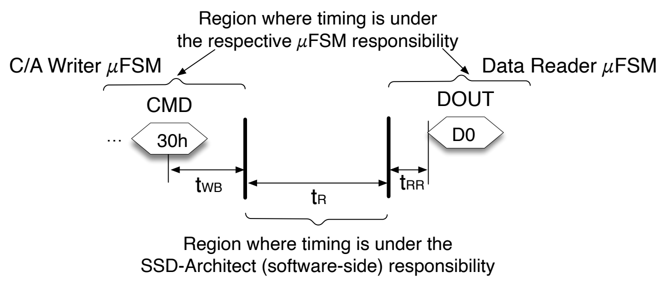
Region where timing is under
the respective μFSM responsibility
C/A Writer μFSM
Data Reader μFSM
CMD
DOUT
30h
D0
tWB
tR
tRR
Region where timing is under the
SSD-Architect (software-side) responsibility

Fig. 7. A portion of a `READ` operation between the end of a `C/A Writer`-issued segment and the start of a `Data Reader`-issued one. The timing within a waveform segment is the responsibility of the μFSM emitting it. In contrast, the timing between two segments should be controlled by the operation logic written by an SSD Architect.

is fundamental in `READ`s with retries. The waveform that initiates a `SET FEATURE` operation has to pause for t_{ADL} nanoseconds before the waveform can express the new feature value. The t_{ADL} delay is called by ONFI the Address Cycle to Data Loading time. The `Timer` μFSM can introduce such a time delay in the waveform. As we will see next, however, BABOL tries to spare the SSD Architect from certain timing aspects in the waveforms.

B. *Inter- and Intra-Segment Timing Handling*

Timing management in BABOL is a responsibility divided between the framework and the SSD Architect. To make the responsibilities clear, BABOL separates the delays specified in a waveform into three categories. The first category corresponds to timed waits *within* a μFSM. For example, we have shown how an *address latch* drives the ONFI pins in Figure 2. There are several specific time delays involved in producing that portion of the waveform, such as $t_{CS}, t_{CH}, t_{CALS}, t_{CALH}$, and others. Controlling these timed waits is the responsibility of the μFSM's implementation. This way, BABOL allows the SSD Architect to reason at a much higher abstraction level than at the ONFI pins level.

The second time delay category involves mandatory wait times *right before or right after* a μFSM. One example of such a delay is depicted in Figure 7. After a `READ` operation's command and address are latched, the standard mandates that a t_{WB} wait be observed. This wait is still a part of the `C/A Writer`'s responsibility. Note that the latter μFSM implementation has several subterfuges to implement this type of wait, since they are usually very short. For instance, the μFSM may leverage the knowledge of BABOL's implementation and infer that there is no way a next waveform segment can be chained very soon after a `READ`-related command/address latch. Therefore, a `C/A Writer` implementation may internally simply ignore t_{WB}. However, latching commands other than a `READ` may require waits longer than t_{WB}. In those cases, BABOL's `C/A Writer` explicitly implements the delays. Either way, as we said, this delay time is the responsibility of the μFSMs.

The third and last type of time delay category involves *waits between consecutive* μFSMs. This type of delay is the responsibility of the SSD Architect who develops the operations' logic. Figure 7 shows the example of t_R, which corresponds to the time needed for the package to move data from the Flash Array into the page buffer, a staging area that holds data before it can be shipped out of the package. The operation logic must explicitly issue this wait through the `Timer` μFSM or some other indirect mechanism (we will show one in the Section V.)

C. *Working with a New Package*

Nowadays, most every Flash package is ONFI compatible, which means that the electrical and command interfaces are standardized. BABOL takes advantage of this fact to allow its μFSMs to generate waveforms that are compatible with these packages. However, each package has unique booting, calibration, and initialization steps that are not covered by ONFI. For example, some packages boot in SDR data mode and can only be reconfigured to faster data modes through that interface. For another example, the traces connecting the controller and Flash packages can be different even in different instances of the same device. The controller may need to individually adjust the waveform phase for each package. For yet another example, the SSD Architect may change the default parameters of packages through the `SET FEATURES` operation. To make matters worse, depending on the package, some or all of these adjustments need to be done at every single boot.

BABOL also provides tools to help the SSD Architect incorporate new packages. There is a calibration tool to detect phase differences and suggest adjustments, and BABOL software-based approach to operation logic is also useful to express the booting and initialization logic of individual packages. We omit the specific description of these tools due to space restrictions but explain how BABOL uses software to define new operations next.

V. SOFTWARE ENVIRONMENT

The μFSMs introduced in the previous section can each emit a waveform segment, a portion of an operation. Several segments need to be concatenated to form a full operation. Simply put, the software environment in BABOL supports describing and concatenating segments.

Single-Transaction Operations. We show how BABOL allows an SSD Architect to perform compositions of μFSMs through some examples, shown in Figure 8. The figure depicts three variations of the `READ` operation.

