PADERBORN UNIVERSITY

DISSERTATION

# Efficient Parallel Branch-and-Bound Search on FPGAs Using Work Stealing and Instance-Specific Designs

*by*
Heinrich Riebler

*A thesis submitted in fulfillment of the requirements for the degree of Dr. rer. nat.*

*in the*

Faculty for Computer Science, Electrical Engineering and Mathematics

September 20, 2019

*Gewidmet meinen Eltern, in Dankbarkeit und Liebe.*

# *Acknowledgements*

I would like to thank:

- My supervisor Prof. Dr. Christian Plessl. This work would not have been possible without your professional and personal support during the last years.
- Prof. Dr. Marco Platzner for serving as a reviewer for this thesis.
- Prof. Dr. Friedhelm Meyer auf der Heide, Prof. Dr. Kevin Tierney and Dr. Theodor Lettmann for serving on the oral examination committee.
- All colleagues from the High-Performance IT Systems research group, the Paderborn Center for Parallel Computing and the SFB 901 – On-The-Fly Computing. A warm thanks goes to Gavin Vaz, Tobias Kenter and Achim Lösch for intensive collaborations on joint projects.
- A special thanks goes to Michael Laß, Robert Mittendorf and Thomas Löcke, who directly contributed to my research covered in this thesis as Master students or student assistants I have supervised. It was a pleasure to work with you.

These acknowledgments would not be complete without thanking my family for their unconditional support: first and foremost, Lina, Mael and our little son on the way. Furthermore, I would like to thank my parents, my sister, my brother and all members of my family-in-law.

# Abstract

In recent years, increasing technization and market analysis purposes have resulted in more and more data being generated. The growing need of data analysis and processing has become omnipresent for many combinatorial optimization or planning problems. To take advantage of the promises of the digital age, efficient search algorithms and their efficient implementations in terms of performance and energy efficiency are important. Only the combination and fine tuning of efficient algorithms *and* their efficient implementation on suitable platforms can lead to a high performance and low energy consumption.

One of the most common methods for processing such very large search spaces is using branch-and-bound (B&B) search algorithms. B&B search algorithms are highly relevant because they are used to solve many real-world operational problems (e.g. production and personnel planning, scheduling, complex decision processes, etc.). The search space in branch-and-bound searches is organized in a tree data structure and the algorithm tries to eliminate infeasible solutions as early as possible by pruning unpromising subtrees through a bounding function. Since these excluded subtrees no longer have to be considered, the computing effort is reduced considerably in some cases.

In this thesis, we study the insufficiently understood efficient realization of branch-and-bound algorithms for field programmable gate arrays (FPGAs). FPGAs are integrated circuits consisting of programmable logic blocks and programmable interconnects that can be specialized for specific applications after manufacturing the chip. Branch-and-bound problems are inherently difficult and not the typical class of problems that have been tackled using FPGAs, because they are control-driven and not data-driven. On the other hand, FPGAs have proven to be highly efficient in terms of chip area, power consumption and performance for a wide range of other suitable application domains. In this thesis, we bridge this gap and show that custom hardware designs can significantly accelerate the execution of these algorithms. First, we identify general elements of B&B algorithms and develop and demonstrate their efficient implementation as a finite state machine on FPGAs. Our architecture shows trade-offs between highly optimized combinational datapaths for the performance-critical parts of the search tree and more resource-efficient pipelined ones for the less frequent and more complex parts.

Then we extend our design with two optimization techniques to further improve the efficiency. For the first optimization we introduce the concept of hardware workers that autonomously cooperate using work stealing to allow parallel execution of branch-and-bound algorithms and full utilization of the target FPGA. The hardware workers dynamically share and balance their work and show near linear speedups. For the second optimization we explore the advantages of instance-specific designs for B&B algorithms that target a specific problem instance to improve performance and combine them with the design using work stealing. The instance-specific design

utilizes the high potential of FPGAs for specialization and custom optimization for a particular problem instance. We present a fully automated generation of custom-tailored designs that existing tools do not deliver. We demonstrate how instance-specific designs can be generated on-the-fly such that the provided speedups outweigh the additional time required for design synthesis.

Finally, we evaluate all of our approaches and compare each result to those obtained using similar techniques in software. Our results show that our hardware implementation targeting a Maxeler FPGA system can outperform a software implementation while being more energy efficient at the same time.

# Zusammenfassung

Durch die zunehmende Technisierung entstehen immer mehr Daten oder es werden Daten für marktanalytische Zwecke generiert. Entsprechend aufwendig werden Entscheidungs-, Planungs- oder Optimierungsprobleme, deren Ziel darin besteht, eine bestimmte – im Idealfall beste – Lösung in den immensen Suchräumen dieser Daten zu finden. Neben effektiven Algorithmen zur Lösung solcher Probleme spielt deren hocheffiziente Implementierung auf modernen Rechenanlagen eine immer wichtigere Rolle. Von besonderer Bedeutung ist dabei das Verhältnis der Datenverarbeitungsleistung zur elektrischen Leistungsaufnahme. Nur durch die Kombination und Feinabstimmung von effizienten Algorithmen *und* effizienten Implementierungen auf geeigneten Rechenanlagen kann eine hohe Datenverarbeitungsleistung bei geringer Leistungsaufnahme erzielt werden.

Eine der verbreitetsten Methoden, um derartige Suchprobleme effizient zu lösen, ist das Branch-and-Bound (B&B) Verfahren. Branch-and-Bound wird beispielsweise vielfach im Bereich der Unternehmensplanung (z.B. zur Produktions- und Personaleinsatzplanung) und zur Entscheidungsunterstützung bei kombinatorischen Optimierungsproblemen verwendet. Der sich aus der Problemstellung ergebende Suchraum wird beim B&B-Verfahren in einer Baum-Datenstruktur organisiert. Das Verfahren schließt durch schrittweises Verzweigen im Baum (branch) und Begrenzen von Teilbäumen (bound) systematisch Teilbereiche des Suchraums aus, die nicht zu einer gültigen bzw. optimalen Lösung führen können. Da diese ausgeschlossenen Teilbereiche nicht mehr betrachtet werden müssen, reduziert sich der Rechenaufwand teilweise erheblich.

In dieser Arbeit konzentrieren wir uns auf die effiziente Implementierung von Branch-and-Bound Verfahren auf Field Programmable Gate Arrays (FPGAs) für derartig gelagerte Suchprobleme. FPGAs sind integrierte Schaltkreise, bestehend aus programmierbaren Logikbausteinen, die für bestimmte Aufgaben spezialisiert werden können. Branch-and-Bound-Probleme gehören nicht zu den klassischen Problemen, die auf FPGAs untersucht werden, da die Berechnungsvorschrift von B&B-Verfahren kontroll- und nicht datengesteuert ist. Andererseits haben sich FPGAs für eine Vielzahl geeigneter Anwendungsbereiche als hocheffizient in Bezug auf Schaltkreisfläche und Verhältnis von Datenverarbeitungsleistung zur elektrischen Leistungsaufnahme erwiesen. In dieser Arbeit schließen wir diese Lücke und zeigen, dass hochspezialisierte FPGA-Implementierungen die Ausführung von Branch-and-Bound-Algorithmen erheblich beschleunigen können. Dazu identifizieren wir zunächst allgemeine Elemente von Branch-and-Bound-Algorithmen und zeigen systematisch Möglichkeiten einer effizienten Implementierung auf FPGAs mit Hilfe von Zustandsautomaten. Wir untersuchen bei der Auswahl unserer Architektur die entstehenden Kompromisse zwischen hochoptimierten kombinatorischen Datenpfaden für die leistungskritischen Teile des Suchraums und ressourceneffizienteren Datenpfaden mittels Pipelining für die weniger häufigeren und komplexeren Teile des Suchraums. Anhand einer konkreten Fallstudie demonstrieren wir, wie

durch Ausnutzung der architektonischen Merkmale, der Spezialisierung und der unterschiedlichen Parallelitätsstufen von FPGAs eine Auslagerung der Berechnung auf den FPGA zu einer Verbesserung der Datenverarbeitungsleistung führt.

