SCHEDULING GENERAL PURPOSE ENCRYPTED COMPUTATION ON MULTICORE PLATFORMS

by

Tommy White

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2023

SCHEDULING GENERAL PURPOSE ENCRYPTED COMPUTATION ON MULTICORE PLATFORMS

by

Tommy White

Approved: ______________________________

Chengmo Yang, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ______________________________

Jamie Phillips, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: ______________________________

Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: ______________________________

Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express much gratitude to my advisor Chengmo Yang for supporting me since my days as an undergraduate. It was her encouragement that led me to consider graduate studies, and she provided me with various opportunities starting with being a grader for CPEG 222. Pursuing FHE was her idea after learning about it from Nektarios Tsoutsos and his research group, and I am very glad our research went the direction it did. Over the course of this research, Dr. Yang has been an invaluable resource for helping clarify ideas, developing new concepts to pursue, and in the actual writing process. Chaz Gouert also played a key role in this thesis, essentially being the de facto FHE expert. His contributions to the Introduction and Background sections in particular were massive. His advisor Dr. Tsoutsos played a similar role and I am very grateful for his involvement.

Many thanks also go to the group of friends I made during my undergrad years, as spending time with them has been a great stress reliever among the pressures of graduate studies. I am also ever grateful for my parents and siblings who have been pillars of support my whole life.

I would finally like to thank the University of Delaware and the Department of Electrical and Computer Engineering for funding my graduate studies. Without that support, this thesis would not have been written.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In cloud computing, outsourcing computation-intensive tasks to cloud servers is a common practice for those companies and users with limited computational resources. If the data computed upon is sensitive, however, providing it to a cloud server would expose it to various security risks. Fully homomorphic encryption (FHE) allows a user to outsource computation to a cloud server while preserving the privacy of their personal data. Specifically, the user does not need to provide unencrypted values (called *plaintexts*) or decryption keys to the server, as FHE allows operations to be performed directly on encrypted data (called *ciphertexts*). While desirable, a major drawback of these encrypted operations is that they can be orders of magnitude slower than their plaintext counterparts. Moreover, the ciphertexts contain some noise for security purposes, and the amount of noise in a ciphertext grows as more operations are performed. Therefore, each ciphertext can only withstand a limited number of operations before accumulated noise renders decryption impossible. To reduce such noise and allow for unlimited computations, an operation known as *bootstrapping* is needed. Notably, bootstrapping is significantly slower than encrypted arithmetic operations (e.g., multiplication and addition), which makes it a main performance bottleneck while evaluating FHE programs.

The allocation and scheduling of bootstrapping operations has not been well investigated, in part due to the complexity of the problem and the difficulty in finding an optimal solution. To bridge this gap, this master thesis makes three contributions. First, it formally formulates the bootstrapping scheduling problem by clearly defining the constraints that a FHE program must satisfy to confine the noise growth within a safe limit. It then develops two Integer Programming (IP) models to enforce these constraints in an optimal fashion. The first model minimizes the number of bootstrapping operations in an FHE program, while the second model optimizes the execution time of the FHE program. Since solving these IP models optimally may take exponential time, this thesis also presents a faster heuristic method for choosing bootstrap points for a target FHE program to satisfy the noise constraints and mapping the FHE program to a multi-core system in polynomial time. Last but not least, this thesis also introduces FHE-Runner, developed to faithfully map the generated FHE schedule to a multi-core system for evaluation. Experiments with realistic benchmarks shows that the proposed heuristic provides as much as a 3.22× speedup compared to the baseline method.

Chapter 1
INTRODUCTION

In cybersecurity, *encryption* is a common method of keeping sensitive data safe from unauthorized users. Given the sensitive data (called a *plaintext*) and an encryption key, an encryption algorithm will produce a new value, called a *ciphertext*. The value of the ciphertext is meaningless unless the user has access to the matching decryption key, in which case a decryption algorithm can be run to transform the ciphertext back to the original plaintext. By keeping their sensitive data encrypted and the decryption key secret, a user can keep their personal data confidential, even if an attacker gains access to their device.

Using the aforementioned approach, traditional encryption algorithms, such as Advanced Encryption Standard (AES) [16], can protect data *at rest* and data *in-transit* over a network. However, the encrypted values cannot be operated upon without first being decrypted. In contrast, Homomorphic Encryption (HE) is a special class of encryption algorithms with the ability to protect data *in-use*. The difference between HE and standard encryption is illustrated in Figure 1.1 that presents an example of adding two ciphertexts and then decrypting the sum. Under a standard encryption scheme, the decrypted value is meaningless, while under HE the decrypted value is the sum of the original plaintexts.

Using this paradigm, HE allows a user to outsource her computation to an untrusted third-party cloud server. A client encrypts her data, uploads the ciphertexts to a potentially untrusted cloud server, and finally downloads the encrypted results after a desired algorithm has been evaluated. This approach offers significant privacy benefits over traditional encryption. As illustrated in Figure 1.2, in traditional cloud computing, the user needs to expose the decryption key to the cloud server. Under

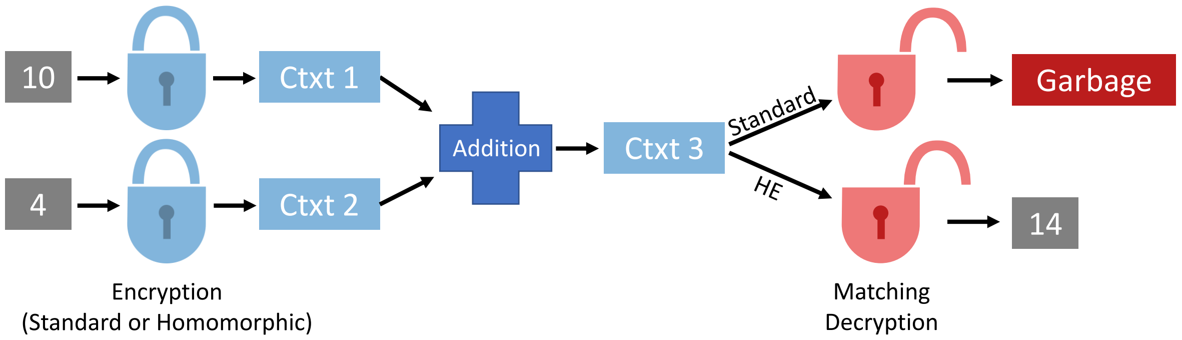


Figure 1.1: Addition of ciphertexts under standard encryption and HE

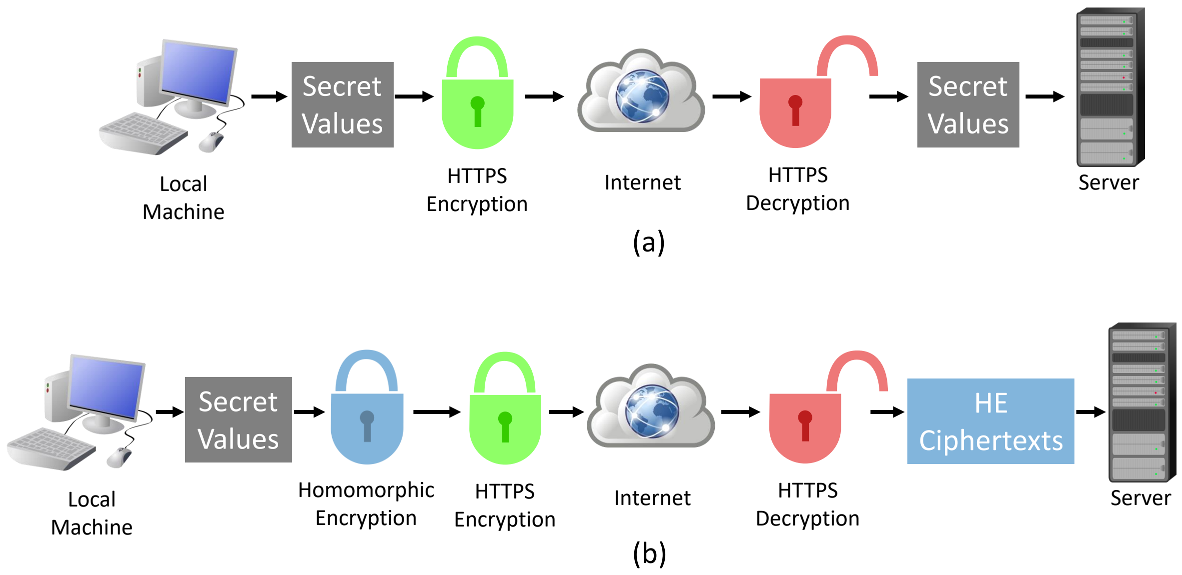


Figure 1.2: Comparison between the traditional and HE models of cloud computing

HE, however, the computation is performed directly on encrypted data, and the third party cloud server has no way to deduce any information about the underlying plaintext data. Therefore, even if a cloud server is compromised, the data will remain safe from attackers.

While desirable, the key challenge of HE is that these encrypted operations are significantly slower than their plaintext counterparts, due to two primary challenges: *ciphertext encoding* and *noise reduction*. First, HE ciphertexts typically take the form of tuples of high degree polynomials, so arithmetic operations on encrypted data correspond to polynomial addition and multiplication [17, 6], which can be orders of magnitude more complex than plaintext arithmetic operations. Second, noise (i.e. randomness) is required in ciphertexts to guarantee security, and the amount of noise grows as more operations are performed on encrypted data [43, 37]. If the noise magnitude reaches a critical threshold, it will corrupt the underlying plaintext message and prevent the client from decrypting it. Noise reduction operations prevent this corruption and allow more arithemtic operations to be performed. The most powerful form of noise reduction in HE is an operation called *bootstrapping*, which can be invoked indefinitely to allow for unlimited encrypted operations [22]. HE schemes that leverage this mechanism are referred to as *fully homomorphic encryption (FHE)*. At the same time, since bootstrapping is significantly slower than the other encrypted operations, it forms the main bottleneck of FHE computation [31].

Previous works have focused on accelerating HE from two main directions. The first approach attempts to accelerate the underlying polynomial operations (called *primitives*) inherent in FHE computation with hardware, such as Graphics Processing Units (GPUs) [32, 12, 27, 40] and Application-Specific Integrated Circuits (ASICs) [45, 46, 20, 39], while other efforts perform algorithm-level optimizations to either enhance thread-level parallelism inherent in certain polynomial operations [28, 7], or leverage CPU extensions to speedup certain polynomial operations [4]. However, an important observation is that none of these earlier works have treated FHE programs from the *application-level*, which also has a significant impact on overall performance. Specifically, while accelerating HE primitives certainly provides significant speedups, the order in which the primitives are invoked must also be carefully considered to achieve peak performance for a given application. Indeed, FHE frameworks targeting only specific use-cases (such as encrypted neural network inference) typically perform bootstrapping procedures at carefully chosen points to minimize latency [18, 5, 38]. Unfortunately, such manual tuning relies on the computational patterns specific to machine learning constructions and leverages mathematical tricks targeting neural network inference, so it is not applicable to general-purpose applications.

This thesis aims to address the aforementioned challenges by developing application-level optimizations for *general-purpose* FHE applications. Specifically, this thesis proposes a new scheduling framework called *FHE-Booster* that aims to minimize the impact of bootstrapping operations, the core bottleneck in FHE computation. Given an input algorithm in the form of a *directed acyclic graph* (DAG), which any FHE application can be viewed as, FHE-Booster takes into account the noise of various FHE operations and identifies the bootstrapping points that optimize the number of bootstrapping operations. Furthermore, FHE-Booster targets the most commonly used hardware platform for homomorphic evaluation – homogeneous CPU platforms – and divides the FHE workload optimally among multiple cores. To verify the efficacy of FHE-Booster, this thesis introduces a new execution engine that runs the generated schedules with the widely-used OpenFHE implementation of the CKKS FHE scheme [1]. The experimental evaluation consists of a set of randomly-generated FHE programs, as well as three realistic FHE benchmarks. In a nutshell, the main contributions of this thesis are summarized as follows:

- Two formal integer programming (IP) models of the bootstrapping scheduling problem, one minimizing the number of bootstrapping operations, and the other optimizing the execution time on multi-core systems;
- Two bespoke scheduling algorithms that employ hardware-aware list scheduling coupled with either the optimal IP model or custom heuristics;
- A robust execution engine that utilizes a state-of-the-art FHE library and runs on general-purpose multi-core systems.

The rest of this thesis, which expands upon the work published in the IEEE International Symposium on Hardware Oriented Security and Trust (HOST) [49] is organized as follows. Chapter 2 offers a brief background in HE and reviews related work, while Chapter 3 provides a technical overview of FHE-Booster. Chapter 4 and 5 respectively present the mathematical formulation of the bootstrapping scheduling problem and showcase the proposed heuristic. Chapter 6 elaborates on the design of the execution engine used for validation, followed by Chapter 7 that presents experimental results. Concluding remarks and a discussion of potential future work are found in Chapter 8.

Chapter 2
BACKGROUND

2.1 Homomorphic Encryption Basics

Homomorphic encryption allows encrypted data to be computed upon without revealing the underlying plaintext. To explain this concept more formally, this chapter uses $E[p]$ to denote the encryption of a plaintext p, and $D[c]$ to denote the decryption of a ciphertext c. Given two ciphertexts $c_1 = E[p_1]$ and $c_2 = E[p_2]$, under homomorphic encryption, computation can be directly performed upon c_1 and c_2. This means $p_1 + p_2 = D[c_1 + c_2]$ and $p_1 \times p_2 = D[c_1 \times c_2]$. More generally speaking, for some function f_{ptxt} that operates on a plaintext, there should exist some function f_{ctxt} that operates on a ciphertext such that $f_{ptxt}(p) = D[f_{ctxt}(E[p])]$. An encryption scheme where such a property is true for all functions is called *fully* homomorphic encryption (FHE).

To provide an intuitive explanation of how such a paradigm is implemented, let us consider a simple example, which is adapted from Craig Gentry's 2009 dissertation, the first work to introduce an FHE scheme [21]. Gentry first defines two parameters, n which is the maximum noise allowed in freshly encrypted ciphertexts, and $N >> n$ which is the noise threshold at which point a ciphertext cannot be decrypted. This example scheme only encrypts a single bit b, and the key is an odd integer $p > 2N$. To encrypt b, a ciphertext c is created as $c = E[b] = b + 2x + kp$, where x is a random integer in the range $(-n/2, n/2)$, thus implementing the random noise necessary for security, and k is "an integer chosen from some range." Note that $(c \bmod p)$ returns $b + 2x$, so $D[c] = (c \bmod p) \bmod 2 = b$. The parameters and functions of this simple example is summarized in Table 2.1.

When two ciphertexts of this kind, c_1 and c_2, are added or multiplied together, the resulting values c_3 and c_4 are:

Table 2.1: A simple FHE scheme

parameter	**value**	**type**	**range**
plaintext	b	bit	[0,1]
noise threshold	N	integer	$N >> n$
max noise of fresh ctxt	n	integer	$n << N$
key	p	odd integer	$(2N, \infty)$
initial noise value	x	integer	$(-n/2, n/2)$
key multiplier	k	integer	$(-\infty, \infty)$
ciphertext	$c = b + 2x + kp$	integer	$(-\infty, \infty)$
partial decryption	$c \bmod p$	function	$(-p/2, p/2)$
decryption	$(c \bmod p) \bmod 2$	function	[0,1]

$$
\begin{aligned}
c_3 =& b_1 + b_2 + 2(x_1 + x_2) + (k_1 + k_2)p \\
=& b_1 \oplus b_2 + 2x_3 + k_3 p \\
c_4 =& b_1 \times b_2 + 2(b_1 x_2 + b_2 x_2 + 2x_1 x_2) + (b_1 k_2 + 2x_1 k_2 + b_2 k_1 + 2x_2 k_1 + k_1 k_2 p)p \\
=& b_1 \times b_2 + 2x_4 + k_4 p
\end{aligned}
$$

Note that decryption gives the correct values, as $D[c_3] = b_1 \oplus b_2$ and $D[c_4] = b_1 \times b_2$. However, this decryption only works if the noise value x_3 and x_4 have not grown too large. In particular, assume $y_3 = b_1 \oplus b_2 + 2x_3$ and $y_4 = b_1 \times b_2 + 2x_4$. Then, decryption works only if y_3 and y_4 are in the range $[-N, N]$, which is guaranteed to be within $[-p/2, p/2]$, the valid range of values for $(c \bmod p)$, assuming a modulo based on rounded division. If y_3 or y_4 is outside the range $[-N, N]$, then $(c \bmod p)$ may not return y_3 or y_4, but rather y'_3 or y'_4. Unfortunately, there is no guarantee that y'_3 and y'_4 respectively matches the parity of y_3 and y_4, so decryption may return the wrong value. Notice that multiplication grows noise much more than addition does, since x_4 is much larger than x_3. Overall, noise growth is a major concern in homomorphic encryption, and must be managed properly.

While the example is simple, it demonstrates the mathematical intuition behind FHE schemes. Newer, realistic FHE schemes use ring learning with error to guarantee security [6, 17, 8, 9], a hard problem which similarly involves polynomials and modulo. Still, various FHE schemes, such as BGV [6] and CKKS [8], differ in their implementation details. However, most of those details are of little importance to this thesis because FHE-Booster does not rely on them and is compatible with any scheme that meets certain criteria. Those details that are important to FHE-Booster are discussed in the rest of this chapter.

2.2 Noise Management in Homomorphic Encryption

While all forms of HE allow for direct computation on encrypted data, not all HE schemes are equal in terms of their computational abilities. Overall, HE schemes can be divided into three distinct classes, namely, *partial HE* (PHE), which cannot be used for arbitrary computation due to its restriction to a single operation type [44, 41], as well as *leveled HE* (LHE) and *fully HE* (FHE) that are capable of executing a functionally complete set of operations on encrypted data.