The `READ STATUS` operation, shown in Algorithm 1 as a mix of C++ and pseudo-code, checks whether a LUN has finished performing a previously requested task, as its name implies. It entails issuing a specific command ID and reading back the status (i.e., a composition of the execution of a `C/A Writer` and a `Data Reader`.)

The composition of μFSMs appears in lines 2..6. This sequence activates the desired LUN (*chip*) in line 2, issues the command `0x70` (`READ STATUS`) in line 3, reads data (setting up a DMA destination address first) in lines 4–5, and deactivates the LUN in line 6. The composition of μFSMs

Algorithm 1: READ STATUS

```
Input: $chip$, $buf$
Output: return code
1  $\lambda \leftarrow$ {
2      $\mu fsm.chip\_control(1 << chip)$;
3      $\mu fsm.write\_ca(\{0x70, CMD\})$;
4      $pkt.set\_dma\_address(\&buf)$;
5      $\mu fsm.data\_read(4)$;
6      $\mu fsm.chip\_control(0)$;
7  }
8  co_await $add\_transaction(\lambda)$;
9  co_await $mem\_change(buf)$;
10 co_return buf;
```

Algorithm 2: READ with Change Column

```
Input: $chip$, $buf$
Output: return code
1  $\lambda \leftarrow$ {
2      $\mu fsm.chip\_control(1 << chip)$;
3      $\mu fsm.write\_ca(\{0, CMD\}, ..., \{0x30, CMD\})$;
4      $\mu fsm.chip\_control(0)$;
5  }
6  co_await $add\_transaction(\lambda)$;
7  repeat
8      $status \leftarrow read\_status()$;
9  until ($status == 0x40$);
10 $\lambda \leftarrow$ {
11     $\mu fsm.chip\_control(1 << chip)$;
12     $\mu fsm.write\_ca(\{0x05, CMD\}, ..., \{0xE0, CMD\})$;
13     $pkt.set\_dma\_address(\&buf)$;
14     $\mu fsm.data\_read(buf.size())$;
15     $\mu fsm.chip\_control(0)$;
16 }
17 co_await $add\_transaction(\lambda)$;
18 co_return 0;
```

Algorithm 3: pseudo-SLC READ

```
Input: $chip$, $buf$
Output: return code
1  $\lambda \leftarrow$ {
2      $\mu fsm.chip\_control(1 << chip)$;
3      $\mu fsm.write\_ca(\{0, CMD\}, ..., \{PSLC, CMD\})$;
4      $\mu fsm.write\_ca(\{0, CMD\}, ..., \{0x30, CMD\})$;
5      $\mu fsm.chip\_control(0)$;
6  }
7  co_await $add\_transaction(\lambda)$;
8  repeat
9      $status \leftarrow read\_status()$;
10 until ($status == 0x40$);
11 $\lambda \leftarrow$ {
12     $\mu fsm.chip\_control(1 << chip)$;
13     $\mu fsm.write\_ca(\{0x00, CMD\}$;
14     $pkt.set\_dma\_address(\&buf)$;
15     $\mu fsm.data\_read(buf.size())$;
16     $\mu fsm.chip\_control(0)$;
17 }
18 co_await $add\_transaction(\lambda)$;
19 co_return 0;
```

Fig. 8. Operation composition and variations using BABOL and C++ coroutines: Algorithm 1 shows how to encode a READ STATUS operation to determine whether a LUN completed its previously assigned task. Algorithm 2 shows a READ with a column change (i.e., it reads a chunk of a page.) It invokes READ STATUS to check when the data can be transferred. Algorithm 3 triggers a full page READ. We show it here in a variation called pseudo-SLC, which sacrifices space for performance and longevity [14]. The latter two algorithms are variations of one another (differences in gray). Encoding each algorithm would require a full hardware implementation, with all the additional validation efforts and area consumption that hardware development entails. With BABOL, it is easy to reuse one operation's logic into another or produce variations via software-based techniques.

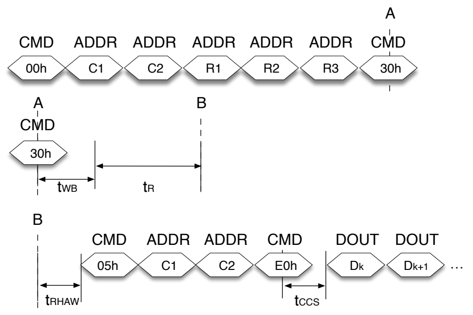

Fig. 9. Waveform of an ONFI READ operation produced by Algorithm 2. (Top) Lines 1..6 enqueue the command and address latch portion of the wave. (Center) Lines 7..9 effectively poll the LUN readiness rather than wait for a fixed t_R. (Bottom) Lines 10..17 enqueue the wave segment that determines the chunk to read and that transfers the data out of the package.

calls is what implements a *transaction*. The latter captures the parameters necessary to generate a waveform segment and is executed atomically. Moreover, a transaction monopolizes the channel for as long as it takes to transmit the waveform segment it encodes.

There are two salient aspects of how the transaction—and, ultimately, the operation—is implemented. First, instead of executing the transaction immediately, Algorithm 1 enqueues it for later execution in line 8. We use *C++ lambdas* to wrap the transaction and the call `add_transaction()` to send it to an execution queue. The second salient aspect is that we use *C++ coroutines* to force the operation to relinquish control through the `co_await`, also in line 8.