Anschließend erweitern wir unser Design noch um zwei Optimierungsverfahren, um die Effizienz der Ausführung weiter zu steigern. Die erste Optimierung erzielen wir durch die Parallelisierung des Branch-and-Bound-Verfahrens auf dem FPGA. Als Parallelisierungsstrategie verwenden wir Work Stealing, womit die autonome Zusammenarbeit mehrerer Instanzen auf einem FPGA ohne zentrale Steuereinheit ermöglicht wird. Jede Instanz bemüht sich selbstständig um Arbeitspakete, indem sie aktiv Pakete von anderen Instanzen *stiehlt*. Dadurch werden eine effiziente Arbeitslastverteilung und eine parallele Ausführung des B&B-Verfahrens sichergestellt. Unsere Implementierung ermöglicht die volle Ausnutzung des FPGAs und zeigt nahezu lineare Skalierungseigenschaften, wenn die Taktrate des FPGAs konstant bleibt. Für die zweite Optimierung untersuchen wir instanzspezifische FPGA-Designs, angewandt auf Branch-and-Bound-Algorithmen. Diese zielen darauf ab, besonders schwierige bzw. zeitintensive Probleminstanzen einer Anwendung zu verbessern. Dazu wird die konkrete Probleminstanz analysiert und die konfigurierbare Schaltung speziell auf das konkrete Problem optimiert. Dies demonstriert das hohe Spezialisierungspotential von FPGAs. Wir beschreiben eine vollautomatische Generierung von maßgeschneiderten FPGA-Designs für das Branch-and-Bound-Verfahren und kombinieren diese zusätzlich mit den Parallelisierungstechniken aus der ersten Optimierung.

Schließlich evaluieren wir alle unsere Ansätze und vergleichen jedes Ergebnis mit denen, die mit äquivalenten Techniken bei einer Ausführung in Software auf Central Processing Units (CPUs) erzielt werden können. Unsere Ergebnisse zeigen, dass unsere Hardware-Implementierung auf einem Maxeler FPGA-System eine Implementierung in Software hinsichtlich der Datenverarbeitungsleistung übertreffen kann und gleichzeitig energieeffizienter ist. Zudem können wir belegen, wie instanzspezifische Designs sogar nach Bedarf *on-the-fly* generiert werden können, sodass die erzielte Beschleunigung die zusätzliche Zeit für die Erstellung des FPGA-Designs mittels Hardware-Synthese überwiegt.

# Table of Contents

# Chapter 1

# Introduction

The branch-and-bound (B&B) algorithmic pattern is a powerful tool for processing very large search spaces or to find optimal solutions in them. It is the most commonly used algorithmic pattern and systematic method to solve combinatorial optimization problems such as scheduling, logistics, applied mathematics, planning, decision processes and many others. Branch-and-bound algorithms explore tree-based search spaces systematically by eliminating unpromising subtrees as early as possible. Nevertheless, the exploration is highly time-intensive for large problem instances because the search trees are growing exponentially and are extremely irregular in size and structure. There are several techniques to tackle those instances. The most promising ones are parallelization and the utilization of problem-specific features.

## 1.1 Contributions Overview

In this thesis, we systematically analyze and study the insufficiently understood efficient realization of branch-and-bound algorithms for field programmable gate arrays (FPGAs). FPGAs have proven to be highly efficient in terms of chip area, power consumption and performance with a high potential for specialization for different workloads. However, the irregular structure of branch-and-bound algorithms and the control-driven execution flow makes them not the typical class of problems that have been addressed with FPGAs. The main contributions of this thesis are:

1. We identify general elements that are required to implement B&B problems with FPGAs and abstract them as a finite state machine design. We present an architecture that uses highly optimized combinational datapaths for the performance-critical levels of the search tree and more resource-efficient pipelined ones for the less frequent and more complex levels. On the basis of a concrete case study, we show how a transformation from software to hardware alone can lead to improvements in orders of magnitude in performance by exploiting the architectural features and different levels of parallelism of FPGAs.

2. We then extend this design in order to allow multiple hardware workers to dynamically share and balance their load using work stealing when exploring the large search space of a branch-and-bound problem. We present a parallel FPGA architecture that is scalable with the available resources and provides speedups proportional to the number of workers.

3. Using the parallelized design, we further accelerate the execution by exploring the advantages of instance-specific computing on FPGAs. We present a fully automated tool flow to generate designs that are custom tailored to a specific problem instance that existing tools do not deliver. Our work shows that this technique is in particular beneficial to accelerate problem instances that are especially difficult and highly time-intensive to solve.

4. We evaluate all of our proposed methods and compare each result to those obtained using similar techniques in software on CPUs. In contrast to existing approaches, we also demonstrate how instance-specific designs can be generated on-the-fly such that the provided speedups outweigh the additional time required for a complete design synthesis.

All presented concepts and results have been peer-reviewed and published in two premier international conferences [3, 2] and one journal article [1]. Beside Heinrich Riebler as lead researcher, first author and main contributor, Robert Mittendorf contributed during his Master's thesis [207] to the first implementation of work stealing in soft- and hardware. Thomas Löcke contributed during his Master's thesis [186] to the first implementation of instance-specific computing in soft- and hardware and, finally, Michael Laß contributed as a student assistant to the evaluation results used in our journal article [1]. A complete list of the author's publications is summarized before the main bibliography starting on page 117.

## 1.2 Thesis Structure

The remainder of this thesis is structured as follows: Chapter 2 starts with the foundations on reconfigurable computing and the design flow and programming model for FPGAs that is used in the practical parts of this work. Chapter 3 takes an excursion to the cryptography and information security domain to motivate the case study used throughout the thesis to apply the concepts to real-world and relevant problems. In Chapter 4, we present our intermediate findings with regard to our case study. The tackled subproblem in this chapter is highly suitable for FPGAs and our presented solution significantly outperforms software-based implementations. The next three chapters present the original research on efficient branch-and-bound algorithms for reconfigurable hardware. In Chapter 5, we present the main concepts and building blocks for a general hardware design for processing large search trees using branch-and-bound with FPGAs. Afterwards, we describe in Chapter 6 the extension of our general, but sequential branch-and-bound hardware design to allow parallelization of the work on FPGAs using hardware workers. The resulting parallel design is then further improved by using instance-specific computing in Chapter 7. We describe how different search trees using B&B can be dynamically constructed by utilizing application- and instance-specific information to improve the search process including an on-the-fly hardware acceleration. Finally, we present related work in Chapter 8 and conclude our work and point to directions for future research in Chapter 9.

# Chapter 2

# Foundations: Reconfigurable Computing

This chapter provides the background information and foundations on reconfigurable computing that are used in the following chapters. The ideas and concepts in this thesis heavily rely on practical parts, implemented and evaluated on reconfigurable hardware, namely on field programmable gate arrays (FPGAs).

In this chapter, we first give a general motivation for reconfigurable computing in Section 2.1. We describe the ideas and building blocks of FPGAs and then outline the hardware acceleration with FPGAs. The hardware acceleration is first explained in Section 2.2 on a general level and then in Section 2.3 concretely on the components and program blocks of the MaxCompiler, which is the programming model used for the design and implementation of our concepts.

## 2.1 Field-Programmable Accelerators

A computer system is typically considered from two perspectives, the software and the hardware side. In a traditional view, the hardware side offers a fixed functionality after fabrication and the software provides the flexibility by executing different applications to change the type of computation. Reconfigurable computing [129, 215] tries to blend both perspectives together by making the hardware (datapath, memory and/or functional units) programmable after fabrication. The configuration (usually of the size of several kilo- or megabytes) is loaded onto the reconfigurable hardware via a data stream of bits and can be partially or completely reassigned [89, 227]. Computer systems using reconfigurable hardware are reprogrammed to adapt their *architecture* to the requirements of the type of computation that is under execution. The ability to make substantial architectural changes can lead to orders of magnitude better implementations in terms of performance (execution time), utilization (chip area efficiency) and power usage [220, 163].

A field programmable gate array (FPGA) [274, 172] is the most prominent universally programmable reconfigurable hardware. The FPGA can be reconfigured in the field by the user after fabrication. The programming (also called configuration) determines the functionality of individual flexible logic blocks and their flexible interconnect (see Figure 2.1). The logic blocks are laid out as an array structure and can be grouped into hierarchical clusters, which are interconnected with configurable switch boxes. The three most important elements of a logic block are lookup tables (LUTs) to implement combinational logic, flip-flops (FFs) to implement sequential

logic to buffer values, and configurable multiplexers to select between these elements. The configuration to program the FPGA is generated by a logic synthesis. The process is similar to the compilation of software but needs hours to days to complete, because of the tremendous search space to place and route the desired functionality to the logic blocks. In fact, branch-and-bound search algorithms are often used for this process [260, 258, 104, 146].

The performance of an FPGA design depends on a number of assessable factors, such as the number of cycles required to perform a task and the read/write rate of the memory and communication buses. The functionality is fixed after configuration and the runtime of an application is given by the number of cycles and the achieved clock frequency. There are no dynamic events such as interrupts. This makes the performance of an FPGA design well predictable and allows the modeling of the expected acceleration before its actual implementation. It can help to verify the results of the design and implementation and to identify possible sources of bottlenecks.