From the perspective of plaintext encoding, both LHE and FHE can take individual bits, modular integers, or floating point numbers and encode them as tuples of high-degree polynomials. All polynomial coefficients are modulo a *product of primes* known as the "ciphertext modulus," which can vary greatly in size depending on the chosen parameters and the number of constituent primes at present. The size of the ciphertext modulus typically ranges between 64 and 2200 bits [32, 30]. Since the modulus size can have a negative impact on security, it must be balanced by increasing the polynomial degree, which typically ranges from 2^{10} to 2^{17} [32, 30]. Random noise is added to the coefficients of these polynomials upon encryption to guarantee the security of the cryptographic constructions. Such noise accumulates as ciphertexts are computed upon and will eventually impact the underlying plaintext message if the noise growth is not mitigated. In fact, the differentiating factor between LHE and FHE refers to the way that the noise is managed.

In LHE schemes, the mechanism to reduce noise is called *modulus switching*, which manipulates the ciphertext modulus to scale down the magnitude of the noise.

More specifically, this operation removes one of the constituent primes in the ciphertext modulus, which in turn decreases the size of the polynomial coefficients and reduces the ciphertext noise enough to allow at least one additional multiplication. Each possible value of the ciphertext modulus is referred to as a *level*, which is the origin of the name of this class of HE schemes. While this operation is relatively inexpensive, it can only be used for a limited number of times, as eventually there will be only one prime left in the ciphertext modulus. Confined by such a limit, the only way to achieve a higher noise threshold is to choose larger parameters, which in turn results in a slower computation speed due to the larger polynomial degrees and coefficient sizes. As a result, LHE is not suitable for sufficiently complex applications.

Any LHE scheme can be turned into an FHE scheme by incorporating the *bootstrapping* procedure. The intuition behind bootstrapping is that decrypting a ciphertext before it exceeds its noise budget would yield the expected, uncorrupted plaintext. Rather than having the cloud transmit a ciphertext back to the client for decrypting, re-encrypting, and sending it back to the cloud, bootstrapping performs a decryption *homomorphically* in the encrypted domain. Essentially, the user uploads an encryption of her secret key, which is referred to as a bootstrapping (or evaluation) key. The cloud evaluates the decryption algorithm using the bootstrapping key homomorphically (i.e., the plaintext is never revealed to the server), and outputs a relatively fresh ciphertext with reduced noise. Unlike modulus switching, bootstrapping enables scalable and unbounded computation for any application, regardless of the number of operations it requires to evaluate. Notably, bootstrapping also regenerates all of the primes in the ciphertext modulus, allowing modulus switching to be used in between bootstraps. Therefore, FHE is suited for all (and especially complex) computations, and is the focus of this thesis.

2.3 The CKKS Cryptosystem

Given the great potential of HE, many cryptosystems have been proposed, such as BFV [17], BGV [6], and CGGI [9]. While the FHE-Booster framework can be utilized by any FHE scheme, this thesis opts to validate it with CKKS [1] for several reasons. First, this is the only FHE scheme that naturally supports encoding floating point numbers, which is useful for many real-life applications such as neural network training and full-precision inference. Second, CKKS provides high encoding efficiency and enables "batched" computation, which is similar to single instruction, multiple data (SIMD) parallel processing. Specifically, if N is the degree of the ciphertext polynomial, batching enables a user to encode up to $\frac{N}{2}$ floating point values in "slots" of a single CKKS ciphertext [8]. Since N is typically in the order of several thousands, a single ciphertext can be used to encrypt a large vector of plaintext values, thus allowing the same algorithm to be executed across multiple independent inputs at virtually no extra cost. Moreover, current implementations of CKKS bootstrapping outperform those of BFV and BGV in terms of throughput and precision [3]. For these reasons, this thesis employs the *OpenFHE library* implementation of CKKS [1]. In this specific case, profiling data show that ciphertext multiplication is approximately $80\times$ slower than ciphertext addition, while bootstrapping is approximately $27000\times$ more expensive. The high cost of bootstrapping further motivates the need for optimal scheduling.

2.4 Related Work

Previous efforts have focused on accelerating HE primitives in hardware, as well as using algorithm-level optimizations.

In the case of hardware acceleration, both GPUs and ASICs have been used to accelerate the underlying polynomial operations inherent in FHE computation and bootstrapping. For the CKKS cryptosystem [8], Jung et al. [32] report a speedup of approximately two orders of magnitude with GPUs relative to a CPU baseline. Similarly, the cuFHE [12], REDcuFHE [27], and nuFHE [40] libraries employ GPUs to decrease the cost of FHE operations for the CGGI cryptosystem [9] by several orders of magnitude. ASIC designs for FHE also report significant speedups over CPUs: the F1 chip accelerates bootstrapping for the GSW [23], CKKS, and BGV [6] cryptosystems and demonstrates speedups of approximately three to four orders of magnitude for selected benchmarks [45]. However, F1 can efficiently accelerate only applications with limited depth, which renders complex applications such as deep neural network inference infeasible. The CraterLake ASIC mitigates this limitation by introducing custom hardware units and optimizing data movement between the ASIC and the hosting CPU [46].

Other works focus on algorithm-level optimizations to either enhance the inherent parallelism of HE schemes or exploit CPU extensions for improved performance. For the former, seamlessly incorporating the residue numbering system to a variety of HE schemes enables a multi-threaded approach to be taken for certain polynomial operations [28, 7]. Similarly, Intel's HEXL [4] utilizes fused multiply-add (FMA) and advanced vector extensions (AVX) instructions to accelerate polynomial modular multiplication and the required Number Theoretic Transforms (NTT).

Only a few works have attempted to accelerate FHE applications from the perspective of *scheduling*. The Cheetah framework [42] is dedicated to privacy-preserving neural network inference with LHE. It incorporates a scheduler called Sched-PA, tuned for fully-connected and convolutional layers, which schedules operations in a way that minimizes noise growth. However, since Cheetah opts to use LHE exclusively, bootstrapping was not considered. Similarly, the ALCHEMY framework [11] uses a custom domain-specific language (DSL) and a compiler that can map DSL programs to HE computation using a BGV implementation [10]. ALCHEMY automatically analyzes noise growth, computes the noise ceiling, and schedules noise maintenance procedures (i.e., modulus switching). However, ALCHEMY does not consider bootstrapping operations either, and hence does not scale well to complex applications [48]. Finally, the T2 compiler [25] translates high-level programs to a variety of backends consisting of popular HE libraries. For the libraries that support bootstrapping, T2 performs noise checks before multiplications and determines if bootstrapping is needed. This is a straightforward *as-late-as-possible* (ALAP) approach that produces sub-optimal results, as will be shown in Chapter 3.

Chapter 3
TECHNICAL OVERVIEW

As discussed in the previous chapter, existing hardware-based acceleration methods, while effective, do not alter the relative cost of bootstrapping compared to ciphertext multiplication and addition. Notably, bootstrapping remains the main bottleneck of FHE computation. Therefore, the fundamental goal of this work is to *minimize the impact of bootstrapping on general-purpose FHE applications at the application-level.* Specifically, this thesis aims to answer the following *two questions*: (1) Given an arbitrary FHE program, what is the minimum total number of bootstrapping operations needed? (2) When mapping an FHE program to a target multi-core system, how to judiciously select bootstrapping points to minimize execution times?

3.1 Motivating Example

Fig. 3.1 presents an example to demonstrate the need for optimizing bootstrapping scheduling. This example uses a task graph containing 5 inputs ($C1 - C5$), 10 ciphertext multiplications/additions ($OP1-OP10$), and 2 outputs ($C01-C02$). Assuming that no more than *two* multiplications can occur along a path without bootstrapping, a straightforward scheduling method would employ *as-late-as-possible* (ALAP) bootstrapping: before each multiplication, the current noise of each input ciphertext is checked and if it is too high, the corresponding ciphertext is bootstrapped; this is the approach adopted by the state-of-the-art T2 compiler [25]. Unfortunately, as illustrated in Fig. 3.1(a), this method will incur two bootstrapping operations respectively on the outputs of operations $OP6$ and $OP7$, which are highlighted in purple. Nevertheless, an alternative solution only needs to perform one bootstrapping operation on the output of $OP4$, as highlighted in Fig. 3.1(b). In fact, this solution requires the

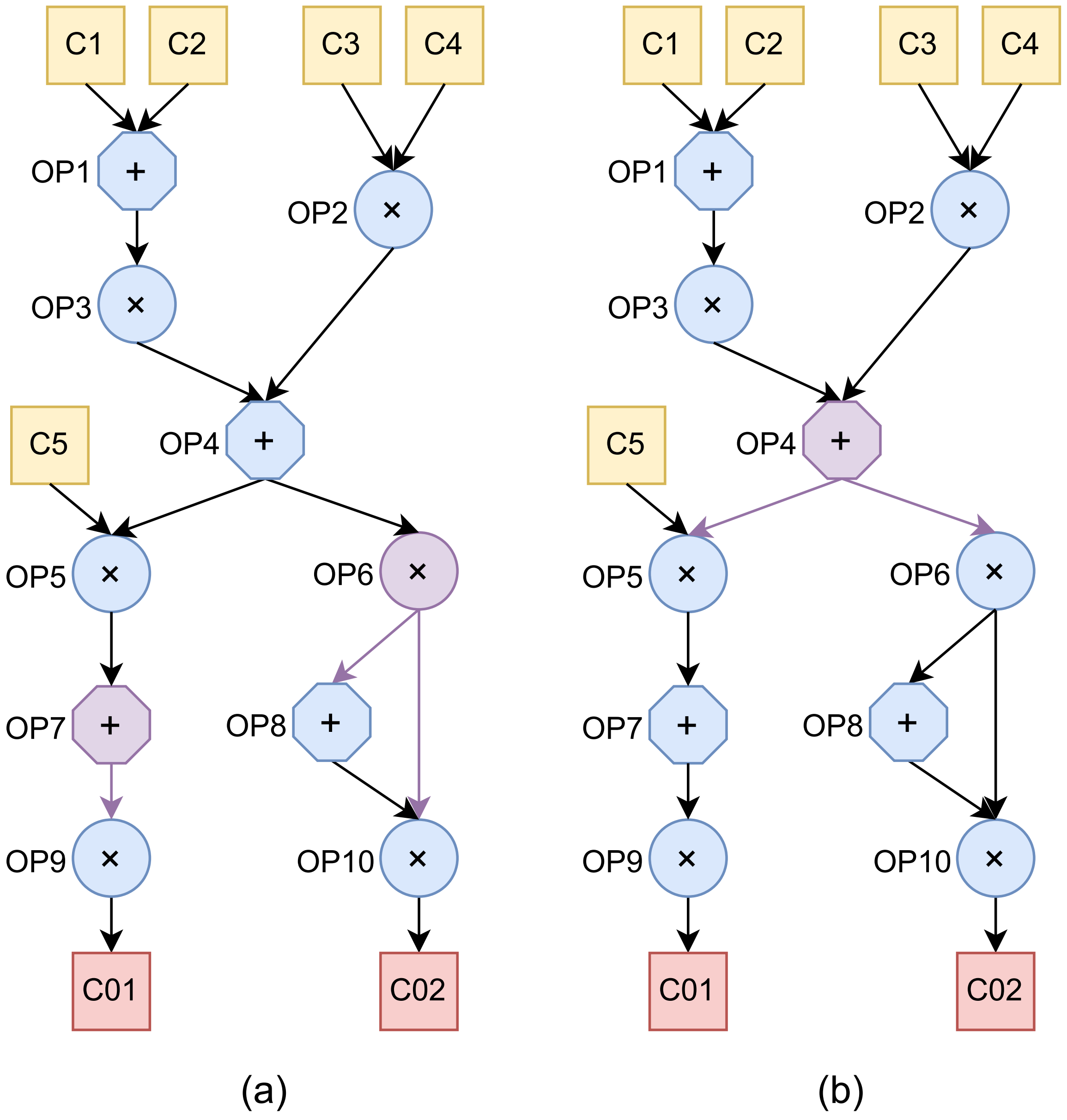


Figure 3.1: FHE task graph and comparison between (a) ALAP and (b) optimal bootstrapping

minimum number of bootstrapping operations and is the optimal solution to the first question raised earlier in this chapter.

3.2 FHE-Booster Framework Overview

Fig. 3.2 presents an overview of the proposed FHE-Booster framework. The parallelograms represent data, while the rectangles represent procedures, which are colored in purple and orange to respectively represent platform-independent and platform-dependent procedures. The primary input to this problem is the FHE program for which bootstrapping will be scheduled. This input is presented in the form of a DAG (e.g., Fig. 3.1), which is the natural encoding for FHE applications [24, 26]. The graph representation includes the input and output ciphertexts, as well as the operations to be performed on each ciphertext. It also needs to incorporate the estimated noise growth of each ciphertext operation, as well as a noise threshold that determines the bootstrapping requirements of the program. Notably, many FHE libraries, such as OpenFHE [1] used in this thesis, are capable of estimating noise growth and computing appropriate thresholds for a given parameter set.

The task graph and the noise information are all the inputs needed to determine after which operations to bootstrap. In other words, the problem of finding the minimum number of bootstrapping points depends only on the FHE application, and not the target hardware platform. Conversely, to create the final FHE execution schedule, the information about the target hardware platform is necessary. Specifically, the number of CPU cores and the execution time of each ciphertext operation (e.g., addition, multiplication, and bootstrapping) on these cores need to be collected. Notably, unlike regular programs whose execution time is typically input-dependent, one important property of FHE programs is that the execution time of each operation *do not depend on the inputs*. As a result, it is highly desirable to use profiling to collect execution time.

As shown in Fig. 3.2, the proposed FHE-Booster provides *three options* for creating the final schedule, which contains the start time and core assignment of each

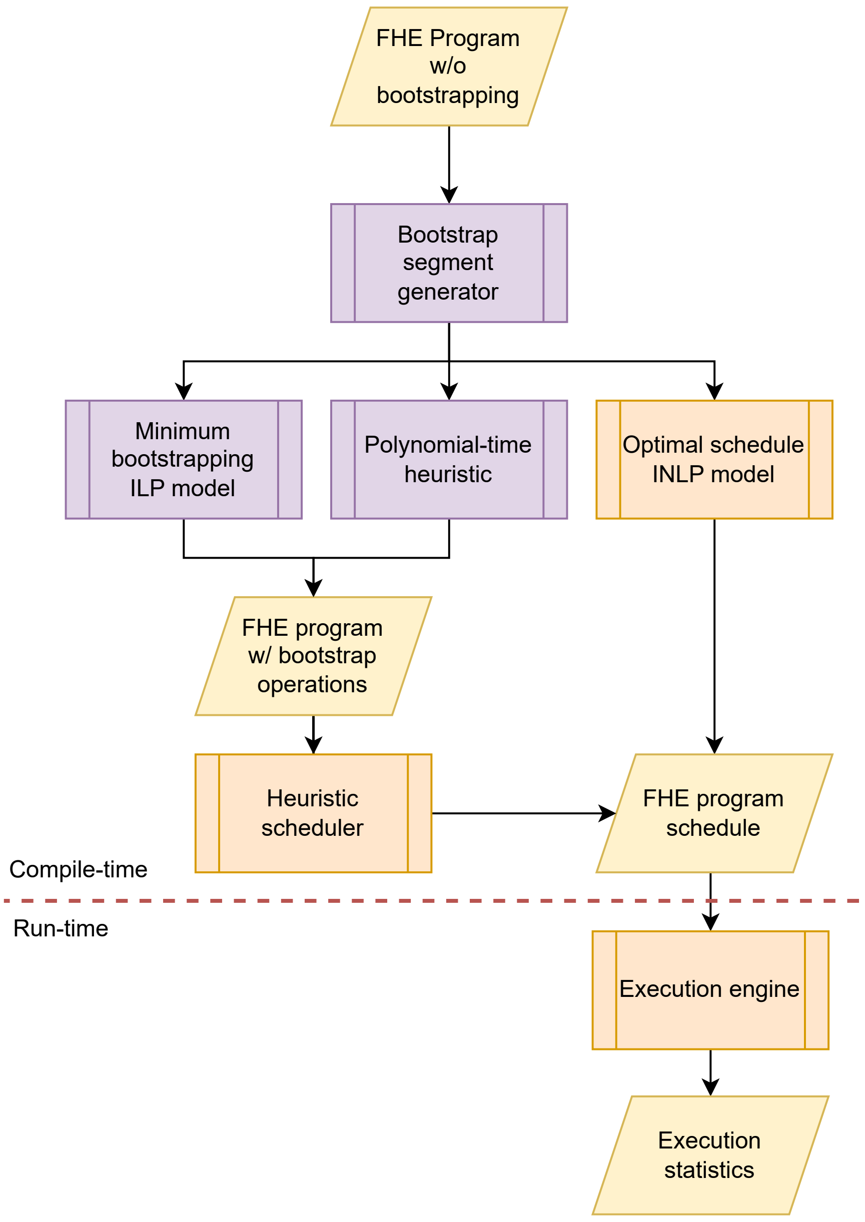

FHE Program w/o bootstrapping
Bootstrap segment generator
Minimum bootstrapping ILP model
Polynomial-time heuristic
Optimal schedule INLP model
FHE program w/ bootstrap operations
Heuristic scheduler
FHE program schedule
Compile-time
Run-time
Execution engine
Execution statistics

Figure 3.2: Overview of FHE-Booster

ciphertext operation and bootstrapping. The first option formulates an integer non-linear programming (INLP) model to produce an optimal schedule. While optimal, this option has no portability across different hardware platforms. In comparison, the other two options first select the operations to bootstrap independent of the hardware, and produce an augmented FHE task graph with bootstrapping operations inserted. Subsequently, the augmented task graph can be scheduled on a given target platform using existing scheduling algorithms. Without loss of generality, FHE-Booster uses a custom heuristic based on *list scheduling* [34]. The main difference between the two hardware-independent options is the strategy for selecting operations to bootstrap. The first option is to formulate an integer linear programming model (ILP) that determines the minimum set of operations that must be bootstrapped for the program to run correctly, while the second option uses a polynomial-time heuristic.

The proposed methodology for formulating bootstrapping constraints, as well as the formal ILP and INLP models used in FHE-Booster are described in Chapter 4, while the heuristics for selecting bootstrapping points are presented in Chapter 5. Finally, the execution engine that implements the generated FHE schedule is described in Chapter 6. Additional technical details are provided in Appendices B and C, which show the custom formats used to encode FHE task graphs and execution engine schedules, repsectively.