We chose lambdas and coroutines here because they are particularly suited to our goals. Lambdas can create anonymous functions that can be passed as parameters for later execution. Coroutines are a lightweight context-switching mechanism that is very programmer-friendly. Note that BABOL *does not mandate that the software environment be implemented with these mechanisms*. Ultimately, other programming language constructs support the same (e.g., function pointers and multi-threading) and can be used to implement BABOL's software environment just as effectively. In fact, we will soon experiment with other constructs to implement the same abstractions.

Our READ STATUS operation loops until the status changes, in line 9. The operation can be descheduled at this point, thanks to another `co_await`. Once the status is known, it is sent as a return code to the operation in line 10. This will be useful because, as we discuss next, an operation can be nested into another.

Multi-Transaction Operations. More sophisticated operations have longer waveforms that are divided into several transactions. This is the case, for instance, of a READ with a Column Address Change, shown in Algorithm 2. This READ variation can transfer bytes starting at any offset into the page; it is helpful when large-page devices (e.g., 16 KB) subdivide a page into 4 KB ones.

The operation's first transaction entails latching the command ID and address, in lines 2..4. Once the LUN receives this initial transaction, it must fetch the page from the Array and put its contents into a Page Register. ONFI mandates a wait time here, called t_R. Due to the nature of Flash memory, this time is highly variable. Therefore, SSD Architects use the READ STATUS operation to poll for the end of a read, instead of using a timed wait, in lines 7..9. The polling loop only breaks when READ STATUS returns a "done" code (`0x40`), at which point the page contents can be transferred from the LUN into the controller.

The last transaction appears in lines 11..15. It triggers the data transfer (command IDs delimiters `0x05` and `0xE0`) and the desired start offset (omitted, shown in an ellipsis). If we change the column to the beginning of the page, a READ with Column Address Change degenerates into a full-page READ. For that reason, many SSD Architects only implement the former operation in actual devices. The waveform for that operation appears in Figure 9.

Lastly, we show a pseudo-SLC variation of a READ operation in Algorithm 3. This is a nonstandard operation that some packages support. It allows using TLC, MLC, and QLC cells as SLC ones for performance and longevity benefits [14]. Note that, thanks to BABOL's software environment, conceiving such an operation—converting Algorithm 2 into Algorithm 3—is trivial.

Operations Interleaving. BABOL's software environment is composed of two additional entities besides the operations defined by the SSD Architect like the ones we described above: a *task* scheduler and a *transaction* scheduler (cf. Figure 5's `Operation Scheduling` module).

The *Task Scheduler* determines when a new operation request from the FTL can be admitted and which previously admitted operation should take control next, given that all operations are implemented as coroutines. A simple version of the Task Scheduler can admit an operation when a given package is available and implement fair scheduling among the running operations. A more complex task scheduler could differentiate task priorities. For example, it could prioritize latency-sensitive workloads such as database logging by making these tasks receive more attention from the scheduler. The *Transaction Scheduler* decides the order in which the transactions sitting on the individual operation use the channel. A simple version of this scheduler can implement a round-robin approach. A more advanced transaction scheduler could prioritize commands for different LUNs.

BABOL does not mandate or enforce any objective for these schedulers. It is not BABOL's goal to decide which approach is the best; there is no single winning approach. It is the job of an SSD Architect to make decisions about scheduling strategy and to implement their chosen set of compromises. BABOL's role here is to make implementing and executing Task and Transaction Schedulers easy.

Discussion. The expressive power of BABOL lies in its repertoire of μFSMs and the composition possibilities it supports (e.g., concatenation, nesting, and use of control logic). The choice of C++'s lambdas and coroutines is far from the only option to implement these concepts. To show this, we developed a second software environment that implemented transaction queuing and context switching using the Real-Time OS FreeRTOS [11]. FreeRTOS is an open-source, publicly available software stack commonly utilized for applications with tight scheduling deadlines. However, the effort involved in writing operations with RTOS is markedly different than with C++. C++ is easier to program but requires a processor with enough speed to sustain its heavy runtime. In turn, FreeRTOS is designed to require a much lighter weight processor to run, but it demands more expertise from the programmer.

Ultimately, the critical factor about the software environment is that it needs to schedule future transactions while the LUNs or the channel are busy. To achieve this, one needs to pair the chosen runtime to a properly sized processor. This pairing should consider that the operations are not too demanding; they simply form new transactions and enqueue them. What takes precious computing cycles is context-switching among tasks and reordering transactions. Figuring out the processing power necessary to achieve this balance is one of the topics of the next section.

VI. EXPERIMENTS

In this section, we validate our technical contributions based on a series of experiments. The experiments are designed to answer the following specific questions:

- Are BABOL-based controllers viable when compared to their hardware-based counterparts (Section VI-A)?
- Why is there a performance difference between BABOL's software alternatives (Section VI-B)?
- How does BABOL perform in an actual device (Section VI-C)?
- How is the relative effort of encoding operations in BABOL versus traditional controllers (Section VI-D)?
- What is the relative resource consumption of a BABOL-based controller (Section VI-E)?