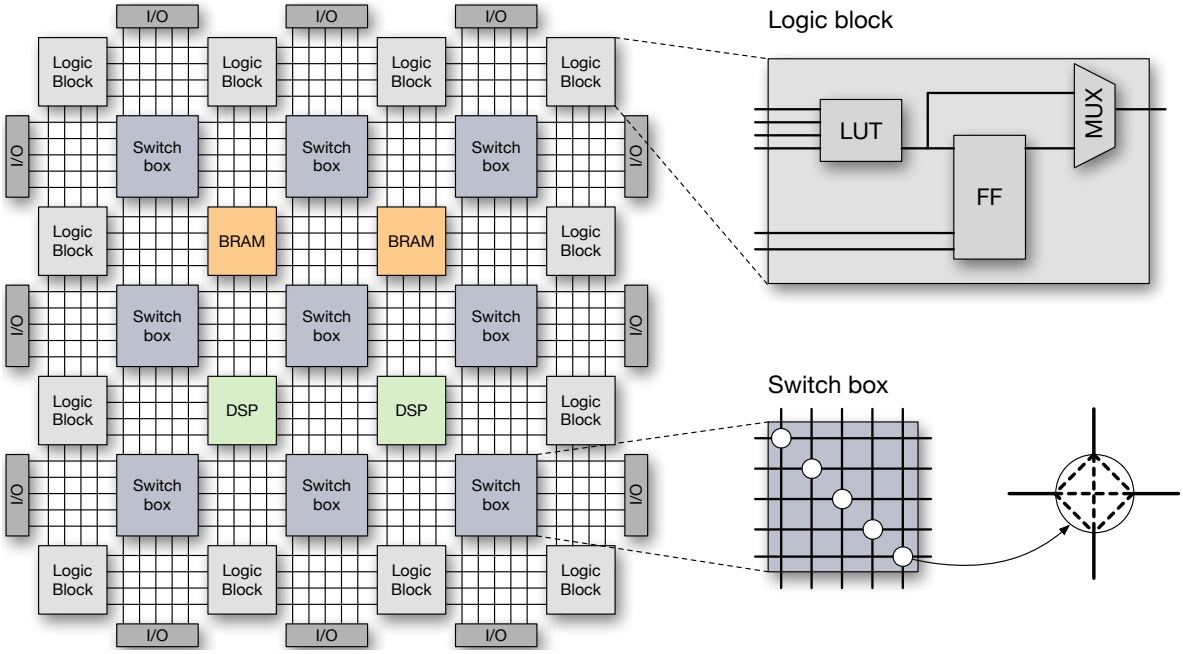



Figure 2.1: Schematic illustration of an FPGA architecture. The array structure consists of logic blocks, switch boxes and specialized elements (DSPs or BRAMs).

Over time, modern FPGAs have expanded their building blocks beyond the basic logic capabilities mentioned above to include more specialized elements fixed in silicon. Conceptually, these elements can also be built from the basic logic primitives, but with higher chip area requirements and lower clock frequency. Examples of these building blocks include digital signal processor (DSP) blocks typically used for multiplications, multi-gigabit transceivers for high-speed communications, hard IP processor cores to mimic system-on-chip behavior and, most importantly, external and local (high bandwidth) memory. External memory is typically on-board DDR RAM. Local memory is embedded into the FPGA's array structure as BRAMs. A single BRAM can store several kBits and multiple BRAMs can be cascaded to form larger memory blocks with configurable address depths and data widths. The logic of lookup tables can also be used as local memory to store several bits. This type of local memory is called LUT RAM.

FPGAs are especially suitable for computationally intensive applications with a regular computation and/or communication pattern. Individual independent operations can be processed completely in parallel on spatially different computing units. The resulting execution is data- and not control-driven. Control-driven systems perform the required calculations at different points in time on the same computing units (temporal processing). The computing unit has to be able to perform a variety of functions and is therefore typically more general than necessary for a concrete task. In contrast, data-driven FPGA designs have their own custom computing unit for the respective calculation (spatial processing). Each custom unit can be exactly specialized for the required task. FPGAs have been used for many data-driven application domains, e.g. from image processing and recognition [21, 271, 108, 71], pattern matching in data or network streams [64, 26, 182, 181], or in the information security domain [285, 267, 231]. FPGAs also have very suitable properties in the cryptography domain [138, 79]. Cryptographic applications usually require a lot of computing power and can be strongly spatially parallelized due to their often independent (sub-)structures. Compared to CPUs, FPGA-accelerated implementations can achieve a much higher throughput and energy savings when encrypting and decrypting data and can still be flexibly configured to different parameters [282, 240, 261, 119].

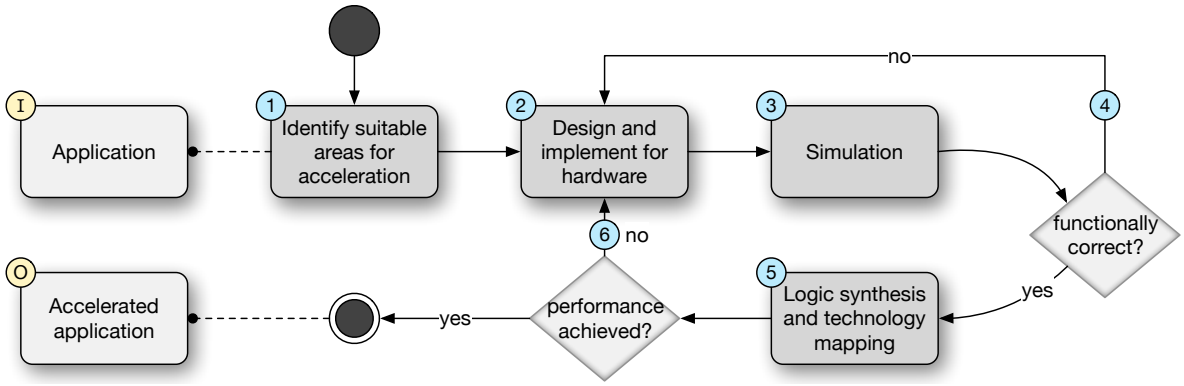



Figure 2.2: Schematic design flow of hardware acceleration.

## 2.2 Design Flow of Hardware Acceleration

In Figure 2.2, we show a schematic sequence of how an application can be accelerated using FPGAs. At the beginning, an original application (I) usually exists in software that should be accelerated in parts in hardware. In the software application, areas that are suitable for acceleration must be identified. This step (1) is usually performed by profiling the application with representative data sets. Loops are typically good candidates for recurring tasks that can be offloaded. Once the affected areas have been identified, the developer transforms the computation into a form suitable for hardware in step (2). The developer can use components from the used programming model and programming language to implement the computationally intensive part in hardware. FPGAs work best when they can continuously perform the same operations in a parallel and pipelined fashion. The major part of the development only takes place on simulation level (3), because the actual hardware synthesis is very time consuming. The process needs to be iterated in (4) until the desired *functionality* is achieved. The simulation can only give hints on the correctness of the

implementation. The actual performance and utilization of the resources in hardware can only be checked after the logic is synthesized in the step (5). If the design and implementation meet the desired functionality *and* performance, the accelerated application (O) is the outcome. Otherwise, the design needs to be adjusted and the steps (1)–(6) have to be repeated.

## 2.3 MaxCompiler Programming Model

The design flow and implementation of the practical parts of this thesis are based on the MaxCompiler [222] programming model and tool chain. It is mainly driven by Java and offers predefined APIs for specifying the FPGA implementation as so-called data flow engines (DFEs). Each DFE is comprised of one or more kernels, which implement the application logic, and of a manager that controls the routing of data streams between kernels, the CPU and off-chip memory. Figure 2.3 outlines the different components and general architecture. The system can connect multiple FPGAs and memory devices via PCIe to the CPU where the host application is located. The computation and communication tasks are divided into kernels and managers. This separation of the application eases the design and implementation process.