Chapter 4
INTEGER PROGRAMMING MODELS

This chapter formulates the bootstrapping scheduling problem mathematically. First, the noise constraints are synthesized to model the bootstrapping requirements. Then, a formal ILP model is developed that determines a *minimum bootstrap set* for a given FHE program, as well as an INLP model that builds on the ILP model to finds the optimal schedule for the FHE program.

4.1 Modeling the Noise Constraints

Fundamentally, the need for bootstrapping comes from the noise threshold. In this thesis's model, every ciphertext has an integer noise level within the range of $[0,\rho]$, where ρ is the noise threshold. Fresh ciphertexts have a noise level of 0. Previous work showed that the noise growth caused by an operation depends only on the type of the operation, not on the input or the target hardware platform [29, 45, 2]. In fact, the noise growth of addition is actually *negligible* compared to multiplication [17, 33]. Based on this fact, the following equation models the noise level of an operation *op*'s output ciphertext *out*, given the input ciphertexts a and b.

$$out.level = \max(a.level, b.level) + \left\{ \begin{array}{lr} 1, & \text{if } op \text{ is a multiplication} \\ 0, & \text{otherwise} \end{array} \right\} \tag{4.1}$$

While bootstrapping can reduce a ciphertext's noise level, it does not bring the level completely down to zero. Instead, there is a constant level to which bootstrapped ciphertexts are reduced. This is called the *bootstrap level*, denoted as l. Same as the noise threshold ρ, the bootstrap level is determined by FHE parameters. From the noise threshold and the bootstrap level, the *level window* t can be calculated as:

$$t = \rho - l \tag{4.2}$$

Level window t can be viewed as the maximum number of multiplication operations that can occur along a path in between two bootstrapping operations. Given this parameter, the proposed method tracks the noise growth along each path in the FHE graph and generates a complete set of "bootstrap segments":

Definition 4.1.1 (Bootstrap Segment)**.** A *bootstrap segment* is a sequence of dependent operations such that the sum of noise growth for all operations on it reaches the given level window t. If none of the output ciphertexts of the operations along the segment are bootstrapped, the noise constraints of the program will be violated. Bootstrapping the output ciphertext of any operation in a bootstrap segment *satisfies that segment*. The noise constraint of the FHE program is satisfied if and only if all bootstrap segments are satisfied.

Assume that for a series of operations s in the graph, the first operation is u_{first}, and the last operations is u_{last}. The following three criteria are used to build the list of bootstrap segments:

1. The segment contains t multiplication operations;

2. u_{last} has at least one multiplication operation as child;

3. u_{first} is a multiplication;

Among the three criteria, Criterion (1) is the most important, since it is sufficient to satisfy all the noise constraints. If the inputs into u_{first} are freshly bootstrapped ciphertexts (i.e. their noise level is the bootstrap level l), and ignoring external inputs to other operations on the segment (i.e. ciphertexts output by operations not on the segment), then a segment of operations that includes exactly t multiplications is guaranteed to have a ciphertext output from u_{last} with a noise level at the threshold ρ.

That ciphertext must be bootstrapped before it can be used as an input to any future multiplication operations.

Although Criterion (1) is sufficient to satisfy noise constraints, it will produce more segments than needed. As a result, the other two criteria are included to remove unnecessary or redundant segments. They do not affect the satisfaction of noise constraints, but improves the efficiency of FHE-Booster. In particular, Criterion (2) prevents the inclusion of unnecessary segments, by enforcing that there exists some multiplication operation that would grow the noise beyond the limit if the segment is not satisfied. Criterion (3) prevents the inclusion of certain "redundant" segments. Since the noise growth of addition is negligible [17, 33], if a segment s starts with an addition operation, removing the addition operation produces s', a shorter segment. As $s' \subset s$, satisfying s' guarantees that s is also satisfied. Therefore, s is redundant and can be ignored.

As an example, consider the task graph shown in Figure 3.1. Assuming a noise threshold of $t = 2$, 18 segments can be found based solely on Criterion (1). Table 4.1 lists those 18 segments, along with the criteria they meet, to help illustrate how bootstrap segments are chosen. Note that only segments $S6$, $S7$, $S8$, $S10$, $S11$, and $S12$ meet all three criteria. Segments $S13-S18$ do not meet Criterion (2) because even though they bring the noise level to the threshold, there are no multiplication operations after $OP9$ or $OP10$ that would grow the noise to exceed the threshold. Similarly, segments $S5$ and $S9$ do not meet Criterion (2) because they only have addition children, so bootstrapping can be delayed to the end of $OP7$, which is not included on those segments. Finally, segments $S1-S4$ and $S13-S15$ do not meet Criterion (3) because they all start with an addition operation.

It is worth mentioning that the three criteria listed here are for freshly bootstrapped ciphertexts with an initial noise level of l. To process inputs with noise level less than l, the list of bootstrapping segments needs to be updated with the policy described in Section 4.2.1.

Table 4.1: Bootstrap segments of Figure 3.1

Segment		Criteria met			Redun-
Label	**OPs**	**1**	**2**	**3**	**dant?**
$S1$	$1, 3, 4, 5$	x			N/A
$S2$	$1, 3, 4, 5, 7$	x	x		N/A
$S3$	$1, 3, 4, 6$	x	x		N/A
$S4$	$1, 3, 4, 6, 8$	x	x		N/A
$S5$	$2, 4, 5$	x		x	N/A
$S6$	$2, 4, 5, 7$	**x**	**x**	**x**	No
$S7$	$2, 4, 6$	**x**	**x**	**x**	No
$S8$	$2, 4, 6, 8$	**x**	**x**	**x**	Yes
$S9$	$3, 4, 5$	x		x	N/A
$S10$	$3, 4, 5, 7$	**x**	**x**	**x**	No
$S11$	$3, 4, 6$	**x**	**x**	**x**	No
$S12$	$3, 4, 6, 8$	**x**	**x**	**x**	Yes
$S13$	$4, 5, 7, 9$	x			N/A
$S14$	$4, 6, 10$	x			N/A
$S15$	$4, 6, 8, 10$	x			N/A
$S16$	$5, 7, 9$	x		x	N/A
$S17$	$6, 10$	x		x	N/A
$S18$	$6, 8, 10$	x		x	N/A

4.2 Algorithm for Finding Bootstrap Segments

This thesis employs a dynamic programming algorithm to efficiently generate a list of all the segments meeting the three criteria described before. The algorithm is built upon a function $segs(op, t)$ which returns all the bootstrapping segments starting from the operation op, assuming a level window of t. The function $segs(op, t)$ depends on several sub-functions, most importantly $allsegs(op, t)$, which includes the same segments as $segs$, but Criterion (3) is ignored, so that any type of operation may start a segment. This is shown in the following equation:

$$segs(op, t) = \left\{ \begin{array}{ll} \emptyset, & \text{if } op \text{ is not multiplication OR is pre-bootstrap} \\ allsegs(op, t), & \text{otherwise, with each segment reversed} \end{array} \right\} \quad (4.3)$$

The adoption of dynamic programming is based on the key observation that segments beginning with a certain operation are related to the segments starting with that operation's children (called *subsegments*). For segments to properly build upon subsegments, Criterion (3) that requires a segment to start with a multiplication must be ignored. In our algorithm implementation, this criterion is ignored when building $allsegs(op, t)$, but enforced when finally constructing $segs(op, t)$, as shown in Equation (4.3). Note that $segs(op, t)$ function also returns the empty set if op is "pre-bootstrap," a condition that will be discussed later in this chapter (Section 4.2.1).

The definition of $allsegs(op, t)$ depends on the value of t. The base case is when $t < 1$ and it returns the empty set. When $t \geq 1$, $allsegs(op, t)$ will equal the union of two subfunctions, namely, $subsegs(op, t)$ and $solo(op, t)$, as shown below:

$$allsegs(op, t) = \left\{ \begin{array}{ll} \emptyset, & \text{if } t < 1 \\ subsegs(op, t) \ \cup \ solo(op, t), & \text{if } t \geq 1 \end{array} \right\} \quad (4.4)$$

The function $subsegs(op, t)$ collects the segments beginning with any of op's children operations, and appends op to the beginning of each:

$$subsegs(op, t) = \cup_{c \in op.children} [\cup_{s \in allsegs(c, r_c, t)} [\{op\} + s]] \quad (4.5)$$

As shown, $subsegs(op, t)$ is computed in three steps. First, for every child operation c of operation op, the *remaining level window* $r_{c,t}$ is determined by the type of c:

$$r_{c,t} = \left\{ \begin{array}{ll} t-1, & \text{if } c \text{ is a multiplication} \\ t, & \text{otherwise} \end{array} \right\} \tag{4.6}$$

Then, for that child operation c, each of its segments from $allsegs(c, r_{c,t})$ are appended to include op at the beginning. Finally, all of these newly-expanded segments are combined into one set of segments to form $subsegs(op, t)$. This creates a recursive definition of $allsegs(op, t)$, and the use of $r_{c,t}$ helps enforce Criterion (1) by finding subsegments with a shorter level window every time a multiplication is added to the segment. This ensures that $allsegs(op, t)$ only returns segments with t multiplication operations.

The second subfunction used in Eq. (4.4) is $solo(op, t)$, which returns the empty set in most cases. The only exception is that if any of op's children has its remaining level window $r_{c,t} = 0$, then $solo(op, t)$ returns a set with one segment containing only op. This is the case when $t = 1$ and op has at least one child operation that is a multiplication.

$$solo(op, t) = \left\{ \begin{array}{ll} \{\{op\}\}, & \text{if any of } op\text{'s children } c \text{ have } r_{c,t} = 0 \\ \emptyset, & \text{otherwise} \end{array} \right\} \tag{4.7}$$

The function $solo(op, t)$ enforces Criterion (2), since it is the only function that actually *creates* segments, compared to $allsegs(op, t)$, which only *modifies* pre-existing segments. Specifically, it creates a segment with only one operation, and that operation has a multiplication child. Therefore, since *allsegs* only adds operations to the *front* of segments, all segments will end with an operation that has a multiplication child.

Algorithm 1 presents a pseudo code that uses this dynamic programming scheme to generate the bootstrap segments. Initially each segment is stored in reverse order. Each segment is only returned to its expected order when added to the final output at the end. The reason for this reversing is because the segments are implemented in

Algorithm 1: Finding Bootstrap Segments

```
input  : $G$ - An FHE program task graph
         $t$ - The level window
output: $S$ - A set of lists of operations, each representing a bootstrap
         segment

for Each operation $op$ in reverse topological order do
    $allsegs[op, 0] \longleftarrow \emptyset$
    AddSubSegs($op, 1$)
    if OpHasMultChild($op$) then
        $allsegs[op, 1].add(\{op\})$
for $i \in \{2, ..., t\}$ do
    for Each operation $op$ in reverse topological order do
        AddSubSegs($op, i$)
for Each operation $op$ do
    if $op$ is a multiplication AND $op$ is not pre-bootstrap then
        $S.add$(all segments in $allsegs[op, t]$, each reversed)
return $S$

Function AddSubSegs($op, i$):
    for $child \in op.children$ do
        if $child$ is a multiplication then
            $r \longleftarrow i - 1$
        else
            $r \longleftarrow i$
        for $s \in allsegs[child, r]$ do
            $allsegs[op, i].add(s + \{op\})$
```

C++ as vectors for efficiency, and it is only efficient to insert to the back of a vector (not the front).

While Algorithm 1 enforces the three criteria mentioned before, it may still produce some redundant segments. Two examples are shown Table 4.1, specifically, segments $S8$ and $S12$. These redundant segments are intentionally generated because they are needed when considering the "selective forwarding" option explained in Section 5.3. Nonetheless, removing these redundant segments has the benefit of reducing the complexity of the minimum bootstrapping linear programming model described in this chapter. For this reason, Algorithm 2 shows the concept of finding these redundant

Algorithm 2: Remove Redundant Bootstrap Segments (Conceptual)

```
input  : $S_{in}$ - A set of lists of operations, each representing a bootstrap
         segment
output: $S$ - $S_{in}$ with redundant segments removed, sorted by segment size

$S \longleftarrow S_{in}$, sorted by segment size
$i \longleftarrow 1$
while $i \leq$ Size($S$) do
    $j \longleftarrow i + 1$
    while $j \leq$Size($S$) do
        if $S_i$ is a subset of $S_j$ then
            $S.remove(S_j)$
        else
            $j \longleftarrow j + 1$
    $i \longleftarrow i + 1$
```

segments and removing them, while Algorithm 3 shows a more efficient implementation, which takes advantage of the sorting of the segment set produced by Algorithm 1, as well as a special property of the redundant segments produced by Algorithm 1. Specifically, a redundant segment s and the shorter segment s' that makes s redundant will share the same u_{first} and m_{last}, where m_{last} is the last multiplication operation in a segment. The following reasons explain why this is the case.

- s' must contain every operation in s, including the same t multiplication operations.
- Since the FHE program is a DAG, the operations shared between s and s' must occur in the same order in both segments.
- Since Criterion (3) requires that all bootstrap segments begin with a multiplication operation, u_{first} of the two segment is the same.

The algorithm begins by performing some further sorting so that the segments can be traversed efficiently while checking for redundancy.

Algorithm 3: Remove Redundant Bootstrap Segments (Efficient)

```
input  : $S_{in}$ - A set of lists of operations, each representing a bootstrap
         segment, sorted by the first operation in the segment
output: $S$ - $S_{in}$ with redundant segments removed, now sorted by first
         operation, then last operation, then segment size

$S \longleftarrow S_{in}$
$offset \longleftarrow 1$
$current \longleftarrow S_1.u_{first}$
$i \longleftarrow 2$
while $i <$ Size($S$) do
    if $S_i.u_{first} != current$ then
        Sort $S_{current}$ to $S_i$ by $m_{last}$, then by size
    $i \longleftarrow i + 1$
Sort $S_{current}$ to $S_i$ by $m_{last}$, then by size

$i \longleftarrow 1$
while $i \leq$ Size($S$) do
    $j \longleftarrow i + 1$
    while $j \leq$ Size($S$) && Size($S_i$) != Size($S_j$) &&
    $S_i.u_{first} = S_j.u_{first}$ && $S_i.m_{last} = S_j.m_{last}$ do
        if SegmentsAreRedundant($S_i$, $S_j$) then
            $S.remove(S_j)$
        else
            $j \longleftarrow j + 1$
    $i \longleftarrow i + 1$

Function SegmentsAreRedundant($s_{small}, s_{large}$):
    $i \longleftarrow 1$
    $j \longleftarrow 1$
    for $i \leq$ Size($s_{small}$) do
        while $s_{small}[i] \neq s_{large}[j]$ do
            $j \longleftarrow j + 1$
            if $j \geq$Size($s_{large}$) then
                return false
        $i \longleftarrow i + 1$
    return true
```

4.2.1 Pre-bootstrap operations

One special consideration is that ciphertexts typically have much less noise when they are "fresh" (i.e., immediately after being generated) than they do after being bootstrapped. This is because bootstrapping only reduces the ciphertext noise to a much lower level, but cannot fully reset it to the original level of a fresh ciphertext. This difference can be modeled by ignoring a certain subset of the earliest operations in the FHE program when creating the bootstrap segments. The correct subset to ignore is the maximum size set of operations that, once completed, produce ciphertexts with noise levels at most equal to the bootstrap level l. A dynamic programming algorithm is able to find these "pre-bootstrap" operations. The algorithm relies on a function $tlp(op, l)$, which returns $true$ if there exists some path from an operation to the top of the graph that includes more than l multiplications, and *false* otherwise. One base case is when $l < 0$, in which case tlp returns $true$.

$$tlp(op, l < 0) = true \tag{4.8}$$

Two more base cases are defined when op has no parent operations: tlp returns true if $r_{op,l} < 0$, false if $r_{op,l} \geq 0$. Note that $r_{op,l}$ is defined the same as in Equation (4.6), though in this case it is the "remaining bootstrap level" as opposed to the "remaining level window".

$$tlp(op \mid op \text{ has no parents}, l \geq 1) = \left\{ \begin{array}{ll} true, & \text{if } r_{op,l} < 0 \\ false, & \text{if } r_{op,l} \geq 0 \end{array} \right\} \tag{4.9}$$

Lastly, in all other cases, $tlp(op)$ is computed by performing an OR operation over all of op's parents' tlp functions:

$$tlp(op \mid op \text{ has parents}, l \geq 1) = OR_{p \in op.parents}[tlp(p, r_{op,l})] \tag{4.10}$$

Algorithm 4 shows how the tlp values are calculated efficiently. It first computes tlp values for all operations in the FHE task graph. At the end, an operation is considered pre-bootstrap if $tlp(op, l)$ is false, meaning there is no path from that operation to the top of the graph with l or more multiplication operations.

Algorithm 4: Find Pre-Bootstrap Operations

```
input  : $G$ - An FHE program task graph
         $l$ - The bootstrap level
output: $I$ - A set of pre-bootstrap operations

for Each operation $op$ in $G$ do
    $tlp[op, -1] \longleftarrow true$
for $i \in \{0, ..., l\}$ do
    for Each operation $op$ in $G$ do
        if $op$ is multiplication then
            $r \longleftarrow (i - 1)$
        else
            $r \longleftarrow i$
        if $op$ has no parent operations then
            $tlp[op, i] \longleftarrow (r < 0)$
        else
            $tlp[op, i] \longleftarrow OR_{p \in op.parents}\{tlp[p, r]\}$
for Each operation $op$ in $G$ do
    if $tlp[op, l] = false$ then
        $I.add(op)$
return $I$
```

4.2.2 Complexity analysis

This section analyzes the complexity of the algorithms presented in this chapter.

Algorithm 1. Assuming a bootstrap level of l and an FHE task graph with L DAG levels[1], the maximum length of a bootstrap segment is $\lambda = L - l$. Considering that each operation may have at most two parents, backtracking from an operation shows that, at most, 2^λ segments can end with that operation. Therefore, the number of bootstrap segments in an FHE task graph is $O(n \cdot 2^\lambda)$, where n is the number of operations in the task graph. The sum of all segment lengths is $O(n \cdot \lambda \cdot 2^\lambda)$.