Experimental Setup. We implemented BABOL using the same hardware as the `Cosmos+` OpenSSD prototyping platform [25]. The `Cosmos+` is a PCIe device based on a Xilinx Zynq 7000 FPGA/SoC containing two ARM Cortex-A9 cores. We generate all the bitstreams and binaries using Vivado/Vitis 2022.1. The μFSMs are written in Verilog, and the operations, task schedulers, and transaction schedulers we use in the experiments, along with the BABOL's modules that support them, are written in C++20.

Flash Packages. The `Cosmos+` supports interchangeable Flash packages mounted on a SO-DIMM form factor. The three package types we used in our experiments are described in Table I. They all comply with ONFI's NV-DDR2 Data Interface (max. 200 megatransfers/second) but the Hynix and Toshiba SO-DIMMs are wired for eight LUNs per channel, and the Micron is wired for only two.

TABLE I
FLASH MEMORY PARAMETERS

Parameters	Value
Page read time (Hynix)	100 μs
Page read time (Toshiba)	78 μs
Page read time (Micron)	53 μs
Page read size	16384 B
Page transfer time (100 MT/s)	185 μs
Page transfer time (200 MT/s)	100 μs

Workloads. For our microbenchmarks, we use a workload generator that injects requests directly into the storage controllers as if they were coming from the FTL. This injection is done as if the controllers were attached to the rest of an SSD machinery, i.e., data is DMA-ed in and out of DRAM, and commands come from the FTL control path (cf. Fig. 5). We use only `READ` operations on our workload because t_R, the wait between requesting a page to an Array and the Array's signaling that it is OK to fetch it is very small relative to the equivalent `PROGRAM` and `ERASE` times. A small wait time means the controller needs to react faster, and it is this reaction time that we wish to evaluate.

BABOL Alternative Implementations. We evaluate two different versions of BABOL. One version is based on the standard C++ coroutines, with operations coded as shown in Section V. The other version is based on an RTOS runtime, where operations use more efficient context-switching mechanisms at the cost of higher code complexity. In our experiments, we quantify how much performance we give up to achieve ease of coding.

We evaluate BABOL on two types of processors: low-frequency Xilinx Microblaze soft-cores and faster ARM cores in the Zynq 7000. The rationale here it to determine what is a minimum processor speed that each controller version requires for it to perform well.

A. Effects of the Software Overhead

In this experiment, we evaluate how different variations of BABOL-based controllers perform. We compare controllers on two axes: varying the processor frequency on which we run the controller from 150 MHz soft-cores to 1 GHz using Zynq's ARM core and varying the channel frequency using 100 MT/s and 200 MT/s. Slow processors will take extra time to schedule operations, and fast channels may become idle if operations are not given quickly enough. The goal is to establish under which conditions the FreeRTOS and the coroutine-based controllers deliver comparable performance to the baseline controller.

The baseline controller is a hand-built, hardware-based controller that implements scheduling logic equivalent to the software ones but in Verilog. We submit a sequence of read operations through each channel controller while varying the number of LUNs targeted from 2 to 8. Recall that Micron packages are physically arranged with only 2 LUNs per channel. Figure 10 shows the results of this experiment.

We observe two main trends. First, as the number of packages in a channel increases, its performance tends to improve. This behavior is expected because no single LUN can use all the available channel bandwidth, but the channel capacity is reached with enough LUNs. The second trend is that, as the processor gets faster, the software-based controllers (RTOS and Coroutine) speed up as well. This behavior is also expected, but it exposes some differences between the RTOS-based and coroutine-based controllers.

The RTOS-based controller performs very similarly to the baseline hardware in most 200 MT/s cases if enough processing capacity is available. It only underperforms on a 150 MHz softcore, the maximum frequency at which we could run our softcore. However, more modern FPGA fabrics can