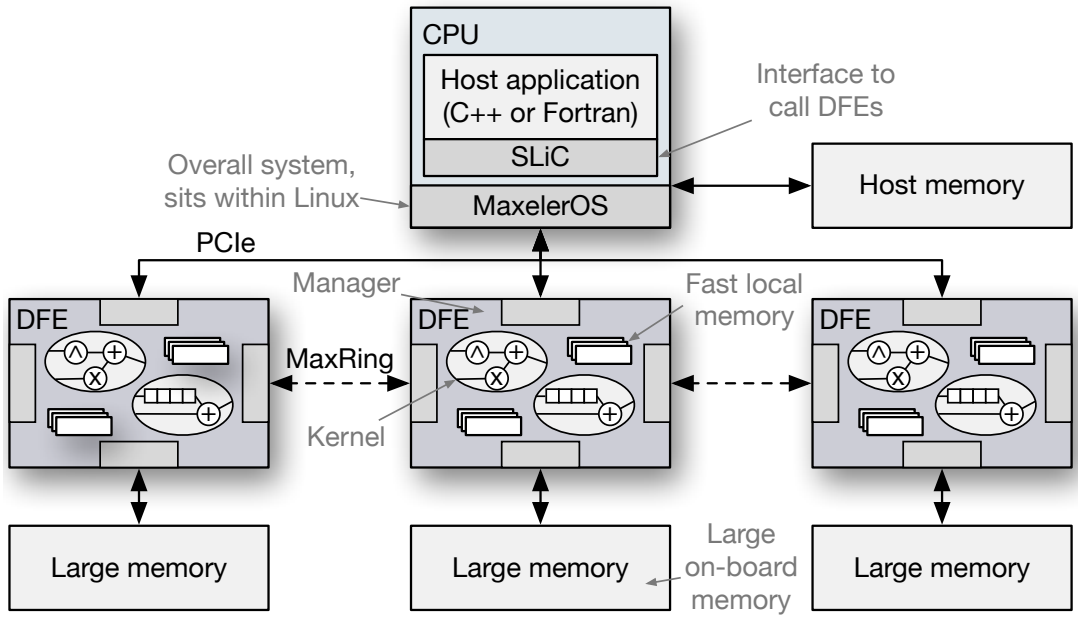



Figure 2.3: Components and overall architecture of the MaxCompiler. The depicted system consists of three FPGA cards connected through PCIe, each with on-board memory.

The development of an accelerated application with the MaxCompiler typically consists of three program parts: a host application, the kernel and the manager. In this thesis, state machines (SMs) will also play an important role and are also explained. They can either be implemented in the kernel or the manager.

### 2.3.1 Host Application

The host application can be written in C/C++ or Fortran. It calls functions to load the FPGA configuration and initializes the direct memory access (DMA) transactions. These are required for the actual data transfer to and from the FPGA. In addition, the host application allocates and initializes the data buffers and transfers them via the so-called simple live CPU (SLiC) interface. The raw data is loaded from the host to the FPGA via PCIe. As soon as enough data is available, the calculations start immediately on the FPGA. At the same time, available results are transferred back to the host application in parallel. The SLiC interface also provides functions to execute specific tasks on the DFE and set system parameters for the whole FPGA.

### 2.3.2 Kernel

A kernel (written in MaxJ) contains the parts of the actual application logic and gets its input data from the manager. It can be seen as a unidirectional cycle-free graph, where the data streams from one side to the other. The kernel execution typically has a fixed number of cycles to complete, dependent on the size of the input stream. The kernel graph is usually divided into a control part implemented with counters and a data part. The graph consists of several node types, which are listed in Table 2.1.

| | |
|---|---|
| ◯ | Calculation nodes perform arithmetic or logical operations (e.g. $+$, $-$, $*$, $<$, & or $\oplus$) or represent type conversions from one data type to another (e.g. floating-point number, fixed-point number or integer). |
| ▭ | Value nodes provide parameters. They are either constant or scalar. The host application can set scalar value nodes at runtime. |
| ◇ | Stream position nodes allow access to different positions in the data stream. They can either go forward or backward. |
| ⏢ | Multiplexer nodes enable the integration of decisions. |
| ⬡ | Counter nodes help the control flow of the application. They allow the reaction to certain positions in the data stream or can indicate boundary conditions such as the start or end of the stream. |
| ⍞ | I/O nodes connect the kernel externally to the manager. They serve as an interface for data input and output. |

Table 2.1: Possible node types of a kernel graph.

A compiler converts a program to generate such a data flow graph. The structure of the data flow graph represents the logic of the application. The graph shows the nodes for the machine commands or executable actions and the edges for the dependencies of the data. The connections are fixed and no additional control instructions are required. As soon as the preconditions for an action are met, it is executed and the result is forwarded to the next action in the graph. Individual independent actions are processed completely in parallel on spatially different computing units. The resulting execution of the statements is therefore data-driven rather than control-driven. In the second step, optimizations and transformations take place in

the graph. The actual code for the FPGA is then generated from the graph in a hardware description language.

A simple example using these nodes is shown in Figure 2.4. It receives an input stream consisting of six elements and computes the moving average with a sample window of three elements. To do this, the current element is simply added to the previous and subsequent elements and divided by three. The left part of the figure shows the start phase with empty actions in the data flow (highlighted in yellow), because the first element has no predecessor. The middle part of the figure shows the first tick when the pipeline is completely filled and correct results are produced in every tick. The final phase on the right side has again empty action nodes, because the end of the stream is reached. The example shows the consistent and regular flow of data through the graph and the respective values. The example highlights that the common fork/join constructs from multi-core processors are no longer necessary because parallelism implicitly exists.

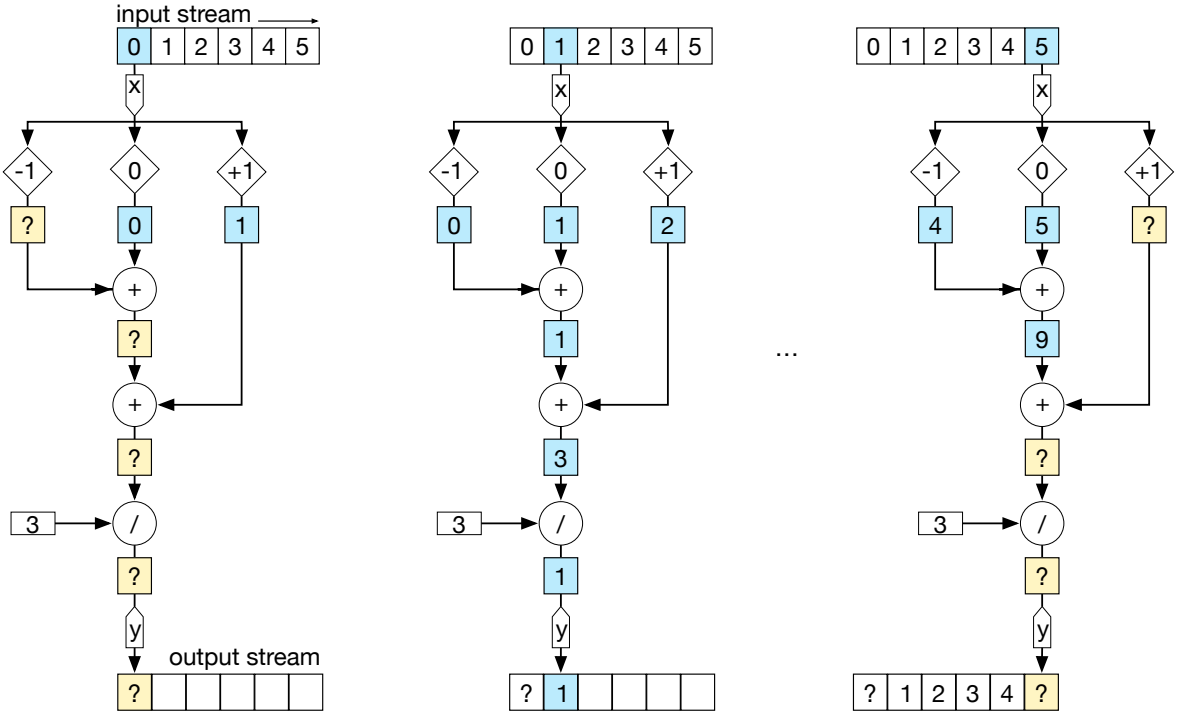



Figure 2.4: Example of three phases in the data flow of an application.

### 2.3.3 Manager

The manager [188] typically instantiates itself and the kernels. Then both components are aligned to each other. For example, it organizes the concrete type and size of the input and output of data through a kernel. The data streams can also be configured to be routed to other FPGAs or to external memory. Figure 2.3 shows the connection between different FPGAs using the dedicated high-speed MaxRing interconnect. Each FPGA can have up to two direct, bidirectional MaxRing connections. A header file is generated from the configuration of the manager via the SLiC interface. This header file defines the signature of the data streams (e.g. the data types) and contains various actions for the host application, such as whether the call to the kernel should be synchronous or asynchronous.