The worst case complexity of Algorithm 1 can be determined by analyzing the second of its three loops, since it dominates the runtime. All the code inside that loop executes, in the worst case, $2s$ iterations, where s is the sum of the lengths of

[1] The number of levels in an FHE DAG is equivalent to the length of longest path from the inputs to the outputs.

all bootstrap segments the program contains. This is because for each operation, the algorithm iterates over each of its children. Since operations have at most 2 parents, each operation is visited at most twice. Then, for each of those visits, all of the operation's segments are copied (with one operation appended to the end) to build one of the parents' segments. Therefore, every element of every segment (for that t value) is visited at most twice. This justifies that the code inside the second for loop is $O(2s)$. Since that code is inside a for loop with $t - 1$ iterations and since s is $O(n \cdot \lambda \cdot 2^{\lambda})$, the entire worst case complexity is $O(t \cdot n \cdot \lambda \cdot 2^{\lambda})$. Exponential worst case complexity imposes a challenge not only to computation time, but also to memory utilization, since every segment needs to be stored in memory. Fortunately, some optimizations can be used to address this issue, which will be discussed in Section 8.2.

Algorithms 2 and 3. When analyzing the complexity of these algorithms, the previously generated bootstrap segment set, instead of the FHE task graph, is the input. Assuming there are k segments and the sum of all segment lengths is s, the complexity of Algorithm 2 is $O(k \cdot s)$. Algorithm 3 is generally more efficient because it operates over portions of the segment set at a time. Specifically, assume the segments begin with c different operations, the maximum number of segments starting with any one operation is k', and the maximum sum of segment lengths starting with any one operation is s', then the complexity is $O(c \cdot k' \cdot s')$. Testing of the two algorithms showed that Algorithm 3 is at least as fast as Algorithm 2 for the tested programs, but usually faster.

Algorithm 4. The nested loops dominate the complexity of this algorithm. The code inside the loops runs in $O(1)$ time, since operations have at most two parents. Therefore, the complexity of Algorithm 4 is $O(l \cdot n)$, where again l is the bootstrap level and n is the number of operations in G.

4.3 Minimum Bootstrapping ILP Model

Table 4.2 summarizes the list of variables used in this model. As shown, a set of binary variables b_i is used to represent if Task i is bootstrapped or not. The only

Table 4.2: Variables used in the IP models

Variables used in minimum bootstrapping model	
G	The directed acyclic graph (DAG) of the target FHE program
n	The total number of tasks in G
S	The set of bootstrap segments of the program
p	The total number of segments in S
S_k	The k^{th} segment of S
b_i	$= 1$ if Task i is bootstrapped; $= 0$ otherwise

Extra variables used in optimal schedule model	
$b_{i,j}$	$= 1$ if Task i is bootstrapped and sends its bootstrapped result to its dependent Task j; $= 0$ otherwise.
c	The number of cores available for bootstrapping
exe_i	Execution time of Task i
exe_b	Execution time of bootstrapping
ST_i	Start time of Task i
FT_i	Finish time of Task i
BFT_i	Finish time of bootstrapping task i's output
$T2C_{i,k}$	$= 1$ if Task i runs on core k; $= 0$ otherwise
$Order_{i,j}$	$= 1$ if Task i and Task j are on the same core and i runs before j; $= 0$ otherwise

set of constraints needed for this model is that each bootstrap segment S_k is satisfied, which entails that at least one task i on S_k is bootstrapped. These constraints are summarized mathematically as:

$\forall\ 1 \leq k \leq p:$

$$\sum_{i \in S_k} b_i \geq 1. \tag{4.11}$$

The objective function is to minimize the total number of bootstrapped operations:

$$\min \sum_{i=1}^{n} b_i. \tag{4.12}$$

The key observation is that this problem is equivalent to the *set cover* problem – finding the minimum set of bootstrapping points that cover all the bootstrap segments. It can be solved optimally by ILP, or by any heuristic developed for the set cover problem.

4.4 Optimal Schedule INLP Model

Unlike the minimum bootstrapping model, the optimal bootstrapping scheduling model is significantly more complicated, as it needs to model and constrain not only bootstrapping, but also task-to-core binding, task execution order, and task start/finish time. The following paragraphs discuss these four aspects one by one. Same as before, the list of variables used in this model are summarized in Table 4.2.

Unlike the minimum bootstrapping model that only tracks whether a task needs to be bootstrapped or not, in an optimal schedule, even if a task needs to be bootstrapped, some of its child tasks may not need the bootstrapped result and hence can benefit from an earlier start time. Based on this observation, in the optimal schedule model, a 2-dimensional binary matrix $b_{i,j}$ is used to represent whether Task i needs to be bootstrapped and the result needs to be sent to a child Task j. Note that $b_{i,j}$ is only defined for a pair of dependent tasks (i, j).

Using this representation, the bootstrapping constraints can be modeled similar to the minimum bootstrapping model. The following constraint ensures that on every

bootstrap segment S_k, at least one task i is bootstrapped, and the subsequent task $i+1$ on S_k receives its bootstrapped results:
$\forall\ 1 \leq k \leq p:$

$$\sum_{i,i+1 \in S_k} b_{i,i+1} \geq 1. \tag{4.13}$$

It is important to note that the bootstrap segments in this model are slightly modified from those described in Sections 4.1, which are those used in the minimum bootstrapping ILP model. Specifically, for each standard segment s with final operation u_{last}, this model augments it and has a segment for each of u_{last}'s children, each of which is simply s appended with that child. These are called "selective" segments.

In addition to the bootstrapping constraint, it is also necessary to define and model the start and finish times of each task. Here it is assumed that scheduling is *non-preemptive*, which means a task being executed cannot be preempted by another task. Moreover, each task has a single execution time, as the target platform is a homogeneous multi-core system. (Note that this assumption can be easily extended to model heterogeneous multi-core systems by using a 2-dimensional array $exe_{i,k}$ to represent the execution time of task i on core k.) Based on these assumptions, the finish time of a task is the sum of its start time and its execution time:
$\forall\ 1 \leq i \leq n:$

$$FT_i = ST_i + exe_i. \tag{4.14}$$

The start time of a child task j depends on whether its parent task i sends to it the original or a bootstrapped result. In the following equation, the bootstrapping latency exe_b is added to FT_i to constrain the start time of the child task j if and only if $b_{i,j}$ is 1:
$\forall\ (i,j)$ dependencies:

$$ST_j \geq FT_i + exe_b \times b_{i,j}. \tag{4.15}$$

Meanwhile, to ensure each task runs on exactly one core, another 2-dimensional binary matrix $T2C_{i,k}$ is used to model the binding of task i to core k and apply the following constraint:

$\forall\ 1 \leq i \leq n:$

$$\sum_{k=1}^{c} T2C_{i,k} = 1. \tag{4.16}$$

For tasks scheduled on the same core, a later task j cannot start until a previous task i finishes. Since task i may need to be bootstrapped by that core, the following equation computes the finish time of bootstrapping task i's result:

$\forall\ 1 \leq i \leq n:$

$$BFT_i = FT_i + exe_b \times OR_{j=1}^{n} b_{i,j}. \tag{4.17}$$

Here, Eq. (4.17) conditionally adds the latency of bootstrapping exe_b to the finish time of task i. Specifically, if task i's output needs to be bootstrapped, at least one $b_{i,j}$ equals 1, and the OR operation $OR_{j=1}^{n} b_{i,j}$ will produce a 1. On the other hand, if task i does not need to be bootstrapped, $OR_{j=1}^{n} b_{i,j}$ will be 0. Using BFT_i, the start time of task j can be constrained:

$\forall\ (i, j)$ where $1 \leq i \leq n, 1 \leq j \leq n,$ and $i \neq j:$

$$ST_j \geq BFT_i \times Order_{i,j}. \tag{4.18}$$

Eq. (4.18), uses a 2-dimensional binary matrix $Order_{i,j}$ to ensure that tasks scheduled on the same core do not overlap. $Order_{i,j}$ is 1 if and only if tasks i and j are on the same core and i runs before j. The following constraint further ensures that, when two tasks i and j are on the same core, one must run before the other:

$\forall\ (i, j)$ where $1 \leq i \leq n, 1 \leq j \leq n,$ and $i \neq j:$

$$Order_{i,j} + Order_{j,i} = \sum_{k=1}^{c} T2C_{i,k} \times T2C_{j,k}. \tag{4.19}$$

If tasks i and j are scheduled on different cores, the variables $T2C_{i,k}$ and $T2C_{j,k}$ will not be 1 for the same value of k. Thus, the right-hand side of Eq. (4.19) is 0, and both $Order_{i,j}$ and $Order_{j,i}$ are 0. On the other hand, if tasks i and j are scheduled on the same core, the right-hand side of Eq. (4.19) is 1, and one of $Order_{i,j}$ and $Order_{j,i}$ is 1, while the other is 0.

Eq. (4.13)–Eq. (4.19) together constrain the optimal bootstrapping scheduling problem. The objective function of this problem is to minimize the *schedule length*, defined as the maximum of the finish times of all tasks in the FHE program:

$$\min\{\max_{i=1}^{n} FT_i\}. \tag{4.20}$$

While this model produces an optimal schedule, it contains a significant number of variables in the search space, so the cost to find the optimal solution can be prohibitive even for somewhat small FHE programs. Furthermore, even this complicated model incorporates certain simplifications – a fully optimized schedule would need to model inter-core communication time and memory transfer overheads as well. Therefore, for realistic FHE applications, heuristic-based scheduling methods are often preferable, as described next.

Chapter 5
HEURISTIC-BASED BOOTSTRAPPING SCHEDULING

The heuristics discussed in this chapter select operations to bootstrap. As mentioned in Chapter 3, this will produce an augmented FHE task graph with bootstrapping operations inserted. The problem of mapping the augmented task graph to a multi-core platform is a classic scheduling problem that can be solved by existing schedulers, such as genetic algorithms and list scheduling. Without loss of generality, FHE-Booster uses a custom heuristic based on list scheduling. List scheduling generally works by assigning a priority to each operation to be executed, and when a processor becomes available, the highest priority operation whose dependencies have completed is assigned to that processor. This thesis chooses to assign priority based on operation slack. Operations with the least slack between their earliest and latest start times are given highest priority. For operations with the same slack, operations that are topologically earlier are given higher priority.

5.1 Minimum Bootstrapping

This method chooses the operations to bootstrap by finding the minimum set of operations that satisfy all the bootstrap segments defined in Section 4.1. As mentioned before, this is equivalent to the *set cover* problem. For somewhat small FHE programs, the minimum bootstrap set can be found in a reasonable amount of time using an ILP solver. For larger FHE programs, an ILP solver may take significantly longer to find the optimal solution.

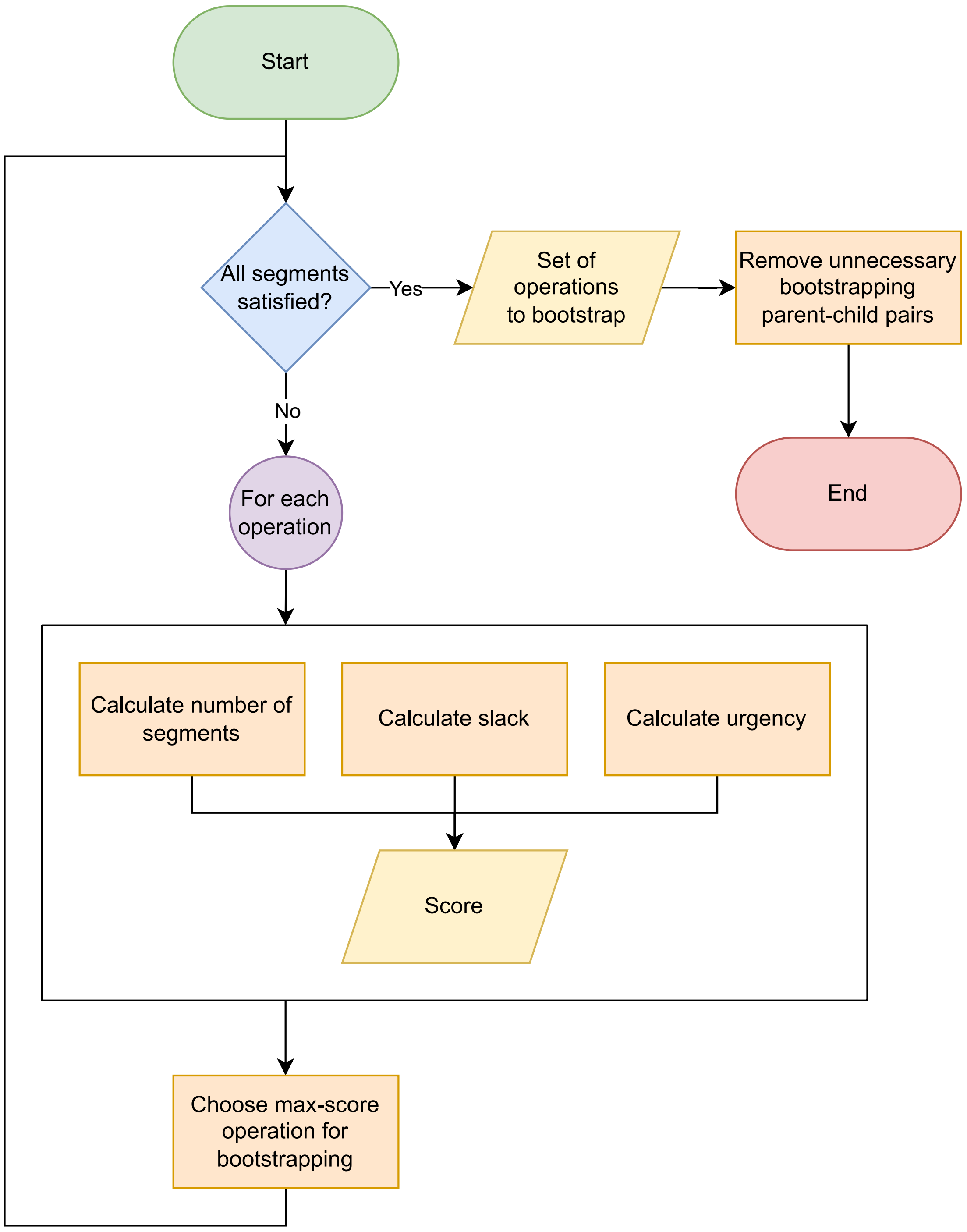

Start
All segments satisfied?
Yes
Set of operations to bootstrap
Remove unnecessary bootstrapping parent-child pairs
End
No
For each operation
Calculate number of segments
Calculate slack
Calculate urgency
Score
Choose max-score operation for bootstrapping

Figure 5.1: The flow of the score-based heuristic (© 2023 IEEE)

5.2 Score-Based Heuristic

The idea behind this heuristic is to assign a score to each operation, and iteratively select the operation with the highest score for bootstrapping until all bootstrap segments are satisfied. Figure 5.1 presents a flow chart of the heuristic, which iteratively selects an operation for bootstrapping until all bootstrap segments are satisfied. In each iteration, a score is calculated for every operation and the operation with the highest score (that is not already chosen) is chosen for bootstrapping. If multiple operations have the maximum score, the one with the lowest ID is chosen, with lower IDs correlated to earlier topological order. Finally, a separate function is invoked so that select operations may use the unbootstrapped results from parent operations that are bootstrapped.

The score combines *three attributes*: bootstrapping urgency, number of unsatisfied bootstrap segments, and operation slack. The rest of this section introduces these attributes one by one, and then presents the score equation for combining them.

5.2.1 Bootstrapping urgency

This heuristic chooses which operations to bootstrap based on how "urgent" the need is. Specifically, in each iteration a set of bootstrap segments are considered *alive*, and the last operations on each of the alive segments are given the highest weight. A segment is *alive* if (1) it has not yet been satisfied and (2) its last operation is "reachable", that is, the first operation on the segment either has no parent, or each of its parent has been bootstrapped or belongs to no bootstrap segments.

On a segment S_k of length l, an operation op at index $i \in \{1, ..., l\}$ of S_k is given the following urgency value:

$$v_{op,S_k} = \left\{ \begin{array}{ll} i/l, & \text{if } S_k \text{ is alive,} \\ 0, & \text{if } S_k \text{ is not alive.} \end{array} \right\} \tag{5.1}$$

As op may be on multiple segments, its overall urgency is the maximum across all segments:

$$urgency_{op} = \max_{S_k \,:\, op \in S_k} v_{op,S_k}. \tag{5.2}$$

Note that upon selecting an operation for bootstrapping, a new set of segments may become alive. Therefore, operations on those segments will have their urgency value updated.

When used alone, this heuristic may produce the same bootstrap set as ALAP. However, because the bootstrapping points are known prior to performing heuristic scheduling, the scheduler has more information to better schedule the operations as compared to the ALAP scheduler, which generates the bootstrap set dynamically. Meanwhile, this attribute can be used in combination with the other attributes defined in this chapter to create schedules very different from ALAP.

5.2.2 Number of unsatisfied bootstrap segments

This is a greedy heuristic designed to approximate the minimum bootstrapping algorithm, by satisfying as many bootstrap segments as possible with a single operation. Specifically, for each operation, this attribute is calculated as the number of remaining unsatisfied bootstrap segments to which the operation belongs. Upon selecting an operation for bootstrapping and thus satisfying a set of segments, this attribute is recalculated for the other operations on those segments.

5.2.3 Operation slack

This heuristic aims to hide the bootstrapping latency as much as possible. For each operation, its *slack* is defined as the interval between its latest start time and earliest start time. By selecting operations with more slack to bootstrap, the time spent in bootstrapping will have lower impact to the overall execution time of the program. As a concrete example, if an operation has two parenting tasks and one of them has been chosen for bootstrapping, the slack of the other parenting tasks will increase significantly. If there is an idle core available, bootstrapping the second parenting task could be more efficient in the end.

5.2.4 Overall score

With the three attributes defined as above, the overall score of an operation can be calculated with the following equation:

$$score = \left\{ \begin{array}{ll} -1, & \text{if } segments = 0 \\ s \times segments + r \times slack + u \times urgency, & \text{otherwise} \end{array} \right\} \tag{5.3}$$

Here, the set of weights (s, r, u) applied to the three attributes can be freely defined by the user. Setting the score to -1 when the operation belongs to no unsatisfied segments ensures operations are not chosen for bootstrapping when doing so will not help satisfy noise constraints. Note that all the attributes are normalized to be in the range [0,1]. This can be done by dividing each attribute by its maximum value across all the operations.