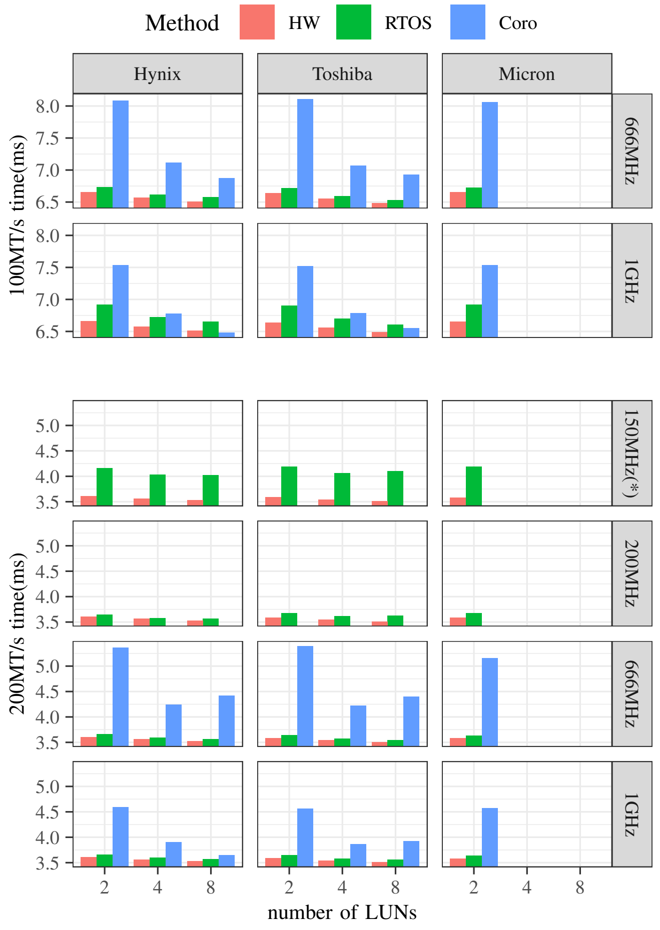

Fig. 10. Performance of the different Flash packages using 100 or 200 MT/s channels and running BABOL-based controllers on CPUs ranging from 150 MHz and up to 1 GHz. The case marked with an '*' is implemented via a soft-core. The other cases are implemented by manipulating the clock frequency of an ARM core. For each frequency, we tested a Coroutine- and an RTOS-based software controller ('RTOS' and 'Coro') and used a hardware-based controller ('HW') as the baseline.

run softcore faster, and at 200 MHz, at least on an ARM core, the RTOS controller is viable.

The coroutine controller is the fastest on 100 MT/s channels with 8 LUNs on a 1 GHz ARM core. The reason for this is that slow channels are busier, giving that controller ample time to schedule commands in advance. With more operations available, the transaction scheduler can perform better. We will explore this case further in the next sections.

Lastly, this experiment shows that SSD Architects can choose BABOL-based controllers in most scenarios. If they prefer an easier-to-program controller, they should limit the channel frequency and provide a fast processor. If they prefer lower-end processors, they can use RTOS-based controllers, but need more programming expertise.

B. Coroutine Controller Overhead Breakdown

In this experiment, we look for the reasons behind the coroutine controller's occasionally slower performance. We issue the same workload as in the previous experiment, which consists of a sequence of `READ` operations described in Figure 8 as Algorithm 2. We run the RTOS and Coroutine controllers using a 1 GHz ARM processor from the Zynq 7000 platform. We use only one LUN in this experiment because we want to establish a baseline performance without any interference.

In order to accurately capture the timing, we connected our Flash Packages to a Logical Analyzer (Keysight 16862A). This equipment can measure the times of ONFI events precisely via hardware probes, which allows us to forego any software timestamping probes that could inject some variance in the process. Figure 11 shows a screenshot of the Analyzer for the RTOS case and one for the Coroutine case.

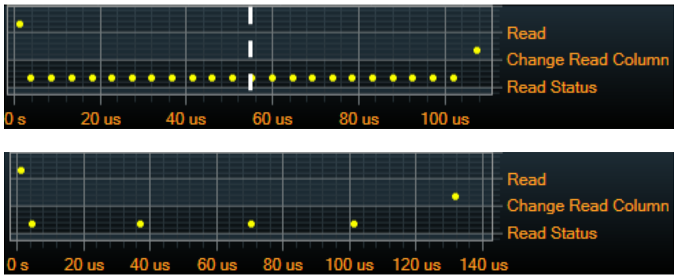

Fig. 11. The ONFI Logical Analyzer screenshots show a `READ` operation's intermediate steps in the RTOS (Top) and Coroutine (Bottom) cases.

What we see in the timing diagram is the following: The `READ` described in Figure 8 as Algorithm 2 starts by issuing a `READ` command and page address latch (lines 1..5). The operation then issues a `READ STATUS` repeatedly (lines 7..9) as a way to poll for the possibility that the package finished reading before t_R expired. Once the `READ STATUS` returns a "done" code, the operation triggers a data transfer by issuing a `CHANGE READ COLUMN` (lines 10..16).

As Figure 11 shows, the polling frequency that RTOS can achieve is higher than the polling frequency that the Coroutine variation can achieve. As a result, the RTOS controller can detect the conclusion of the `READ` step quicker. In contrast, the Coroutine controller takes in the order of 30μs at each polling cycle, resulting in a commensurate delay in detecting the end of the `READ`. Note that this is a worst case scenario. In practice, it can happen that the a polling cycle starts right after the `READ` completed, resulting in much less delay. The time differences we see in the previous experiment are due to the accumulation of delay that the RTOS controller experiences. We note that, when many LUNs are used, the channel gets busy enough that polling frequency becomes less relevant to performance. This phenomenon makes the Coroutine controller viable in these cases, as we will see on the next experiment.