Other important aspects of the manager are the settings of the hardware level parameters. These are mainly the parameters to guide the hardware synthesis process (target frequency, seed, level of effort to place and route, etc.). The settings not only help to improve performance, but also can determine the success of the hardware synthesis.

### 2.3.4 State Machines

Most importantly for this thesis, the MaxCompiler offers also an API [189] to describe finite state machines (FSMs) that can control memory streams and datapaths for applications that can not be expressed as a simple streaming datapath such as regular kernels. FSMs and state machines (SMs) allow the implementation of fine-tuned control blocks in the kernel and manager. They provide deeper control mechanisms for the data flow with better accessibility than a kernel with counters alone. Figure 2.5 illustrates the execution model and the connection between the inputs and outputs when using state machines. Two functions describe the behavior in each cycle: `NEXT_STATE` and `OUTPUT_STATE`. Starting from the current state in cycle $t$ and the current input, the first function calculates the next state for cycle $(t+1)$. The second function controls the output of the machine depending on the current state and the input. Both functions are executed completely in parallel. The state machine stores the data of the current cycle for the next cycle.

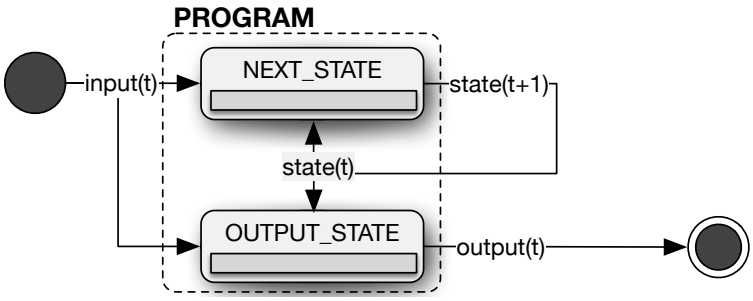



Figure 2.5: Execution model for state machine transitions.

The MaxCompiler offers state machines in two different variants: embedded in the kernel as a kernel SM or directly integrated in the manager as a manager SM. Both variants follow the execution model described above in the illustration but differ in two decisive aspects:

First of all, a kernel state machine still strictly follows the data flow model described above for kernels and only uses the state machine APIs to simplify a complex control block. The state machine embedded in a kernel is executed in every tick completely aligned to the surrounding kernel. It usually receives an input in every tick and produces an output in every tick. Consequently, kernel scheduling directly determines state machine scheduling.

In contrast, the developer can use a manager state machine if the regular data flow model does not fit the application. The manager state machine does not necessarily process an input in every tick and generate an output in every tick. The manager state machine is executed at every system clock and has no surrounding kernel. Hence, a manager state machine does not have a fixed number of cycles to complete, but is rather signal driven. Its scheduling is typically directly linked to the production of an output, the result. The connection to the input or output is

controlled by the state machine itself via signals with requests and responses. The developer has to manually take care of the communication control. In particular, the state machine needs to signal when new data should be read and when valid data is present at the output. An essential part of the design and implementation in this thesis is based on manager state machines.

### 2.3.5 Compilation Tool Flow

Finally, the MaxCompiler offers a supporting tool for the development of the described components, the MaxIDE. The interaction of the components and the compilation flow is shown in Figure 2.6. The kernel and manager compilers (1) translate the respective programs (written in MaxJ, an extended form of Java) into machine language. In the first step, the normal Java compilation with syntax checking takes place. The generated `.class`-files are then executed. The execution includes three intermediate steps: the construction of the data flow graph, the optimization and transformation of the graph, and the generation of the configuration for the FPGA by backend (FPGA vendor) tools (2). The MaxCompiler is able to take care of type conversions and can automatically perform optimizations such as retiming [173], buffer size optimization and pipelining. The execution of these three steps produces a `.max`-file (3) that can either be simulated or completely synthesized in hardware. The `.max`-file contains the configuration of the FPGA as a bitstream and further data that the FPGA needs to control the data transfers. Finally, the host application is compiled, merged with the `.max`-file and linked against the required libraries (4). The resulting binary file (5) can be invoked standalone and the MaxelerOS serves as a bridge between software and hardware, similar to a regular hardware driver.

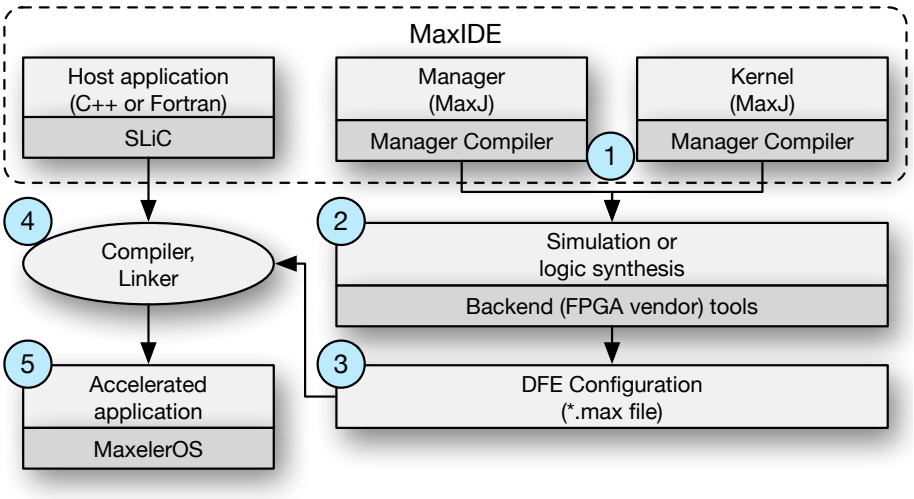



Figure 2.6: Compilation tool flow.

## 2.4 Chapter Conclusion

In this chapter, we provided background information and foundations on reconfigurable computing required for this thesis. The ideas and concepts introduced in the following chapters heavily rely on practical parts and are designed and implemented for FPGAs. We outlined the architecture and building blocks of FPGAs and introduced the MaxCompiler as the main programming model that is used for the actual implementation.

# Chapter 3

# Excursion to Cryptography and Information Security

In this chapter, we will take an excursion to the cryptography and information security domain. The overall ideas for studying general branch-and-bound (B&B) on FPGAs started with the examination of very uncommon problems for FPGAs, namely the so-called cold-boot attacks (CBAs). Cold-boot attacks are part of side-channel attacks against computer systems that rather exploit specific aspects of the whole ecosystem where algorithms are implemented and executed, instead of exploiting an algorithm itself. Side-channel attacks form a very interesting and important area in the information security domain. We analyzed and studied cold-boot attacks for FPGAs and designed and developed very efficient implementations. From the analysis and implementations we were able to generalize the solution to be able to transfer the lessons learned to other branch-and-bound problems outside the specific domain. With this background information in mind, the following structure of this thesis might be easier to assess.

In this chapter, we will give a general motivation and background information required to understand the challenges imposed by our case study. In Section 3.1, we give an introduction to general side-channel attacks and the specialties of cold-boot attacks. We work out three typical phases that are required to spawn a real attack vector. Then, in Section 3.2, we present two distinct error models that are required by the branch-and-bound algorithm used in this thesis to be able to prune the search space. Finally, we describe the functionality of the advanced encryption standard (AES) in Section 3.3 because our case study makes heavy use of its internal operations.

## 3.1 Introduction to Side-Channel Attacks

According to a study by the Ponemon Institute [225], 12,000 laptops are lost at US airports per week. The study states that about 53% of business travelers carry sensitive data. In a later study [226] of the same institute, 275 organizations in Europe were interviewed. The researchers found that about 8% of all laptops in companies are lost during their lifetime. During the twelve-month study period, about 72,000 laptops disappeared. The Ponemon Institute estimates the cost per laptop at about 35,000€. Only a small fraction of this sum is made up of hardware costs. The loss of availability, integrity and, most importantly, confidentiality of information represents the far larger cost share. Even though some definitions and numbers presented by the studies are controversial [32, 217], the overall risk and consequences associated with lost laptops should raise awareness. A recent survey [165] indicates

that most companies have unprotected data and poor information security practices, making them vulnerable to attackers. The consequences of such incidents for a company can include significant financial and customer confidence losses.

The users of modern communication and information systems therefore demand security technologies against the threats of unintentionally loosing or leaking any kind of data. Cryptography and information security techniques offer the key technologies for effectively counteracting threats of integrity and confidentiality. Companies and individuals increasingly encrypt the data stored on hard disk drives by using full disk encryption tools and the data transferred during communication by using secure protocols (for example, HTTPS [235], VPN [62] or WPA2 [170]).