5.3 Selective Bootstrapping Conversion

In addition to the heuristic used for selecting bootstrapping points, this thesis also considers another optimization to selectively forward the bootstrapped results to a dependent task. This idea is also used in the optimal schedule model of Section 4.4. As mentioned before, even if a task needs to be bootstrapped, some of its child tasks may not need the bootstrapped result. These tasks do not need to wait for the bootstrapping operation to complete and hence can benefit from an earlier start time. Motivated by this observation, the conceptual Algorithm 5 was developed for creating a *selective* bootstrap set that specifies the child tasks that should receive the bootstrapped ciphertext, for each bootstrapped operation. The algorithm checks each parent-child bootstrapping pair, and removes a pair only if none of the segments on which it is located require that pair to be satisfied.

Algorithm 6 shows a more complete and more efficient implementation. The first loop converts the set of complete-forwarding bootstrap segments to selective-forwarding segments. The next two loops find a set of "candidate" parent-child pairs that may be able to be removed from the bootstrap set. This is done by traversing each selective bootstrap segment and finding the parent-child pairs that satisfy it. When multiple

Algorithm 5: Selective Bootstrapping Conversion (Conceptual)

```
input  : $S$ - Selective-forwarding bootstrap segments
         $P_{in}$ - A set of parent-child operation pairs for complete forwarding
output: $P$ - A set of parent-child operation pairs for selective forwarding

$P \longleftarrow P_{in}$
for $pair \in P$ do
    if No segment in $S$ relies on $pair$ then
        $P.remove(pair)$
```

pairs satisfy a segment S_i, each *pair* is used as a key to the C map, with the index i being added to the list of indicies at $C[pair]$. When only a single pair satisfies a segment, it is considered "needed" and is added to the N set. In this way, all pairs in the N set can finally be removed as a candidate from the C map. At that point, any one parent-child pair in C could be removed from the bootstrap set without violating the program's noise constraints. However, removing one pair may make another pair in C necessary to bootstrap to satisfy the noise constraints. Therefore, the final loop essentially implements the conceptual Algorithm 5, but instead of considering every parent-child pair, it only needs to consider those pairs used as keys in the C map. Furthermore, for each of those pairs, only the associated bootstrap segment indicies need to be checked to see if there are any segments that rely on that pair after some other parent-child pairs have been removed from the set.

A direct implementation of the conceptual algorithm would result in a complexity of $O(m \cdot s)$, where m is the number of edges in the task graph and s is the sum of the lengths of all the bootstrap segments. In contrast, the efficient algorithm has a complexity of $O(s + p \cdot s')$ where p is the number of pairs in C, and s' is the sum of the lengths of the bootstrap segments whose indicies are stored in C.

Note that the assumptions of a complete bootstrap set, which considers bootstrapping in terms of single operations, are different from the assumptions of a selective bootstrap set, which considers bootstrapping in terms of parent-child operation pairs. Therefore, when the selective forwarding method is used, the output of Algorithm 1 is used directly, without using Algorithm 3 to remove redundant segments.

Algorithm 6: Converting to Selective Bootstrapping (Efficient)

```
input  : $S_{in}$ - Complete-forwarding bootstrap segments (the direct output
         of Algorithm 1)
         $B$ - A set of operations to bootstrap w/ complete forwarding
output: $P$ - A set of parent-child operation pairs for selective forwarding

$S \longleftarrow \{\}$
for $s \in S_{in}$ do
    for $child \in s.u_{last}.children$ do
        $S.add(s + \{child\})$

$C \longleftarrow \{\}$
$N \longleftarrow \{\}$
for $i \in \{1, ...,$Size($S$)$\}$ do
    $SP \longleftarrow$ GetSatisfyingPairs($S_i, P$)
    if Size($SP$) $> 1$ then
        for $pair \in SP$ do
            $C[pair].add(i)$
    else
        for $pair \in SP$ do
            $N.add(pair)$
for $pair \in N$ do
    $C.remove(pair)$
return $C$

for $\{pair, indicies\} \in C$ do
    if PairNotNeeded($P, C, pair, indicies$) then
        $P.remove(pair)$

Function PairNotNeeded($P, C, pair, indicies$):
    for $i \in indicies$ do
        $s \longleftarrow S_i$ if SegmentReliesOnPair($P, s, pair$) then
            return false
    return true

Function SegmentReliesOnPair($P, s, pair$):
    for $i \in \{1, ...,$Size($s$)$-1\}$ do
        if
          ($pair.parent\ != s_i$ || $pair.child\ != s_{i+1}$) && $P.contains(\{s_i, s_{i+1}\})$
          then
            return false;
    return true
```

Chapter 6

EXECUTION ENGINE AND TESTING

The scheduling heuristic outlined in Chapter 5 produces an assembly-like text file describing a multi-core schedule of operations. The format of such a schedule is described in Appendix C. Each operation has an identifier outlining the exact worker thread for running the operation. This schedule is then processed by our novel FHE execution engine, called *FHE-Runner*, which is developed as part of the FHE-Booster framework. FHE-Runner is implemented in C++ using the OpenFHE library implementation of CKKS [1]. The supported operations are addition, subtraction, multiplication, negation, and bootstrapping (ADD, SUB, MUL, INV, and BOOT). For the operations that take two inputs (i.e., ADD, SUB, MUL), both ciphertext-ciphertext and ciphertext-plaintext operations are supported.

FHE-Runner parses the schedule and creates a queue of encrypted operations for each worker thread. Each queue reflects a correct execution order determined by the scheduling algorithm. Next, each worker thread will pop operational assignments from its corresponding queue and execute them one by one until the queue is empty. Once all the queues are completed, the encrypted results (i.e., the output nodes) are collected and returned to the user for post-processing.

A static single-assignment form (SSA) is used to simplify cross-thread synchronization. By enforcing SSA, and since all FHE programs are directed acyclic graphs, a *post/wait*-style synchronization scheme can be utilized for a task to post its result and send it to the dependent tasks that are waiting for it. To implement such a scheme, *mutexes* (mutual exclusion objects) are used. Any ciphertext that is an output of a certain operation has an associated mutex, which is locked upon creation. Then, when executing the provided schedule, each operation must lock and immediately unlock any mutex associated with its inputs (thus implementing the *wait* function) before performing the operation, and unlock the mutex associated with its output (thus implementing the *post* function) upon completion of the operation. Compared to FHE operations, the overhead introduced by the lock/unlock functions is negligible.

As a comparison baseline, the FHE-Runner engine also supports the *as-late-as-possible* (ALAP) scheduling approach adopted by the state-of-the-art T2 compiler [25]. In this approach, the input is the original FHE DAG without BOOT operations and bootstrapping is performed dynamically on any output ciphertext that exceeds a certain noise threshold and will be scheduled as an input to a downstream multiplication operation.

FHE-Runner also has the ability to verify that noise constraints are satisfied by a schedule. This is done by executing the schedule in both the plaintext and ciphertext domains, decrypting the ciphertext results, then comparing them to the plaintext results. If the two sets of results are within a certain approximation range, then the FHE schedule works as expected and the noise constraints were not violated. This approach is useful when testing real applications with real inputs, such as the realistic benchmarks used in this thesis. As confirmed in Section 7.3, benchmarks are run with example inputs and the results are validated. Note that for synthetic task graphs that operate on random inputs, the verification may not work as the randomness may lead to ciphertext values that are not within the valid range of CKKS ciphertexts.

In terms of memory requirements, for the parameter set used in this thesis's experimental evaluation (corresponding to 128 bits of security, a bootstrap level of 21, and a level window of 9), each ciphertext is 20.3 MB in size, while the bootstrapping key size is 391.7 MB. Therefore, for the first GB of RAM, about 31 ciphertexts can reside along with the bootstrapping key. For every subsequent GB of RAM, about 50 more ciphertexts can be stored in memory at once. Under such memory requirement, the testing device with 16 GB of RAM is able to store about 781 ciphertexts in memory simultaneously, while a cloud server typically offers much more memory. Moreover, at any one point during the execution of the FHE program, only *alive* ciphertexts and *final* ciphertexts are kept in memory. Here, *alive* ciphertexts are those that will be used as inputs to some operation in near future, and final ciphertexts are the outputs of the program, which are never used as inputs to future operations. The alive status of a ciphertext is tracked After every operation, its ciphertext input is checked, which is deleted if it is no longer alive. This means that worst case memory usage for a schedule run through the execution engine is related to the maximum number of alive ciphertexts at any point, plus the number of final ciphertexts.

Chapter 7
EXPERIMENTAL EVALUATION

We evaluate FHE-Booster on both synthetic FHE task graphs and three real FHE benchmarks performing different floating point programs. Specifically, to evaluate our heuristic on a variety of test cases, we have generated 25 synthetic FHE task graphs with the number of operations ranging from 400-640. These graphs are randomly generated using a realistic format such that each operation can have either one or two inputs, and each input can be either a fresh ciphertext or produced by a parent operation. The graph generator also takes several parameters constraining the properties of the graphs, including a target number of operations (that can be slightly exceeded), the minimum and maximum depth of a graph (set to 35 and 100 when generating the graphs) and the minimum and maximum number of operations on each level (set to 1 and 25).

All of the experiments were conducted on a standard laptop with an Intel i7-7700HQ processor that has a 2.8 GHz base clock and 16 GBs of RAM; all tests had the number of threads set to 4 and the level windows t set to 9, unless stated otherwise. Also, the bootstrap level l is 21 in the execution engine.

7.1 Estimated vs. Actual Execution Times

When FHE-Booster generates a schedule that can be passed to the execution engine, it also generates an estimate of the schedule's execution time, in an unspecified time unit, ϵ. Addition and subtraction are considered to take 1 ϵ, multiplication 5 ϵ, and bootstrapping 300 ϵ. Note that these are the default estimated latencies, but they are configurable so that users can more accurately match them to the actual latencies of these FHE operations on their hardware. As Figure 7.1 shows, there is a

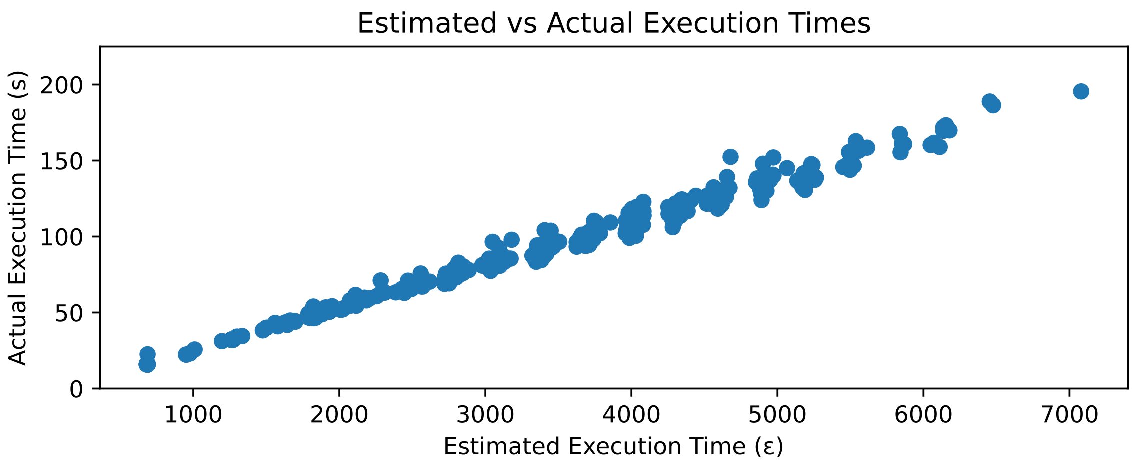


Figure 7.1: Comparison between estimated and actual execution times of synthetic FHE programs

strong corellation between these estimates and the actual running time of the program. Therefore, these estimates can be a helpful tool for comparing the performance of different schedules without needing to actually execute them. In this chapter, the shown correlation will be used in one case to avoid the lengthy process of running many FHE programs.

7.2 Results Using Synthetic FHE Task Graphs

The first set of experiments were conducted on the synthetic FHE task graphs, with the goal of determining the best selection of parameters.

The first parameter to test is the way that bootstrapped results are forwarded to dependent tasks. Since selectively forwarding bootstrapped results only removes unnecessary waiting time of a dependent task and does not add any overhead, it is expected that it will always perform better on average. To confirm this, both forwarding types (complete vs. selective) were tested with the minimum bootstrapping heuristic. Indeed, the average execution time[1] for the selective forwarding type is 68.68 s, while

[1] As some graphs take 10× time to finish than other graphs, in all the experiments the

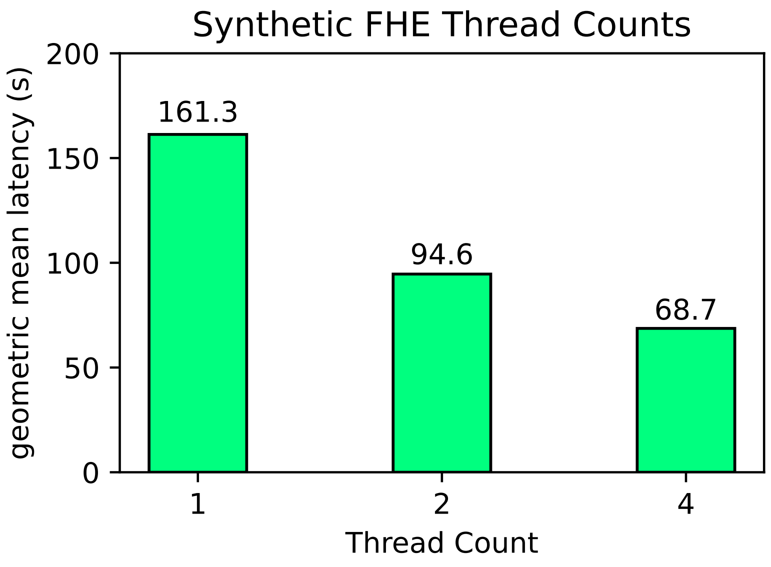


Figure 7.2: Average execution time of the synthetic FHE programs for the minimum bootstrapping heuristic with different thread counts

the average execution time for the complete forwarding type is 73.52 s. Because of this improvement, only the selective forwarding type is used for the remaining experiments (unless stated otherwise), except for the baseline ALAP approach, to which selective forwarding is not applicable.

Another aspect of FHE-Booster to explore is how well it exploits parallelism. To examine this, the 25 synthetic programs were tested with different thread counts, using the minimum bootstrapping heuristic with selective forwarding. The comparison is found in Figure 7.2. Clearly, FHE-Booster is able to take advantage of parallelism inherent in the task graphs, with a significant performance improvement when going from 1 to 2 threads. There are diminishing returns going from 2 to 4 threads, but still a significant improvement. This thesis opts not to test more than 4 threads since the testing device has only 4 physical CPU cores. Considering schedules with four threads perform best, this configuration is used for all future tests.

average execution time is computed as the geometric mean to equalize the importance of each graph. For the same reason, the average number of bootstraps is also computed as the geometric mean.

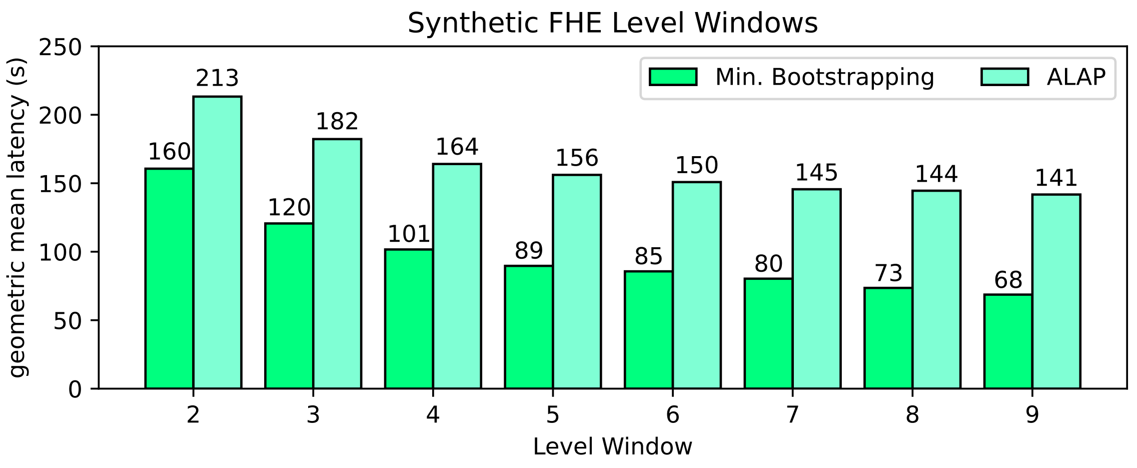


Figure 7.3: Average execution time of the synthetic FHE programs for the minimum bootstrapping and ALAP heuristics with different level windows

Next, this thesis considers how different level windows affect performance. To do so, all synthetic graphs were run with different level windows using both the minimum bootstrapping heuristic and the ALAP heuristic. Figure 7.3 shows two clear patterns from these results. First, average execution times decrease as the level window increases. This is simply because a larger level window generally means that less bootstrap operations are required to satisfy the noise constraints. However, the performance improvement from increasing the level window diminishes as it is further increased. Second, the discrepancy between the minimum bootstrapping heuristic and the minimum bootstrapping heuristic generally grows with the level window. This is because a larger level window affords more opportunities for optimizing the bootstrap set, and because a single bootstrap operation is less costly for lower level windows. Given these results, all future tests use a level window $t = 9$, unless stated otherwise.

The next set of experiments explores the weight parameters s, r, and u defined in Eq. (5.3). All seven permutations of using these parameters alone and in different combinations are tested for the score heuristic. The results of average execution time and average number of bootstrapping operations are presented in Fig. 7.4 and 7.5, respectively. Note that the figures label the score-based heuristics by listing the weights as s, r, u. For instance, the bar named "1,0,0" has s=1, r=0, and u=0, meaning that the score heuristic only considers the number of unsatisfied bootstrap segments in that configuration.