C. Evaluating End-to-End Performance

In this experiment, we evaluated how a BABOL controller behaves inside an actual SSD device. To investigate this, we modified the `Cosmos+` OpenSSD [25], replacing its original storage controller with BABOL. We ran both the RTOS and Coroutine versions of BABOL on a 1 GHz ARM core available on the OpenSSD's Zynq 7000 platform. We used only one channel of the OpenSSD and populate it with Hynix Flash packages, varying the number of LUNs attached to the channel ("ways") from 1 to 8. We initialized the baseline and the modified OpenSSDs with data and issued two READ workloads against them using the `fio` tool: one sequential and one random. Figure 12 shows the results of this experiment.

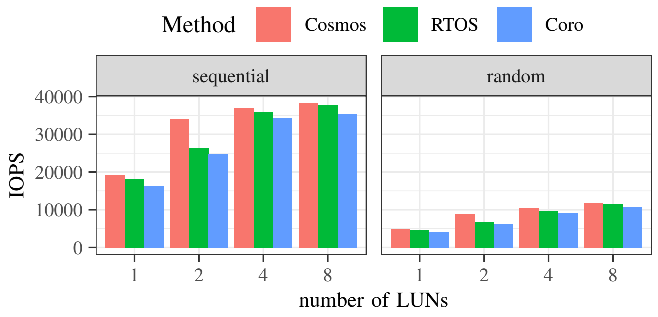

Fig. 12. Performance results from executing READ operations against the standard and a modified version of the `Cosmos+` OpenSSD that uses a BABOL storage controller: (left) a sequential workload and (right) a random workload. In each workload, we varied the number of "ways" (LUNs) attached to the channel from 1 to 8 ways.

We can observe that when the channel is fully populated with 8 LUNs, the difference in bandwidth between the baseline `Cosmos+` OpenSSD and the BABOL modified ones is less than 2% and 8% for the RTOS and Coroutine cases, respectively, in the sequential workload, and 3% and 9%, in the random workload. The reason is that, as the channel becomes naturally busy, the polling delay we reported in the previous experiment is less important. With a busy channel, the operations have far fewer polling resubmissions than with an idle channel because other operations compete for the channel. Despite the performance difference, we think that the RTOS and Coroutine stacks have their roles. We speculate that the RTOS stack may reach the same performance as the baseline if we used more powerful processors available in platforms newer than Zynq 7000. The Coroutine stack makes it very easy to quickly try operation variations, and this flexibility comes handy during the development of a new SSD.

D. Ease of Programmability

In this experiment, we try to contrast the effort involved in the development of ONFI operations using hardware and our software-defined methods. We compare BABOL's programmable software environment with two baselines: a synchronous hardware controller from Qiu et al. [50] and an asynchronous but non-programmable hardware controller found on the Cosmos+ SSD prototyping platform [25]. For each of these implementations, we count the number of lines in basic `READ`, `PROGRAM`, and `ERASE` operations. Table II shows the results of this evaluation.

TABLE II
NUMBER OF LINES OF CODE INVOLVED IN DIFFERENT OPERATIONS.

	Synchronous HW-based [50]	Asynchronous HW-based [25]	BABOL
READ	420	454	58
PROGRAM	420	260	44
ERASE	327	203	27

Admittedly, the process of comparing the effort of developing C++ software and Verilog hardware by lines of code is far from accurate. However, we anecdotally noticed that the effort to encode operations significantly decreases as one transitions from conceiving hardware circuits to writing software with BABOL's aid. This is thanks to (a) the fact that redundant logic related to BTCs that each operation had to encode was eliminated by the μFSM abstraction and (b) the fact that working with enqueuing operations asynchronously and in software liberates the programmer from timing closure issues and other hardware issues.

E. Evaluating the Necessary Area

In this experiment, we measure the controllers sizes in terms of their FPGA resource usage independent of the operations. We use the same controller variations as in the previous experiment. Table III shows the results of this evaluation.

TABLE III
FPGA RESOURCES USED FOR EACH TYPE OF CONTROLLER.

	Synchronous HW-based [50]	Asynchronous HW-based [25]	BABOL
LUT	9343	3909	3539
FF	13021	3745	3635
BRAM	11.5	8	6

We observe that BABOL utilizes fewer hardware resources than the other controller variations. This is attributed to the complex logic being transferred to software, leaving only the essential modules in the hardware. We did not account for the area needed for a processor to run the BABOL's software environment because, in SoCs, this area does not come from the FPGA resources.

VII. RELATED WORK

There is a rich body of literature about NAND Flash controllers. As we hinted previously, the two closest works to ours are the OCOWFC controller [50] and the asynchronous one on the `Cosmos+` OpenSSD project [25]. Other similar controllers exist, such as in BluDBM [35] [17], Dysource [55], SoftSSD [56], BlueSSD [29], FSSD [57], and SSDe [37] that were introduced in the context of more flexible SSD platforms for rapid prototyping. Some of these Flash controllers also resort to software to some extent but, to the best of our knowledge, BABOL is the first work that fully describes a systematic approach to conceive and implement Flash manipulation operators and to interleave them, all via a unified framework composed of software and μFSMs hardware. BABOL's code base is flexible and its footprint is small, making it possible to embed it in differnt of the SSD prototyping platforms.