The tools and protocols support different ciphers (for example, AES [12], RSA [237], ECC [151], Serpent [18], or Twofish [243]) in different combinations, configurations and modes. While modern cipher algorithms themself are considered secure, actual implementations have to keep the secret key material in main memory. For efficiency reasons, auxiliary key material is stored in addition to the secret key itself. The key material is derived from the secret key itself and can contain round keys in the case of symmetric ciphers such as AES/Serpent/Twofish or cofactors/modulus in the case of public-key cryptosystems such as ECC/RSA. This additional material is always needed in the encryption and decryption process. It is very sensitive, because it can reveal characteristics of the secret key itself. Keeping the sensitive key material in the main memory of a computer system for efficiency reasons was assumed to be secure. Firstly, the main memory was expected to be volatile and quickly change into a default state erasing its contents when removing the power supply. And secondly, memory cells were expected to be isolated, not interfering each other states allowing privilege separation between different processes.

In recent years, both assumptions have been invalidated by security experts. In the first case, Skorobogatov [255] and Halderman et al. [126] have shown that the memory contents of SRAM and DRAM decay surprisingly slowly over time. The decay can be slowed further by cooling the chips, which opens the possibility to attack the secret keys and thus to circumvent the cryptography and information security procedures. In the second case, Kim et al. [149] first were able to bypass the isolation between memory cells to alter values in order to violate the systems integrity and later Kwong et al. [164] exploit the same *row hammering* technique to also read the memory contents from this side-channel affecting the system's confidentiality.

In the following, we will mainly focus on specific details imposed by cold-boot attacks. The basic ideas, attack vectors and algorithmic challenges are very similar to other side-channel attacks.

### 3.1.1 Cold-Boot Attacks

The cold-boot attack (CBA) is a serious problem for software-based encryption programs. By observing, measuring and combining certain properties, such as patterns in the main memory, the attacker attempts to draw conclusions about parts of the secret key material in the main memory. The cold-boot attack exploits especially the long decay time of memory cells, also called the remanence effect [141].

### 3.1.2 Remanence Effect of Main Memory

In modern computer systems, at runtime the main memory typically contains the programs to be executed accompanying the corresponding data – also known as the von Neumann architecture [276]. The main memory module is an electronic circuit consisting of a large number of cells, whereby each cell essentially encodes a single bit of data. Figure 3.1 shows the most commonly used variant of a single dynamic random access memory (DRAM) cell realized as a one-transistor one-capacitor (1T1C) pair. Conceptually, the capacitor holds a charge to store a binary value (high charge for logic 1 and low for logic 0) and the transistor acts as a controller enabling reads and writes to retrieve and change the value.

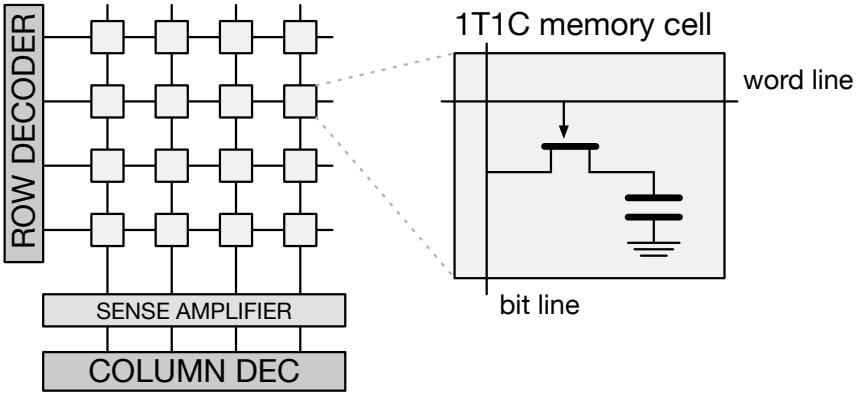



Figure 3.1: Schematic illustration of the memory chip organization realized with one-transistor one-capacitor (1T1C) DRAM memory cells.

The memory cells implemented as static random access memory (SRAM) retain its data while the power supply remains. In contrast, DRAM cells are volatile, which means that the charge in the capacitor will slowly leak away. Depending on the memory chip organization, the cell is hard-wired to either power or ground leading to a decay of the cell to either logic 0 or 1 – the so-called *ground state*. To prevent the cell from losing its stored value and returning to its ground state, it must be refreshed periodically with current. As a result, a DRAM main memory module has a defined maximum refresh interval before the cells start to return to their ground state. As bit errors in the main memory have serious consequences for the program execution and for the stability of an entire system, the manufacturers of DRAM use very short refresh intervals (in the range of 64 milliseconds or less) to keep the probability of unwanted memory decay extremely low. In fact, modern memory cells are even refreshed after read operations to prevent the destruction of data, due to the very cost-efficient nature of the design.

When a system is turned off or the main memory module is suddenly removed, the DRAM cells are not refreshed anymore and the stored data is lost. However, it has been shown by Halderman et al. [126] that the memory cell contents are not lost immediately. The decay can be slow enough to allow (partially) retrieving its contents, especially at low temperatures by artificially cooling the memory chip. This remanence effect is primarily attributed to the high density of memory cells and opens an attack vector on cryptographic applications that keep secret keys in DRAM for encryption and decryption of data.

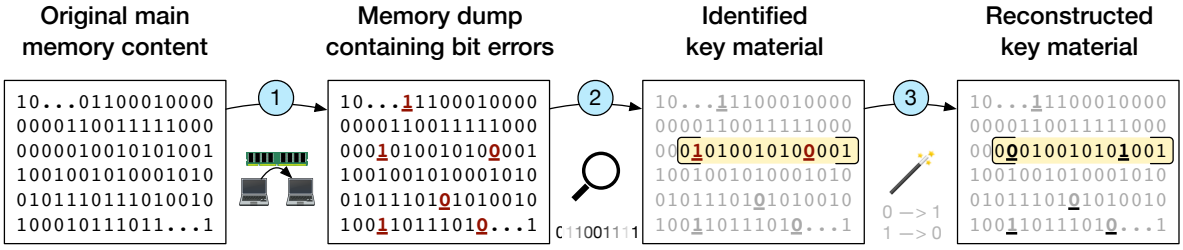



Figure 3.2: Main phases of a cold-boot attack.

### 3.1.3 Attack Vector and Relevance

In a cold-boot attack [47, 16, 218, 211, 250], physical access to the target system by the attacker is a prerequisite. The attacker can get temporary access to the victim's computer (e.g. in the office during lunch break) or steal the victim's laptop at an inattentive or hectic moment. Another prerequisite is that the computer is in an active state (screen lock, standby, hibernate, etc.) at the time of the attack. The victim must have booted up the computer and logged in. Otherwise, the main memory contains no secret key material. The attack vector is depicted in Figure 3.2 and can be divided into three successive phases for better structuring [275]:

1. Acquisition of the main memory contents.
2. Search and extraction of the secret key material.
3. Analysis and reconstruction of the secret key.

The attacker's goal is thus to read the contents of the main memory, to find all the key material in it, and then to find and reconstruct the secret key. In the case of a successful attack, the attacker is able to decrypt the secret data or undermine the secured communication channel.

1 **Acquisition of the Main Memory Contents** The acquisition of the memory content does not require any special hardware or software. Depending on how much time and resources the attacker has at her control and how many countermeasures are expected to be taken by the victim's system, the attacker can choose and adjust a suitable method presented by Halderman et al. [126]. The attacker typically obtains an image of the main memory contents by either rebooting the machine from a USB drive that dumps the contents to a persistent storage, or by physically transplanting the memory modules into another machine that is under the attacker's control. If the reboot process or the transplantation is done quickly, only a small fraction of the bits will have changed their value due to memory decay. The main memory can be strongly cooled during this process, which reduces the expected number of decayed errors. But as this procedure cuts the power to the memory module, the stored contents often contain bit errors. As individual bits can change their stored value, the acquisition of the memory content does not provide an exact copy.

We call the proportion of decayed to total bits the *error rate*. The error rate is highly dependent on the type of the side-channel attack. For cold-boot attacks, the error rate is directly related to the moment of the last refresh of the memory. The longer the memory is not refreshed, the more bits decay. The result of a decay due to a missing refresh is a bit error in the corresponding cell of the memory. Details on the specific behavior and distribution of the decay over time are presented by Halderman et al. [126].

② **Search and Extraction of the Secret Key Material** In the next step, secret key material–in essence, (pseudo) random numbers of 128, 192 or 256 bits length–has to be identified in the memory dump with the typical size of several gigabytes (*key search*).