The results clearly show that the score-based heuristics consistently outperform the ALAP scheduler used by the T2 compiler [25] in all the different parameter combinations tested. The shortest average execution time is 43.5 s, which is 47.5% better than ALAP and only 3.08% slower than minimum bootstrapping. The shortest execution time is achieved when only the number of unsatisfied segments is used to generate the score. Notably, this configuration produces the smallest average number of bootstrapping operations. In fact, as expected, the results show a strong correlation between the number of bootstraps and the execution time. However, there is an exception, where bar 8 (using only the urgency attribute) and bar 9 (ALAP) perform fewer bootstraps but have longer execution time than bars 6 and 7. This can be linked to the fact that bars 6 and 7 include the slack attribute in their scores, allowing bootstraps to take better advantage of parallelism. Also, urgency and ALAP perform the same number of bootstrap operations in every scenario, yet urgency preforms much better because it knows the bootstrap set *a priori*, and hence can use selective forwarding to reduce the waiting time of some dependent tasks, as well as better balance the workload across different worker threads. Yet these optimizations are not possible to ALAP due to its dynamic nature.

The results suggest that the most useful attribute is the number of unsatisfied bootstrap segments containing an operation, since the configurations with this attribute (bars 2-5 in Fig. 7.4) outperform those without (bars 6-8). Using this heuristic by itself (bar 2 in Fig. 7.4) performed best overall. This is expected since this heuristic strives to mimic minimum bootstrapping, and using it alone produces the least amount of bootstraps on average, as shown in Fig. 7.5. The other two heuristics, namely, operation slack and bootstrapping urgency, perform better in conjunction with the segment heuristic than on their own.

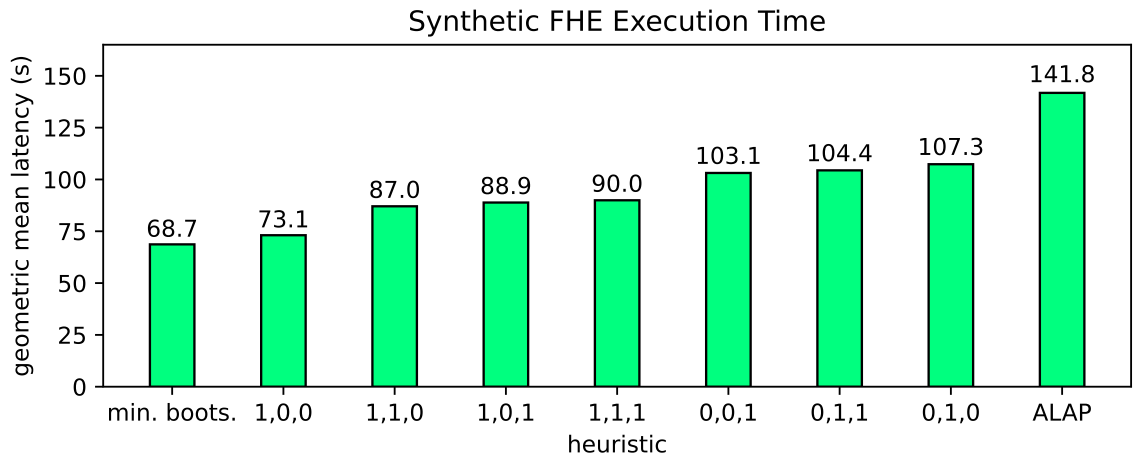


Figure 7.4: Average execution time of the synthetic FHE programs with different heuristics

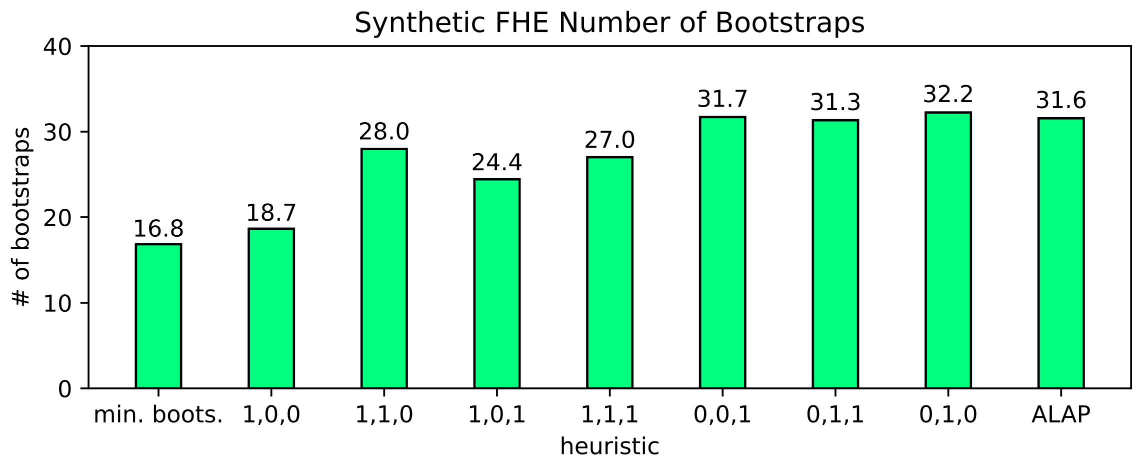


Figure 7.5: Average number of bootstraps of the synthetic FHE programs with different heuristics

7.3 Case Studies

This section considers three realistic FHE programs and examines the performance of FHE-Booster in relation to each. The average execution times shown in this section come from executing the corresponding program three times with that specific setup.

7.3.1 PID

This first case study examines the PID program of the floating point benchmark suite FPBench [15], sourced from [13] and [14]. The program is slightly modified for use in this thesis, and the C++ implementation is found in Appendix D. This program is passed through the FHE-Booster framework and data is collected in the same way as in Section 7.2. Same as before, Figures 7.6 and 7.7 show that the heuristics significantly outperform the standard ALAP approach. The power of selective forwarding conversion is also clear, as the minimum bootstrapping with complete forwarding offers a 1.43× speedup over ALAP, while minimum bootstrapping with selective forwarding offers a 1.83× speedup. In fact, the four heuristics using the segments attribute and selective forwarding outperform minimum bootstrapping with complete forwarding, despite higher total bootstrap operation counts. Furthermore, heuristics including the segments attribute continue to outperform those that do not, but in this case, the same bootstrap set is chosen whenever $s = 1$, regardless of the values of r or u. Therefore, any variation in runtimes between those heuristics is due to random factors as opposed to any significant reason. Importantly, since example inputs are provided, FHE-Runner's verification capabilities can be used to confirm that the program executes correctly. All executed versions (the C++ implementation, the FHE schedule run in plaintext, and the FHE schedule run in ciphertext) return $m = 0.535712$, with the largest reported error of the decrypted value being approximately 0.001%.

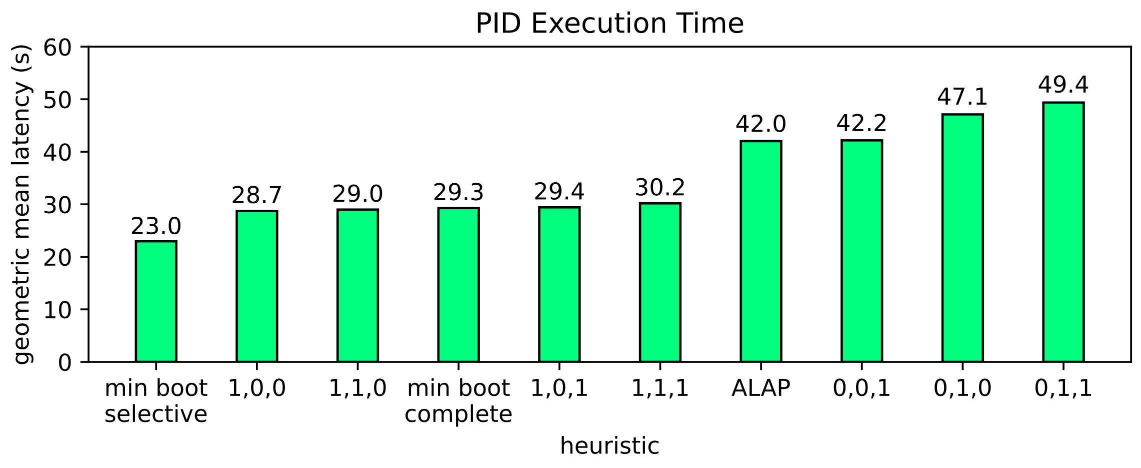


Figure 7.6: Average execution time of the PID benchmark with different heuristics

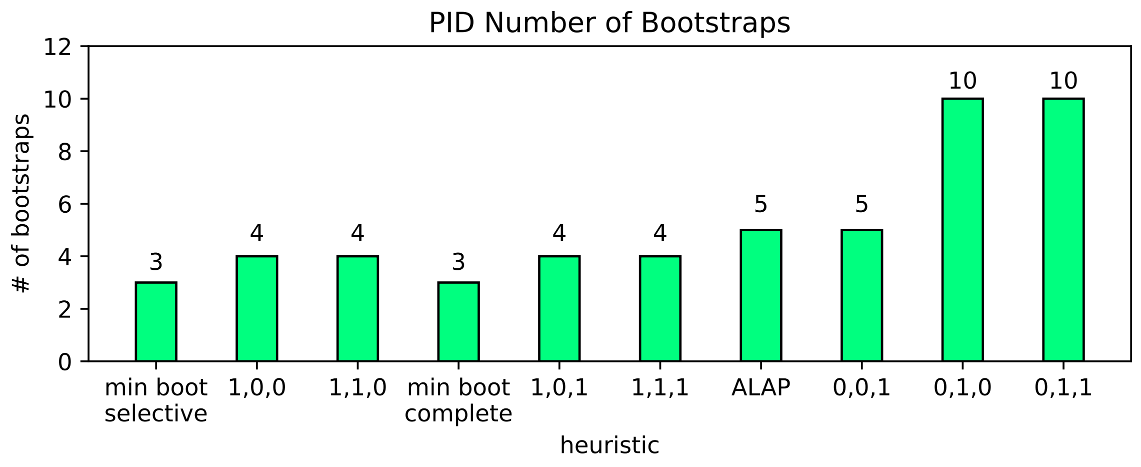


Figure 7.7: Average number of bootstraps of the PID benchmark with different heuristics

7.3.2 Pendulum

This case study program also comes from the FPBench suite [15], and the C++ implementation is found in Appendix E. Like the PID program, the outputs of the FHE program are verified for correctness. The execution engine was set to report any errors greater than 0.5%, and none were reported. The outputs of the execution engine were also confirmed to match those of the direct C++ implementation. The results for this program, shown in Figures 7.8 and 7.9 offer a departure from the previous results. In particular, the segments-based heuristics do not categorically perform best, and, most importantly, ALAP performs very well, essentially matching minimum bootstrapping.

To fully understand the reason for this, this thesis examines the pendulum program with other level windows. Figures 7.10 and 7.11 show the execution times and number of bootstraps with the level window $t = 8$. The key observation about the results when $t = 8$ is that the five best performing heuristics execute the same number of bootstrap operations as when $t = 9$, and have a faster average execution time. This may seem contradictory to Figure 7.3, which showed that average execution times increase as the level window decreases. However, this is only generally true, and does not hold in all cases. The pendulum program happens to be structured in such a way that the number of bootstrap operations needed does not change when going from $t = 9$ to $t = 8$. Since bootstrap operations execute faster for lower values of t, performing the same number of bootstraps means the program runs faster overall when $t = 8$.

The key takeaways are significantly different when $t = 10$, however, with the results shown in Figures 7.12 and 7.13. In this case, the results begin to look much more like the overall results for the synthetic graphs. Mimimum bootstrapping performs best by a non-negligible margin, and ALAP is the worst heuristic, along with two of the score-based heuristics that perform the same number of bootstraps. This essentially reveals how ALAP performed so well when $t = 9$ and $t = 8$: The pendulum task graph is structured such that using ALAP chooses a minimal bootstrap set *by chance* with these specific bootstrap level and level window combinations. Therefore, minimum

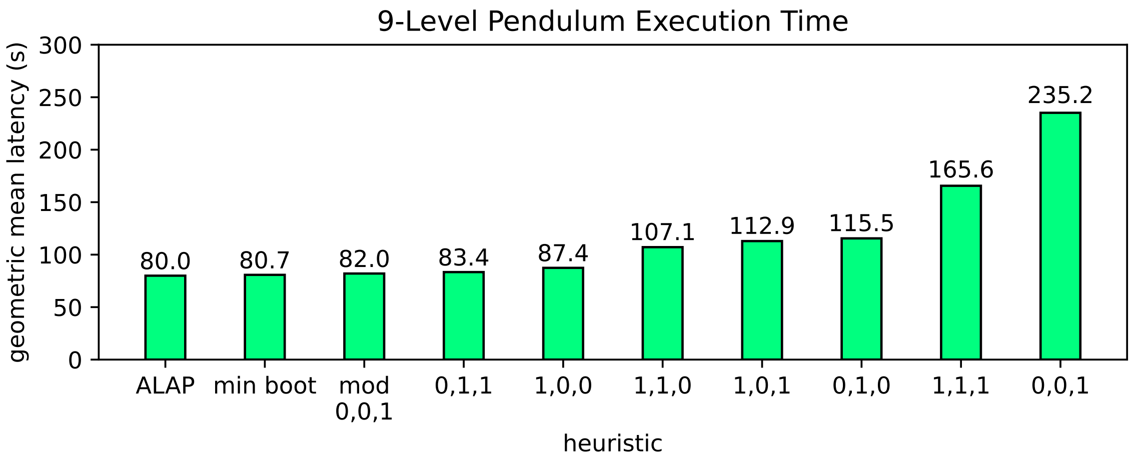


Figure 7.8: Average execution time of the pendulum benchmark with different heuristics at a window level $t = 9$

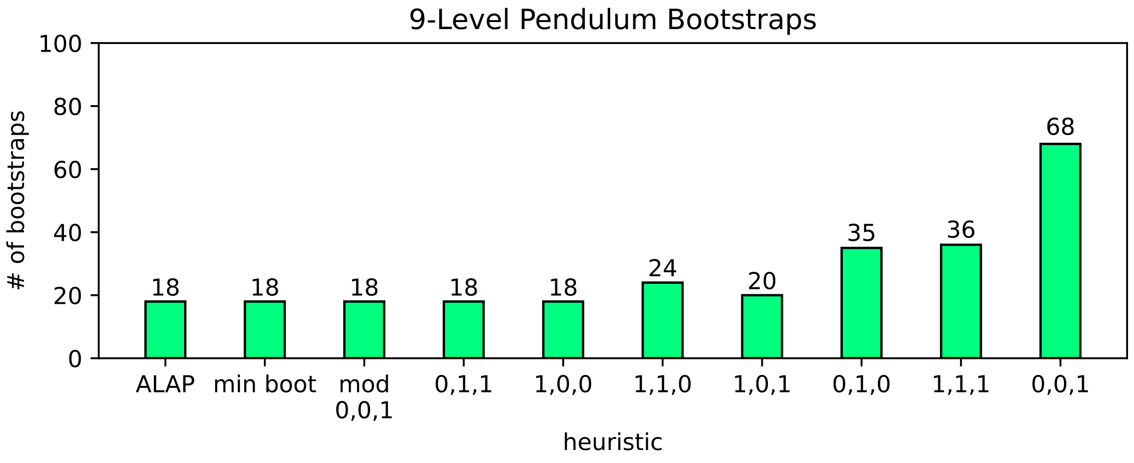


Figure 7.9: Average number of bootstraps of the pendulum benchmark with different heuristics at a window level $t = 9$

bootstrapping is not *gauranteed* to perform better than ALAP. Rather, for some graphs, ALAP may choose the same ciphertexts to bootstrap depending on certain parameters.

One more interesting point about setting the level window to $t = 10$ is that its best case performs about the same in terms of execution time as the best case when $t = 8$. Again, despite the general point made clear by Figure 7.3, it does not always hold true for every program that increasing the level window will improve performance, even if it decreases the number of bootstraps. This further shows that choosing an ideal level window is not as simple as "bigger is better."

A final important feature of the pendulum results is that ALAP performs extremely differently from the pure urgency heuristic ($s = 0, r = 0, u = 1$) when $t = 9$ and $t = 10$. ALAP and pure urgency performed the same number of bootstrap operations for 22 of the 25 synthetic graphs with only small differences in the other 3 cases, so the massive discrepancy between the two heuristics here is a major difference for this program. However, a simple modification to the method for selecting a bootstrap set reveals the reason for this difference. When handling maximum score ties by choosing the operation with the highest ID (corresponding to the latest topological order) instead of lowest ID, the urgency heuristic obtains a bootstrap set identical to ALAP, bringing the performance of the two in line. This modified version of the heuristic is labeled in the figures as "mod 0,0,1."

Because of this, it may be worthwhile to consider that choosing the highest ID operation is a better tie-breaking method. To test this, schedules for the 25 synthetic graphs were generated using this method and their estimated execution times are compared to tie-breaking via lowest ID in Figure 7.14. The results do not show any clear pattern, so which tie-breaker is preferable likely changes on a case-by-case basis.