The literature is also abundant in works that propose different ways to expose or structure Flash Controllers. Regarding exposing a controller, SDF [46] and OCOWFC [50] allow applications to directly access the Flash Channels, an approach known as Open Channel [4]. Regarding innovations on controller internals, Flash-Cosmos [47] introduced new flash command sets that can perform bit-wise computations over the pages they retrieve. The Decoupled SSD [21] and Networked SSD [22] controllers propose design alternatives to improve performance or reliability by exploring more complex interconnection structures for Flash memories. While we have not integrated these architectures into BABOL at this juncture, we think that the same principles that allow it to use software for operations over traditional architectures can also apply.

A consequence of allowing very flexible SSD configurations is the difficulty in finding the right parameters for a given scenario. Autoblox [31] is a framework that automates such choice of parameters. We see this type of work as complementary to BABOL.

Lastly, some ideas used in BABOL were tried in different contexts. For example, the software/hardware codesign in software-defined radios resembles the contract that BABOL adopts [1]. The idea of using elementary instructions to build larger operations was also used in building DRAM controllers, although the instructions there are issued by hardware [27].

VIII. CONCLUSION

In this paper, we introduced BABOL, a framework to develop NAND Flash channel controllers. BABOL exposes an asynchronous programming model in which Flash operations that are written in software enqueue instructions that are later executed by programmable hardware. We showed how this flexibility allows SSD Architects to develop advanced, optimized operations more easily than in traditional synchronous, hardware-only controllers.

We proposed two alternative software environments for BABOL. The first one provides an easier programming environment that we believe can suit even regular software C++ application programmers with little firmware experience. The second software environment is stricter, requiring more experience from the programmer in real-time applications. Because more responsibility is shifted to the programmer, it can be supported by less powerful processors.

IX. ACKNOWLEDGEMENT

This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00971, Logic Synthesis for NVM-based PIM Computing Architecture) and by the Swiss State Secretariat for Education (SERI) in the context of the SmartEdge EU project (grant agreement No. 101092908).

References

[1] R. Akeela and B. Dezfouli, "Software-defined radios: Architecture, state-of-the-art, and challenges," *Computer Communications*, vol. 128, pp. 106–125, 2018.

[2] S. Aritome, *NAND flash memory technologies*. John Wiley & Sons, 2015.

[3] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee, "FlashNeuron: SSD-Enabled Large-Batch training of very deep neural networks," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 2021, pp. 387–401.

[4] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The linux Open-Channel SSD subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[5] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[6] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[7] P. Chen, C. Zhang, H. Jiang, Z. Wang, and S. Yue, "High performance low complexity BCH error correction circuit for SSD controllers," in *2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, 2015, pp. 217–220.

[8] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5, pp. 332–343, 2009.

[9] C. Dirik and B. Jacob, "The performance of pc solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[10] Z. Du, Z. Dong, K. You, X. Jia, Y. Tian, Y. Wang, Z. Yang, X. Fu, F. Liu, Q. Wang, L. Jin, and Z. Huo, "A novel program suspend scheme for improving the reliability of 3D NAND flash memory," *IEEE Journal of the Electron Devices Society*, vol. 10, pp. 98–103, 2022.

[11] "freeRTOS," https://freertos.org.

[12] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A survey of FPGA-based LDPC decoders," *IEEE Communications Surveys Tutorials*, vol. 18, no. 2, pp. 1098–1122, 2016.

[13] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, p. 1141–1155, jun 2013.

[14] HyerStone, "How pseudo-SLC mode can make 3D NAND flash more reliable," https://www.hyperstone.com/en/How-pseudo-SLC-mode-can-make-3D-NAND-flash-more-reliable-2524.html.

[15] S. Im and D. Shin, "ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer," *J. Syst. Archit.*, vol. 56, no. 12, p. 641–653, dec 2010.

[16] S. Im, D. Shin, D. Shin, D. Shin, and D. Shin, "Flash-aware RAID techniques for dependable and high-performance flash memory SSD," *IEEE Trans. Comput.*, vol. 60, no. 1, p. 80–92, jan 2011.

[17] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: an appliance for big data analytics," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, p. 1–13.

[18] M. Jung, "OpenExpress: Fully hardware automated open research framework for future fast NVMe devices," ser. USENIX ATC'20. USA: USENIX Association, 2020.

[19] K. Kambatla and Y. Chen, "The truth about mapreduce performance on SSDs," in *Proceedings of the 28th USENIX Conference on Large Installation System Administration*, 2014, p. 109–117.

[20] M. Kang, W. Lee, J. Kim, and S. Kim, "PR-SSD: Maximizing partial read potential by exploiting compression and channel-level parallelism," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 772–785, 2023.

[21] J. Kim, M. Jung, and J. Kim, "Decoupled SSD: Rethinking SSD architecture through network-based flash controllers," ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023.

[22] J. Kim, S. Kang, Y. Park, and J. Kim, "Networked ssd: Flash memory interconnection network for high-bandwidth ssd," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 388–403.

[23] S. Kim, J. Bae, H. Jang, W. Jin, J. Gong, S. Lee, T. J. Ham, and J. W. Lee, "Practical erase suspension for modern low-latency SSDs," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19, 2019.