The secret keys alone are very small and contain a high entropy, which is difficult to search for. However, since the actual encryption and decryption operations require not just the secret key, but also auxiliary key material (for example, round keys), the key material is usually pre-computed and stored as a contiguous block in main memory. By exploiting the publicly known cryptographic structure of a cipher and layout of the key material in memory, the resulting memory image can be searched for sections that could correspond to (decayed) cryptographic keys.

③ **Analysis and Reconstruction of the Secret Key** Finally, the same information about the cipher can be used to correct bit errors in the extracted key material (*key fix*). The key material might contain many redundancies compared to the searched key, because it is derived from it. In this step, the redundancies can be used to detect and correct bit errors imposed by memory decay. If successful, the secret keys can be recovered and the cryptography and information security procedures can be circumvented.

Please note that especially the first two phases might slightly differ for other side-channel attacks, but follow exactly the same principles. For example, the most recent RAMBleed [164] attack uses a specific sequence of main memory accesses to cause bit errors in other locations of the memory than the one accessed. Leveraging those induced bit flips allows the attacker to read portions of the secret keys. This attack can even be performed remotely. The obtained secret keys have also bit flips similar to the decayed values for cold-boot attacks and therefore require same the reconstruction techniques as discussed in phase ③.

In the next section, we show how resulting bit errors caused by the side-channel attack can be modeled. Bit errors typically follow patterns and other dependencies that can be used in the search and the reconstruction processes to prune the search space and recover the secret key. We present two distinct models for bit errors caused by memory decay during cold-boot attacks.

## 3.2 Modeling Bit Errors

As mentioned earlier in Section 3.1.2, DRAM contents will gradually decay when the memory module is not refreshed. Halderman et al. [126] have investigated the main memory decay using different DRAMs (architectures, manufacturers, models, etc.) and test scenarios (with/without cooling, refresh at different time intervals, etc.) and

found some interesting decay patterns. Two of the most important findings can be summarized as follows:

1. Most bits decay continuously into the ground state of the main memory cell. The probability of this observation increases the longer the main memory is not refreshed. With a sufficiently long period without refreshing the cells, the entire memory cells decays to the ground state.

2. Only a very small fraction of bits (about 0.1%) flips in the opposite direction of the ground state. Wang [281] found for other DRAMs a slightly higher fraction of opposite bit flips, but confirms the overall observation.

The decay of memory is extremely asymmetric: this means that bit flips from 0 to 1 (denoted as $0 \rightsquigarrow 1$) and bit flips from 1 to 0 (denoted as $1 \rightsquigarrow 0$) occur with different probabilities depending on the ground state of the memory cells. Consequently, the number of errors is dominated by the decay of the bits into the ground state (with about 99.9%), compared to the rare decay in the opposite direction (remaining 0.1%). Hence, Halderman et al. proposed to completely ignore the unlikely flip direction opposite the ground state of the memory cell and assume only bit errors in the dominant direction. The model following this observation is called perfect asymmetric decay.

### 3.2.1 Perfect Asymmetric Decay

Perfect asymmetric decay (PAD) assumes that only the dominant decay into the ground state of the cell exists and neglects the other 0.1% of the bit errors that can occur in real scenarios. This assumption offers very elegant and efficient implications in the search and recovery of secret keys. The most important implication is the *known bit*.

**Lemma 3.1 (Known Bit)** Under the assumption of perfect asymmetric decay, only bit flips in the direction of the ground state of the cell $M$ exist. Consequently, all bits with the opposite value of the ground state are considered correct and can be directly determined. Equation 3.1 represents the relationship between the ground state and the known bit.

$$\texttt{KNOWN\_BIT} = \begin{cases} 1, \text{if ground state of memory cell } M = 0 \\ 0, \text{if ground state of memory cell } M = 1 \end{cases} \tag{3.1}$$

If, for example, $d = \texttt{0x94} = 148_{10} = 1001\,0100_2$[1] is the obtained decayed byte and $M = 0000\,0000_2$ is the ground state of the considered cell. Then every bit with the value 1 (here $\underline{1}00\underline{1}\,0\underline{1}00_2$) is considered correct (or *known*) because a decay from $0 \rightsquigarrow 1$ cannot take place in the PAD model. Consequently, all divergent bit values ($\neq 1$) at these positions can be immediately excluded (or bounded) in the recovery process of the secret key, because they are not compatible to the model.

The above considerations show that PAD provides a very simple way to determine the set of compatible candidates at an early stage. The next lemma shows how a candidate (a possible correct part of the secret key) can be checked.

[1] Throughout the thesis, we will use numbers to different base systems. The general format is $number_{base}$. If the base is omitted, the decimal system (base 10) is used. Numbers with the prefix `0x` or in `monospace font` are hexadecimal (base 16).

**Lemma 3.2 (Compatibility of Bytes with PAD)** If $c$ is a candidate byte, $d$ the decayed byte and $M$ the ground state, then $c$ and $d$ are considered compatible if the following equation is holds:

$$\texttt{is\_compatible}_{\texttt{PAD}}(c,\ d,\ M) = (c \oplus d) \wedge (d \oplus M) = 0$$

For example, the candidate byte $c = 1110\,1011_2$ cannot decay to an observed byte $d = 1101\,0011_2$ when $M = 0000\,0000_2$ is the ground state, because the fourth bit must have flipped from 0 to 1:

$$\begin{aligned}&\texttt{is\_compatible}_{\texttt{PAD}}(c = 1110\,1011,\ d = 1101\,0011,\ M = 0000\,0000) \\ &= (1110\,1011 \oplus 1101\,0011) \wedge (1101\,0011 \oplus 0000\,0000) = 000\underline{1}\,0000 \neq 0\end{aligned} \tag{3.2}$$

On the other hand, $c = 1110\,0011_2$ and $d = 1100\,0000_2$ with $M$ unchanged are compatible:

$$\begin{aligned}&\texttt{is\_compatible}_{\texttt{PAD}}(c = 1110\,0011,\ d = 1100\,0000,\ M = 0000\,0000) \\ &= (1110\,0011 \oplus 1100\,0000) \wedge (1100\,0000 \oplus 0000\,0000) = 0\end{aligned}$$

The following, very important observation should be made clear by the formula and the examples: the compatibility between two bytes is decided in this model completely independent of past or following decayed bytes. It is a decision based on *local* knowledge ($d$ and $M$), can be evaluated using simple operations and is therefore very suitable for hardware.

However, the PAD model fails for certain problem instances because in reality bit errors in both directions are possible. The next model includes this observation and is called *expected value as threshold*.

### 3.2.2 Expected Value as Threshold

In our previous work [2], we proposed a threshold-based error model that takes both error directions into account and generalizes Halderman's observations. Our expected value as threshold (EVT) model separates the overall error rate of bit flips $r$ (see Section 3.1.3) into bit flips in each direction $1 \rightsquigarrow 0$ (denoted as $r_{1\rightsquigarrow 0}$) and $0 \rightsquigarrow 1$ (denoted as $r_{0\rightsquigarrow 1}$):

$$r = r_{1\rightsquigarrow 0} + r_{0\rightsquigarrow 1}$$

The perfect asymmetric decay (PAD) model is subsumed in this model when $r_{1\rightsquigarrow 0} = 0$ or $r_{0\rightsquigarrow 1} = 0$: i.e. reducing to a decay in only one direction. With given $r_{1\rightsquigarrow 0}$ and $r_{0\rightsquigarrow 1}$ rates, we can compute the expected number of bit flips in each direction, denoted as $n_{1\rightsquigarrow 0}$ and $n_{0\rightsquigarrow 1}$, by multiplying the rates with the total number of bits $N$ in the full key material:

$$\begin{aligned}n_{1\rightsquigarrow 0} &= r_{1\rightsquigarrow 0} \cdot N \\ n_{0\rightsquigarrow 1} &= r_{0\rightsquigarrow 1} \cdot N\end{aligned}$$

Using this information, the next lemma shows how a candidate can be checked with the EVT model to estimate if the candidate is a possibly correct part of the secret key:

**Lemma 3.3 (Compatibility of Bytes with EVT)** We compute the number of bits that would have flipped from 1 to 0 and from 0 to 1 for a candidate byte $c$, denoted by $c_{1\rightsquigarrow 0}$ and $c_{0\rightsquigarrow 1}$, and compare them against the number of expected bit errors *for the entire key material*. If one of the numbers exceeds its expected value, the candidate is not compatible, as outlined in Listing 3.1.

```
1  # Initialization with given r_{1~>0}, r_{0~>1} and N.
2  n_{1~>0} ← r_{1~>0} · N
3  n_{0~>1} ← r_{0~>1} · N
4  # Global information storing the already consumed bit flips.
5  n̄_{1~>0} ← 0
6  n̄_{0~>1} ← 0
7
8  is_compatible_EVT(c, d, M) :
9    c_{1~>0} ← compute_bit flips_1_to_0(c, d, M)
10   c_{0~>1} ← compute_bit flips_0_to_1(c, d, M)
11
12   # Check if bit flips induced by candidate exceed expected numbers.
13   if ( (c_{1~>0} + n̄_{1~>0} ≤ n_{1~>0}) ∧ (c_{0~>1} + n̄_{0~>1} ≤ n_{0~>1}) )
14     # Candidate is compatible. Update consumed bits.
15     n̄_{1~>0} ← c_{1~>0} + n̄_{1~>0}
16     n̄_{0~>1} ← c_{0~>1} + n̄_{0~>1}
17     return true
18   else :
19     return false
```

Listing 3.1: Compatibility check with the EVT error model.