7.3.3 Matrix convolution

The final case study does not come from FPBench, but rather is a simple implementation of matrix convolution. The C++ code is found in Appendix F. The results of this program were verified in the same way as the pendulum program. Due

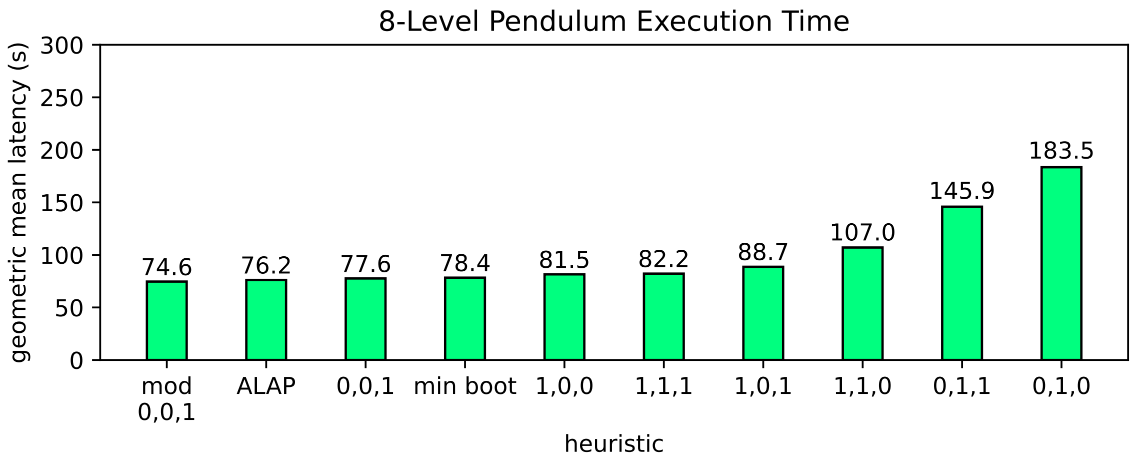


Figure 7.10: Average execution time of the pendulum benchmark with different heuristics at a window level $t = 8$

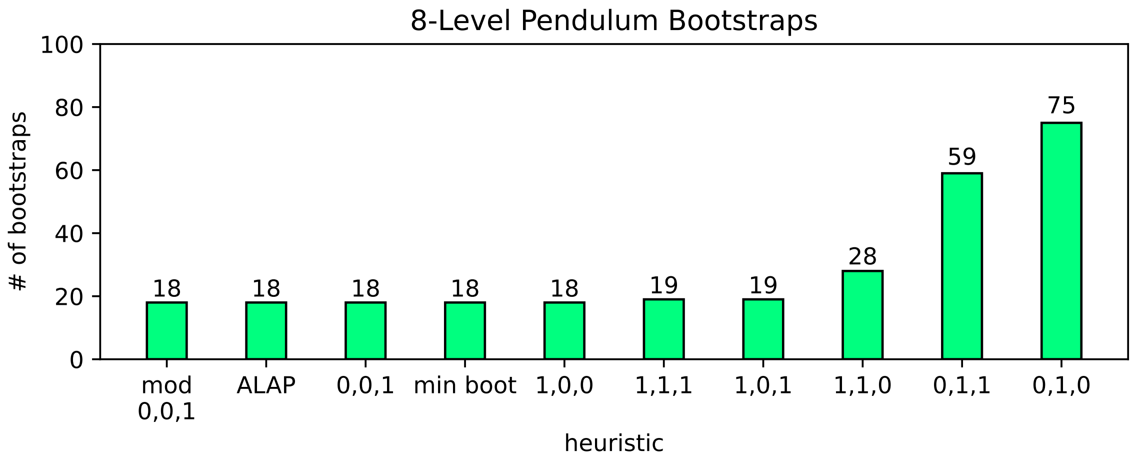


Figure 7.11: Average number of bootstraps of the pendulum benchmark with different heuristics at a window level $t = 8$

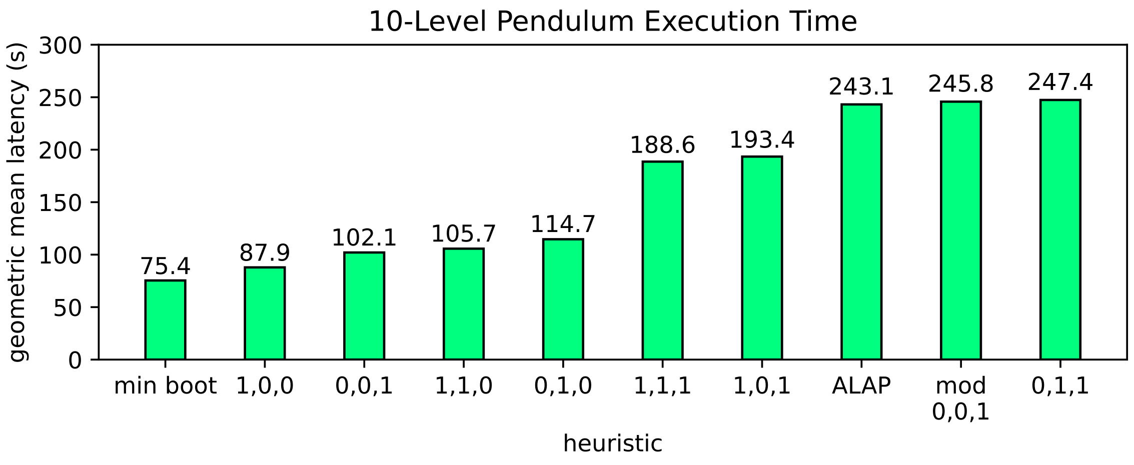


Figure 7.12: Average execution time of the pendulum benchmark with different heuristics at a window level $t = 10$

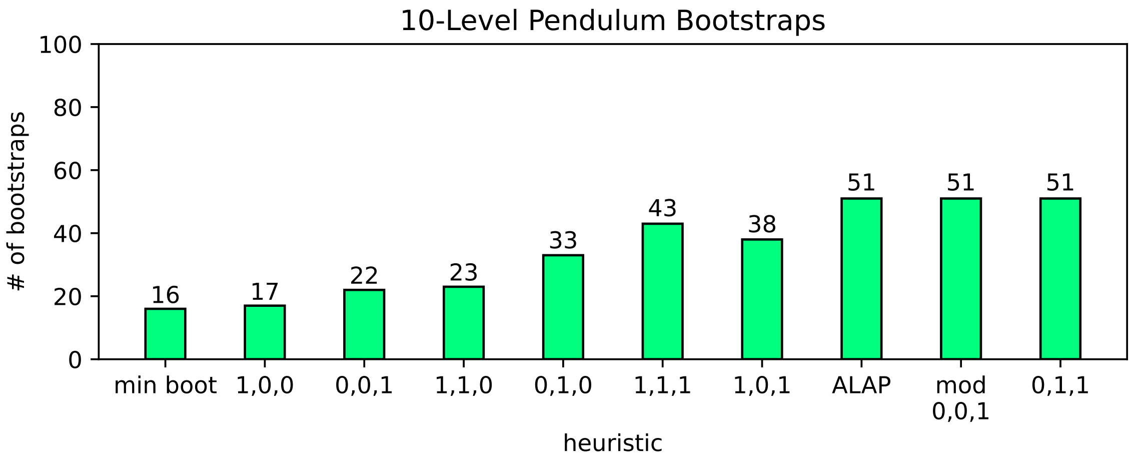


Figure 7.13: Average number of bootstraps of the pendulum benchmark with different heuristics at a window level $t = 10$

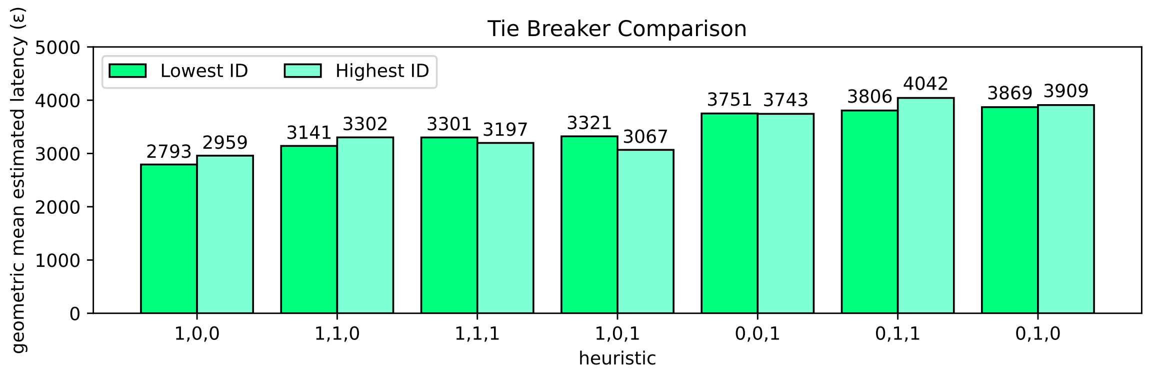


Figure 7.14: Comparing tie-breaking methods via average estimated execution time of the synthetic FHE programs with different heuristics

to memory constraints, the testing device was unable to produce bootstrap segments for level window $t = 9$, so instead this program was tested for $t = 3$, $t = 4$ and $t = 5$. Because the ILP solver used for this thesis restricts the amount of memory it can use, minimum bootstrapping results are also not reported for this benchmark. The results are shown in Figures 7.15 and 7.16. It is evident from the results that this program differs from the previous two. First, no matter the level, all the shown heuristics choose approximately the same bootstrap set. The only significant variance in execution times is ALAP, since it does not benefit from selective forwarding. The reason for this homogeneity in bootstrap sets is that the structure of the matrix convolution task graph is fairly simple and choosing the ideal bootstrap set is easy to do, even manually. Furthermore, that ideal bootstrap set will match with the ALAP set, no matter what the level window is set to. For this reason, even though minimum bootstrapping results could not be obtained, there is no reason to believe that heuristic would perform any better than the tested heuristics. Based on these results, matrix convolution can act as an example for a class of programs that have similar characteristics. For example, the simple multiply-accumulate program shown in Algorithm 7 would also have a minimum bootstrap set that is equivalent to ALAP for any level window.

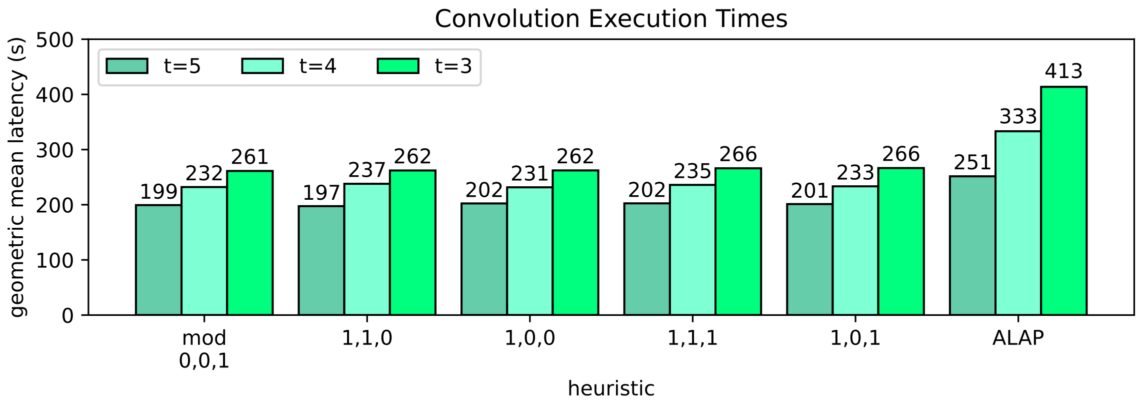


Figure 7.15: Average execution time of the convolution benchmark with different heuristics

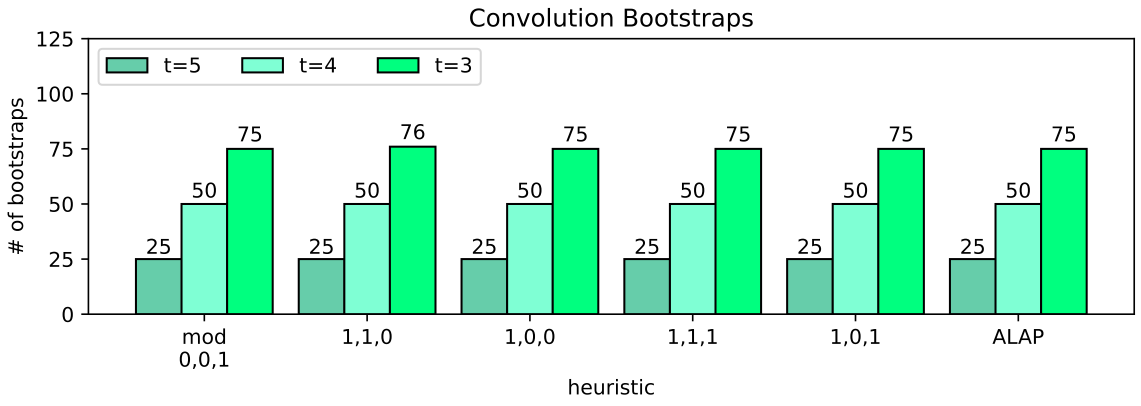


Figure 7.16: Average number of bootstraps of the convolution benchmark with different heuristics

Algorithm 7: Multiply Accumulate Example

```
input  : $a$ - The initial accumulation value
         $b$ - An array of numbers
         $N$ - Desired number of iterations
output: $accum$ - The accumulated value

$accum \longleftarrow a$
for $i \in 0, ..., N$ do
    $accum \longleftarrow accum + (accum \times b[i])$

return $accum$
```

Table 7.1: Performance results of FHE-Booster compilation steps, in seconds

	generate segments	**min bootstrap ILP**	**select bootstrap set for 1,1,1**	**convert complete 0,1,0 to selective**	**list schedule 0,1,0 w/ complete forwarding**
Geomean	3.924	336.565	0.072	0.563	0.271
Min	0.021	0.030	0.004	0.003	0.111
Median	5.054	3601.160	0.079	0.938	0.241
Max	572.319	3765.030	2.313	30.306	16.383

7.4 Execution Time of Scheduling Steps

Although is has now been shown that FHE-Booster achieves its goal of improving the performance of FHE programs, it is also important to consider the performance of FHE-Booster itself. Therefore, this section discusses the performance of the various steps of the FHE-Booster framework. The results are summarized in Table 7.1, with the data accounting for all 25 synthetic graphs and the 3 case studies, assuming bootstrap level $l = 21$ and level window $t = 9$, except for the convolution program, which has $t = 5$. Note that the heuristics included for analysis in the table were chosen because they are expected to be worst case scenarios in terms of FHE-Booster performance.

The first step is to generate bootstrap segments. While this step usually runs quickly, testing with larger graphs has shown that this step in the FHE-Booster framework can be a bottleneck, in large part due to memory utilization. A potential solution to this issue is discussed in Section 8.2.

The next step is to choose a bootstrap set. One major benefit of using a score-based heuristic is that a set of bootstrapping operations can always be chosen in polynomial time with respect to the sum of the sizes of all the bootstrap segments. In contrast, the minimum bootstrapping problem is NP-hard, so the related ILP model can take a very long time to complete. The minimum bootstrapping ILP models were executed using the LINGO 19.0 software provided by LINDO [36]. The execution time was limited to 1 hour, and if the limit was reached, the minimal solution identified up to that point was returned. This means that slightly better solutions may have been identified by allowing the solver to run longer. However, considering minimum bootstrapping generally performed best anyway, this is not a significant concern in the tests. With a bootstrap level of $l = 21$ and a level window of $t = 9$, 13 of the 25 synthetic FHE task graphs reach this limit, as well as 1 of the real benchmarks, although minimum bootstrap sets could not be calculated for the convolution program, due to memory limitations set by the ILP solver. To compare the latency of ILP solving to our polynomial-time heuristic, data for choosing a bootstrap set with the heuristic $s = 1, r = 1, u = 1$ are also presented. While the geometric mean of the execution times of the ILP model was 336.565 s, the same metric for choosing a bootstrap set with the heuristic $s = 1, r = 1, u = 1$ was 72 ms, an improvement of four orders of magnitude.

Once a bootstrap set is chosen, the only remaining steps are selective bootstrap forwarding conversion (which is optional), and creating a schedule, which is done using a custom list scheduler. As Table 7.1, these steps run quickly.

7.5 Optimal Model

Finally, this section discusses the optimal model presented in Section 4.4. Graph 17 of the synthetic set was chosen for testing because it had a comparatively low number of bootstrap segments while still offering some complexity that the IP model could tap in to. Unfortunately, after running the solver for over 14 hours, no solutions had been found. The same occurred with the PID benchmark (run for over 12 hours) and Graph 12 (run for over 16 hours), which has the least bootstrap segments of the synthetic graphs. Therefore, it seems that programs deep enough to require bootstrapping with the execution engine parameters are unlikely to achieve a feasible solution in a reasonable amount of time, so this thesis opts not do any comparisons between the optimal model and the heuristics.

Still, to confirm the model even works, the solver was run on the task graph shown in Figure 3.1, assuming a bootstrap level of $l = 0$ and level window of $t = 2$ to match the example. Thread count was set to four. For this very simple model, the solver found the optimal solution in 0.56 s. The output was successfully converted to an execution engine schedule that satisfies the noise constraints.

Chapter 8
CONCLUSION AND FUTURE WORK

8.1 Conclusion

Fully homomorphic encryption (FHE) allows a user to outsource computation to a cloud server while preserving the privacy of their personal data. This thesis has presented FHE-Booster, a complete scheduling framework for optimizing the performance of arbitrary FHE programs on multi-core platforms. In particular, FHE-Booster has included three novel approaches for inserting and scheduling bootstrapping operations. A key benefit of FHE-Booster is that it is applicable to all modern FHE libraries. To demonstrate its effectiveness, an execution engine utilizing a state-of-the-art CKKS implementation has been developed and used to run experiments with three real applications. Overall, the proposed FHE-Booster scheduling optimizations enable significant speedups compared to the scheduler of the state-of-the-art T2 compiler.

8.2 Future Work

The final part of this thesis summarizes limitations of FHE-Booster and how they can be addressed in future research.

Section 4.2.2 noted that the process of generating bootstrap segments can be a bottleneck in terms of runtime and memory utilization, especially for large FHE programs with many operations and complex dependencies. Because this issue stems from considering the entire FHE program (and therefore all the bootstrap segments) at once, a potential method to overcome this limitation is to consider select portions of a task graph at a time, with bootstrap segments only generated for that portion. In this way, bootstrap points can be chosen to satisfy those segments, and therefore the noise of all ciphertexts exiting that portion of the graph will be known. This process can be repeated until all operations in the graph have been considered and all noise constraints are satisfied. The trade-off, of course, is that not considering all bootstrap segments at once may lead to more and/or inferior bootstrap points being chosen than by the current method.

FHE-Booster's static nature also presents some limitations. Particularly, it requires knowledge of the noise levels of the input ciphertexts, which may be challenging for programs where the noise level of the inputs could be any value between 0 and ρ, the noise threshold. Such a situation is realistic, for example, when operating on an encrypted database. To overcome this limitation, the framework could determine bootstrap points in a just-in-time fashion while the FHE program is running. This dynamic method would take in the current noise levels of alive ciphertexts, determine bootstrap segments for a certain number of upcoming levels in the task graph, and then choose ciphertexts to bootstrap just for that section of the program.