[24] J. Kwak, J. Lee, D. Lee, J. Jeong, G. Lee, J. Choi, and Y. H. Song, "GALRU: A group-aware buffer management scheme for flash storage systems," *IEEE Access*, vol. 8, pp. 185 360–185 372, 2020.

[25] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ OpenSSD: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, 2020.

[26] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, p. 273–286.

[27] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[28] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory SSD in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, 2008, p. 1075–1086.

[29] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, J. Kim *et al.*, "Bluessd: An open platform for cross-layer experiments for nand flash-based ssds," in *WARP-5th Annual Workshop on Architectural Research Prototyping*, 2010.

[30] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: The design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, p. 447–461.

[31] D. Li, J. Sun, and J. Huang, "Learning to drive software-defined solid-state drives," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[32] H. Litz, J. Gonzalez, A. Klimovic, and C. Kozyrakis, "RAIL: Predictable, low tail latency for NVMe flash," *ACM Trans. Storage*, vol. 18, no. 1, jan 2022.

[33] C.-Y. Liu, J. B. Kotra, M. Jung, M. T. Kandemir, and C. R. Das, "Soml read: Rethinking the read operation granularity of 3D NAND SSDs," ser. ASPLOS '19, 2019, p. 955–969.

[34] C.-Y. Liu, Y. Lee, M. Jung, M. T. Kandemir, and W. Choi, "Prolonging 3D NAND SSD lifetime via read latency relaxation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2021, p. 730–742.

[35] M. G. Liu *et al.*, "BlueFlash: a reconfigurable flash controller for BlueDBM," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[36] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, mar 2017.

[37] Y. Lu, L. Yu, and D. Chen, "Ssde: Fpga-based ssd express emulation framework," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[38] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 3, dec 2018.

[39] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Comput. Surv.*, vol. 46, no. 3, jan 2014.

[40] R. Micheloni, *3D Flash Memories*. Springer Publishing Company, Incorporated, 2016.

[41] R. Micheloni, A. Marelli, and K. Eshghi, *Inside Solid State Drives (SSDs)*. Springer Publishing Company, Incorporated, 2012.

[42] R. Micheloni, A. Marelli, and R. Ravasio, *Error correction codes for non-volatile memories*. Springer Science & Business Media, 2008.

[43] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (O3): An out-of-order flash memory controller architecture," vol. 60, no. 5, p. 653–666, 2011.

[44] "NVM express base specification revision 2.0c," https://nvmexpress.org/specifications/, 2022.

[45] "Open NAND flash interface specification revision 5.1," https://www.onfi.org/specifications, 2022.

[46] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, “Sdf: software-defined flash for web-scale internet storage systems,” *SIGPLAN Not.*, vol. 49, no. 4, p. 471–484, 2014.

[47] J. Park, R. Azizi, G. F. Oliveira, M. Sadrosadati, R. Nadig, D. Novo, J. Gómez-Luna, M. Kim, and O. Mutlu, “Flash-cosmos: In-flash bulk bitwise operations using inherent computation capability of nand flash memory,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 937–955.

[48] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu, “Reducing solid-state drive read latency by optimizing read-retry,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21. Association for Computing Machinery, 2021, p. 702–716.

[49] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, “Exploiting internal parallelism of flash-based SSDs,” *IEEE Computer Architecture Letters*, vol. 9, no. 1, p. 9–12, jan 2010.

[50] Y. Qiu, W. Yin, and L. Wang, “A high-performance open-channel open-way NAND flash controller architecture,” pp. 91–98, 2021.

[51] Y. Qiu, W. Yin, and L. Wang, “A high-performance and scalable NVMe controller featuring hardware acceleration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1344–1357, 2022.

[52] Y. Shim, M. Kim, M. Chun, J. Park, Y. Kim, and J. Kim, “Exploiting process similarity of 3D flash memory for high performance SSDs,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. Association for Computing Machinery, 2019, p. 211–223.

[53] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu, “The impact of solid state drive on search engine cache management,” in *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2013, p. 693–702.

[54] G. Wu and X. He, “Reducing SSD read latency via NAND flash program and erase suspension,” in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST’12. USA: USENIX Association, 2012.

[55] L. Wu, N. Xiao, F. Liu, Y. Du, S. Li, and Y. Ou, “Dysource: A high performance and scalable NAND flash controller architecture based on source synchronous interface,” ser. CF ’15. New York, NY, USA: Association for Computing Machinery, 2015.

[56] J. Xue, R. Chen, and Z. Shao, “Softssd: Software-defined ssd development platform for rapid flash firmware prototyping,” in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, 2022, pp. 602–609.

[57] L. Yu, Y. Lu, M. Mandava, E. Richter, V. S. Mailthody, S. W. Min, W.-m. Hwu, and D. Chen, “Fssd: Fpga-based emulator for ssds,” in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 101–108.

[58] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, “FlashGraph: Processing billion-node graphs on an array of commodity SSDs,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, p. 45–58.