Another feature that would make using FHE-Booster more practical is compilation from a higher-level language to a representation that can be used by the framework. A simple, limited C-like format should be sufficient, similar to the format used by the T2 compiler [25]. To further improve FHE-Booster's practicality, some "server-ready" features can be implemented. For example, in its current state, plaintext values would need to be present on the server at some point, defeating the purpose of FHE. To address this, FHE-Runner should be updated to optionally accept pre-encrypted data as inputs, as well as the FHE parameters necessary to operate on those ciphertexts.

Another potential direction for future work is exploring new heuristics for choosing bootstrap points. One benefit of the score-based scheduling heuristic is that it is fairly general and could be easily altered or expanded to include other heuristics. Similarly, any heuristics for integer programming models can be applied to both the minimum bootstrapping model and the optimal model. For example, genetic algorithms have been shown as an powerful tool for nonlinear models [19].

Finally, application-level optimizations beyond the placement of bootstrapping operations could be added to the FHE-Booster framework. Optimizations could include both traditional compiler optimizations, as well as the ones uniquely developed for the different assumptions specific to FHE. The `my_sine(x)` function in Appendix E offers a good example. This thesis opts to calculate the necessary powers of x for the sine approximation using a very specific set of 7 multiplications, even though they can be calculated with only 6 multiplications. Using these 7 multiplications decreases the multiplicative depth from six to four, potentially reducing the minimum number of bootstraps for the program. Another example of a potential optimization is replacing certain multiplication operations with additions. Specifically, if a plaintext in a ciphertext-plaintext multiplication is an integer i, then the multiplication can instead be calculated by summing the ciphertext with itself $i - 1$ times. Altering the computation in this way should offer benefits so long as it meaningfully reduces the number of bootstrap operations needed to run a program. Some previous works have considered FHE-specific optimizations that can be automatically applied to programs, including the T2 compiler [25] and ALCHEMY's PT2CT compiler [11], but there is still a large optimization space to explore in this domain.

BIBLIOGRAPHY

[1] Ahmad Al Badawi et al. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.

[2] Ahmad Al Badawi, Chao Jin, Jie Lin, et al. Towards the AlexNet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1330–1343, 2020.

[3] Ahmad Al Badawi and Yuriy Polyakov. Demystifying bootstrapping in fully homomorphic encryption. Technical report, Duality Technologies, 2022.

[4] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

[5] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO*, pages 483–512. Springer, 2018.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[7] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2019.

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, pages 409–437. Springer, 2017.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[10] Eric Crockett and Chris Peikert. $\Lambda\circ\lambda$: Functional lattice cryptography. In *ACM Conference on Computer and Communications Security (CCS)*, pages 993–1005, 2016.

[11] Eric Crockett, Chris Peikert, and Chad Sharp. ALCHEMY: A language and compiler for homomorphic encryption made easy. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1020–1037, 2018.

[12] Wei Dai and Berk Sunar. cuFHE (v1.0). [Online]. Available: `https://github.com/vernamlab/cuFHE`, 2018.

[13] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems*, pages 31–46, Cham, 2015. Springer International Publishing.

[14] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Transformation of a pid controller for numerical accuracy. *Electronic Notes in Theoretical Computer Science*, 317:47–54, 2015. The Seventh and Eighth International Workshops on Numerical Software Verification (NSV).

[15] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, et al. Toward a standard benchmark format and suite for floating-point analysis. July 2016.

[16] Morris J Dworkin et al. Advanced encryption standard (AES). 2001.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. 2012.

[18] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. REDsec: Running Encrypted Discretized Neural Networks in Seconds. In *Network and Distributed System Security Symposium (NDSS)*, pages 1–17, 2023.

[19] Kerry Gallagher and Malcolm Sambridge. Genetic algorithms: A powerful tool for large-scale nonlinear optimization problems. *Computers & Geosciences*, 20(7):1229–1236, 1994.

[20] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, et al. BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2205.14017*, 2022.

[21] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[22] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[23] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, pages 75–92. Springer, 2013.

[24] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, et al. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893*, 2021.

[25] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. SoK: New insights into fully homomorphic encryption libraries via standardized benchmarks. 2022.

[26] Charles Gouert and Nektarios Georgios Tsoutsos. Romeo: conversion and evaluation of hdl designs in the encrypted domain. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[27] Charles Gouert and Nektarios Georgios Tsoutsos. (RED)cuFHE: Evolution of FHE acceleration for Multi-GPUs. [Online]. Available: https://github.com/TrustworthyComputing/REDcuFHE, 2022.

[28] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.

[29] Xiao Hu, Minghao Li, Jing Tian, and Zhongfeng Wang. Efficient homomorphic convolution designs on FPGA for secure inference. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(11):1691–1704, 2022.

[30] Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies*, 2021(3):246–264, 2021.

[31] Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. *arXiv preprint arXiv:2202.08814*, 2022.

[32] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, et al. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 114–148, 2021.

[33] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In *ASIACRYPT*, pages 608–639. Springer, 2021.

[34] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, dec 1999.

[35] Tys Lefering. vcgviewer. https://www.softpedia.com/get/Multimedia/Graphic/Graphic-Others/vcgviewer.shtml.

[36] LINDO Systems, Inc. LINGO. https://www.lindo.com/, 2022.

[37] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, pages 1–23. Springer, 2010.

[38] Souhail Meftah, Benjamin Hong Meng Tan, Chan Fook Mun, et al. DOReN: Toward efficient deep convolutional neural networks with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 16:3740–3752, 2021.

[39] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, et al. CoFHEE: A co-processor for fully homomorphic encryption execution. *arXiv preprint arXiv:2204.08742*, 2022.

[40] NuCypher. nuFHE (v0.0.3). https://github.com/nucypher/nufhe, 2019.

[41] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238. Springer, 1999.

[42] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, et al. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39. IEEE, 2021.

[43] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, 2009.

[44] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[45] Nikola Samardzic et al. F1: A fast and programmable accelerator for fully homomorphic encryption. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*, pages 238–252, 2021.

[46] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, et al. CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA*, pages 173–187, 2022.

[47] Georg Sander. Visualization of compiler graphs. https://www.rw.cdl.uni-saarland.de/people/sander/private/html/gsvcg1.html, 1995.

[48] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108. IEEE, 2021.

[49] Tommy White, Charles Gouert, Chengmo Yang, and Nektarios Georgios Tsoutsos. FHE-Booster: Accelerating fully homomorphic execution with fine-tuned bootstrapping scheduling. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023. © 2023 IEEE.

Appendix A

PERMISSIONS

This thesis acts as an expansion of an article previously published for the 2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) [49]. Information about IEEE's conference copyright policies can be found at the archived page here: https://web.archive.org/web/20230402142028/https://conferences.ieeeauthorcenter.ieee.org/get-published/about-transferring-copyright-to-ieee/. The following paragraph is taken from that page, per IEEE's request.

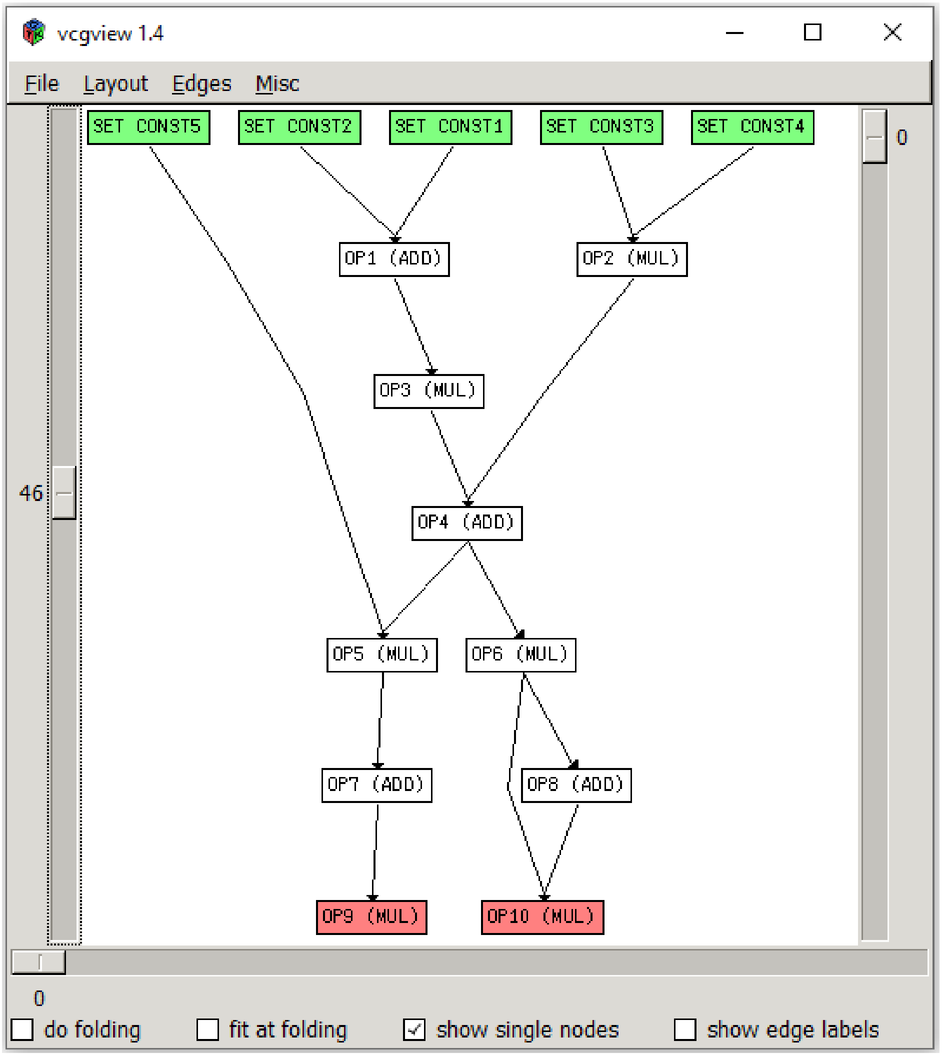
Appendix B
DAG FILE FORMAT

In order for FHE-Booster to operate on an FHE program, the program must first be encoded as a DAG. A simple format was developed for this thesis that can be easily parsed by code. As an example, the task graph from Figure 3.1 is encoded in our custom file format below. First, the five inputs are listed. Next a "∼" character acts as a separator. Finally, the operations are listed, with each line using the format

<ID>,<operation_type>,<input1>,<input2>

where <input2> is optional. Specifically, inputs that begin with "k" refer to the ciphertexts listed in the first section of the file, while inputs that begin with "c" refer to the outputs of the operation with the corresponding ID.

```
1,SET
2,SET
3,SET
4,SET
5,SET
~
1,ADD,k1,k2
2,MUL,k3,k4
3,MUL,c1
4,ADD,c2,c3
5,MUL,k5,c4
6,MUL,c4
7,ADD,c5
8,ADD,c6
9,MUL,c7
10,MUL,c6,c8
```



☐ do folding ☐ fit at folding ☑ show single nodes ☐ show edge labels

Figure B.1: DAG rendered by vcgview.

Furthermore, the VCG file format [47] is useful to visualize the task graphs, with a simple C++ program converting the custom DAG format to VCG. Then, vcgviewer [35] was used in the workflow of this thesis to render the VCG file, though any program that renders VCG files may be used. Figure B.1 shows a screenshot of vcgviewer rendering the task graph from Figure 3.1.

Appendix C
SCHEDULE FILE FORMAT

Given a DAG file and a bootstrap set, the list scheduler produces an assembly-like schedule that can be used by FHE-Runner. Alternatively, if no bootstrap set is provided, a schedule without bootstrapping will be created for use with the ALAP bootstrapping method. Each line of the schedule describes an operation using the following format:

<operation> c<output> c/k/p<input1> c/k/p<input2> t<thread_num>

The second input is optional depending on the operation: MUL, ADD, and SUB require two inputs, while BOOT and INV require one input. Ciphertexts IDs should begin with "c" (indicating the output of an operation) or "k" (indicating a ciphertext that is an input to the program). Plaintext values are also supported and their IDs should begin with "p". At least one of the inputs to any of the operations must be a ciphertext. Bootstrapped ciphertexts are generally distinguished by inserting a zero after the "c" in their ID. Below is the schedule generated by the optimal INLP model for the task graph of Figure 3.1 with a bootstrap level of $l = 0$, level window of $t = 2$, and a thread count of four.

```
ADD c7 c5 c5 t1
ADD c1 k1 k2 t2
ADD c4 c2 c3 t2
BOOT c04 c4 t2
MUL c6 c04 c04 t2
MUL c10 c6 c8 t2
MUL c3 c1 c1 t3
MUL c5 k5 c04 t3
MUL c9 c7 c7 t3
MUL c2 k3 k4 t4
ADD c8 c6 c6 t4
```

A file format is also defined in order to assign inputs when running a schedule in FHE-Runner. Below is an example for the above schedule, which has five inputs.

```
k1,1.0
k2,-3.0
k3,7.54
k4,0.82
k5,-0.5
```

Appendix D
PID BENCHMARK CODE

```
#include <iostream>

double ex0(double m, double kp, double ki, double kd, double c)
{
        const int num_iterations = 20;

        const double dt = 0.2;
        const double invdt = 1.0 / dt;
        double e = 0.0;
        double p = 0.0;
        double i = 0.0;
        double d = 0.0;
        double r = 0.0;
        double eold = 0.0;

        for (int iter = 0; iter < num_iterations; iter++)
        {
                e = c - m;
                p = kp * e;
                i = i + ((ki * dt) * e);
                d = (kd * invdt) * (e - eold);
                r = (p + i) + d;
                m = m + (0.01 * r);
                eold = e;
        }
        std::cout << "m: " << m << std::endl;
        return m;
}

int main()
{
        const double m = -5.0;
        const double kp = 9.4514;
        const double ki = 0.69006;
        const double kd = 2.8454;
        const double c = 1.0;
        ex0(m, kp, ki, kd, c);
}
```

Appendix E
PENDULUM BENCHMARK CODE

```
#include <iostream>

double my_sine(const double x)
{
        const double r1 = 1.666666666666666667e-1;
        const double r2 = 8.333333333333333333e-3;
        const double r3 = 1.984126984126984127e-4;
        const double r4 = 2.755731922398589065e-6;
        const double r5 = 2.505210838544171878e-8;

        const double x2 = x * x;
        const double x3 = x2 * x;
        const double x5 = x3 * x2;
        const double x6 = x3 * x3;
        const double x7 = x5 * x2;
        const double x9 = x6 * x3;
        const double x11 = x6 * x5;

        return x - (x3 * r1) + (x5 * r2) - (x7 * r3) + (x9 * r4) - (x11 * r5);
}

double ex0(const double t0, const double w0, const double N)
{
        const double h = 0.01;
        const double half_h = 0.005;
        const double gL = -4.903325;
        double t = t0;
        double w = w0;
        for (int n = 1; n <= N; n++)
        {
                double k1w = gL * my_sine(t);
                double k2t = w + (half_h * k1w);
                double k2w = gL * my_sine(t + (half_h * w));
                t = t + (h * k2t);
                w = w + (h * k2w);
        }
        std::cout << "w: " << w << std::endl;
        std::cout << "t: " << t << std::endl;
        return t;
}

int main()
{
        ex0(1.0, 2.0, 12);
}
```

Appendix F

MATRIX CONVOLUTION BENCHMARK CODE

```
#include <array>
#include <iostream>

int main()
{
    const int image_height = 5;
    const int image_width = 5;
    const int kernel_size = 3;
    const int padding_each_side = kernel_size - 2;
    const int padded_height = image_height + (padding_each_side * 2);
    const int padded_width = image_width + (padding_each_side * 2);
    const int num_kernels = 32;

    std::array<std::array<char, image_width>, image_height>
        input_image =
            {{{1, 1, 1, 1, 1},
              {1, 1, 1, 1, 1},
              {1, 1, 1, 1, 1},
              {1, 1, 1, 1, 1},
              {1, 1, 1, 1, 1}}};

    std::array<
        std::array<std::array<char, kernel_size>, kernel_size>,
        num_kernels>
        kernels;

    kernels.fill({{{0, 0, 0},
                   {0, 1, 0},
                   {0, 0, 0}}});

    std::array<std::array<char, padded_width>, padded_height>
        old_image;
    std::array<std::array<char, image_width>, image_height>
        new_image;

    for (int i = 0; i < padded_height; i++)
    {
        for (int j = 0; j < padding_each_side; j++)
        {
            old_image[i][j] = 0;
            old_image[i][padded_width - j - 1] = 0;
        }
    }

    for (int i = 0; i < padded_width; i++)
    {
        for (int j = 0; j < padding_each_side; j++)
        {
            old_image[j][i] = 0;
            old_image[padded_height - j - 1][i] = 0;
        }
    }
```

```
for (int i = 0; i < image_height; i++)
{
    for (int j = 0; j < image_width; j++)
    {
        const int y = i + padding_each_side;
        const int x = j + padding_each_side;
        old_image[y][x] = input_image[i][j];
    }
}

for (int i = 0; i < num_kernels; i++)
{
    for (int j = 0; j < image_height; j++)
    {
        for (int k = 0; k < image_width; k++)
        {
            new_image[j][k] = 0;
            for (int q = 0; q < kernel_size; q++)
            {
                const int old_y = j + q;
                for (int z = 0; z < kernel_size; z++)
                {
                    const int old_x = k + z;
                    new_image[j][k] +=
                        old_image[old_y][old_x] *
                        kernels[i][q][z];
                }
            }
            }
        }

        for (int j = 0; j < image_height; j++)
        {
            for (int k = 0; k < image_width; k++)
            {
                const int y = j + padding_each_side;
                const int x = k + padding_each_side;
                old_image[y][x] = new_image[j][k];
            }
        }
    }

    for (int i = 0; i < image_height; i++)
    {
        for (int j = 0; j < image_width; j++)
        {
            std::cout << (int)new_image[i][j] << ", ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```