Compared to the PAD model, the compatibility can no longer be decided locally between the decayed byte and the ground state. The bit errors from past and following bytes have an effect on each other and must always be carried throughout each compatibility check. This approach therefore requires *global* knowledge for the decision of the compatibility and has a high computational intensity compared to the PAD model, as will be shown later.

We also give an example for the compatibility check with the EVT model to create a better intuition. We assume the following scenario: The overall error rate is given by $r = 5\%$ with a ground state $M = 0$. Separating the error rate into the individual asymmetric direction of bit flips gives $r_{1\rightsquigarrow 0} = 4.9\%$ and $r_{0\rightsquigarrow 1} = 0.1\%$. To compute the expected number of bit flips, we multiply the individual rates with the total number of bits in the full key schedule ($N = 1408$ for AES-128), which gives us $n_{1\rightsquigarrow 0} \approx 69$ and $r_{0\rightsquigarrow 1} \approx 1$. This means that candidate bytes are compatible as long as any of these numbers is not exhausted. This is especially critical for $r_{0\rightsquigarrow 1}$, where only one bit flip opposite the ground state will be tolerated.

We recall the candidate byte from Equation 3.2. In the PAD error model the candidate $c = 1110\,1011_2$ cannot decay to the observed byte $d = 1101\,0011_2$, because the fourth bit (underlined) must have flipped against the ground state from 0 to 1. However, with the EVT this candidate is compatible, as shown in the compatibility check in Listing 3.2 .

The expected value as threshold model considers bit errors in both directions and is therefore the better model for practical considerations. However, it has some important differences to note. On the one hand, it is very difficult to determine the required thresholds (error rates in each flip direction) correctly. Due to the sensitivity

```
1  # Initialization expected number of errors:
2  #    r = r_{1~>0} + r_{0~>1} = 0.049 + 0.001 = 0.05  and  N = 1408.
3  n_{1~>0} ← 69  # = 0.049 · 1408
4  n_{0~>1} ← 1   # = 0.001 · 1408
5  # So far, no bit flips have been consumed.
6  n̄_{1~>0} ← 0
7  n̄_{0~>1} ← 0
8
9  # Compatibility check of first candidate.
10 is_compatible_EVT(c = 1110 0011, d = 1101 0011, M = 0000 0000)
11 #  Returns true, candidate is compatible.
12 #  Candidate deviates at three bits.
13 #  Number of consumed bits is updated.
14 n̄_{1~>0} ← 2 # 67 more bits can flip in this direction.
15 n̄_{0~>1} ← 1 # No more bit flips in this direction will be tolerated.
```

Listing 3.2: Compatibility check with the EVT error model applied to example.

of the model, the thresholds must be as accurate as possible, because invalid candidates pass the compatibility check if the threshold is too high and valid candidates do not pass the bound if the threshold is too low. From a statistical point of view, the search should not be bounded when the number of bit flips exceeds the expected number of bit flips, but instead when exceeding a certain number of errors that is unlikely to occur based on the given expected values. For simplicity, we assume that this deviation is already considered in the given error rates.

In this thesis, we will focus on the search and reconstruction of advanced encryption standard (AES) keys. Therefore, we describe the required functionality of AES in the next section, explain the basic cryptographic principles behind the individual operations and indicate the requirements for an implementation in hardware.

## 3.3 Advanced Encryption Standard

The advanced encryption standard (AES) [72] is one of the most used encryption algorithms and the default block cipher in a variety of systems and tools, e.g. disk encryption [103], WLAN, HTTPS, SSH [288] and VoIP [114]. AES is the successor of the data encryption standard (DES) [78, 66]. From 1976, DES has been the official encryption standard of the US government for over 20 years. To stop the increasing successful attacks against DES and to increase the key length of effectively only 56 bits, DES was executed three times in a row with three different keys (called Triple-DES or 3DES [251]). This temporary solution worked for that time, but was much slower. As a result, in 1997 a call for proposals for a successor of DES took place and yielded the algorithm Rijndael by Joan Daemen and Vincent Rijmen as the winner against 14 competing algorithms. Other finalists from the selection process are also known and used ciphers, for example, Twofish [243] or Serpent [18]. AES is a symmetric block cipher, has a fixed block size of 128 bits and features a variable key length of 128, 192 or 256 bits (each referred to as variant AES-128, AES-192 or AES-256).

### 3.3.1 Key Schedule: Secret Key and Round Keys

The AES algorithm requires a set of round keys to encrypt and decrypt data. The way in which these are generated is decisive for the security of the cryptographic

process and defined by the AES key expansion function [72]. The secret master key is used as the first round key from which all other round keys are derived. The so-called *key schedule* is comprised of the master key and the (key size dependent) number of other round keys. The key schedule is the secret key material that is the target for side-channel attacks. The number of rounds varies and depends on the key length. Table 3.1 shows the common key sizes $l$ with the number of rounds $r$.

| **number of round keys** $r$ | **key size** $l$ **in bytes (bits)** | **key schedule in bytes (bits)** |
|---|---|---|
| 10 | 16 (128) | 176 (1,408) |
| 12 | 24 (192) | 312 (2,496) |
| 14 | 32 (256) | 480 (3,840) |

Table 3.1: Number of rounds $r$ for key size $l$.

| **w** / **b** / **r** | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 3 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 4 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 5 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 6 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
| 8 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 |
| 9 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 |
| 10 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 |

w: word, b: byte, r: round

Table 3.2: Overall structure of an AES-128 key schedule $KS$ and illustration of the ascending addressing scheme.

**Notation and Addressing Scheme** Table 3.2 shows the overall schematic structure of an AES-128 key schedule $KS$ consisting of 11 round keys, including the secret master key in round 0 (highlighted in blue). A real AES-128 key schedule is shown in Table 3.4. To be able to reference specific parts of the key schedule, we define a set of operations and their notation:

1. Each round $r \in \{0, \ldots, 10\}$ consists of 16 bytes, divided into four words $w \in \{0, \ldots, 3\}$ of four bytes $b \in \{0, \ldots, 3\}$ each.

2. A specific byte can therefore be addressed uniquely via $KS_{r,w,b}$. The asterisk character $*$ can be used as a wildcard for certain information.

3. The operations $KS_{r,w}$ (short form of $KS_{r,w,*}$) and $KS_r$ (short form of $KS_{r,*,*}$) retrieve a 4-byte word or a 16-byte round, respectively. An alternative, flat addressing is also possible to retrieve a single byte with $KS[i]$, $i \in \{0, 1, \ldots 175\}$ or a range or bytes with $KS[i \ldots j]$, $i, j \in \{0, 1, \ldots 175\}$, $i \leq j$. In Table 3.4 the ascending addressing positions (0–175) are shown.

4. Both notations can be combined to retrieve the four bytes of a complete word containing the byte $KS[i]$ via $KS_w[i]$ or the 16 bytes of a complete round containing the byte $KS[i]$ via $KS_r[i]$.

5. Finally $|KS|$ refers to the size of (parts of) the key schedule in bytes.

For example, $KS_0$ and $KS_{0,*,*}$ and $KS[0\ldots15]$ all retrieve the 16 bytes of the secret master key in round 0 highlighted in blue. $KS_{0,0}$ and $KS_{0,0,*}$ retrieve the 4 bytes of word 0 in round 0. $KS_{8,2,3}$ references the same byte as $KS[139]$. Finally $KS[139]_w$ references the same word as $KS[136\ldots139]$ with a size of $|KS_w[139]| = 4$ bytes.

### 3.3.2 Secret Key Expansion

Figure 3.3 and Figure 3.4 show how the 10 round keys in the AES-128 key schedule are generated from the secret master key in round 0 using two different functions. The first word of each round key is computed by applying a *complex* expansion using word 0 and word 3 of the preceding round key, including a non-linear substitution (`SBox`) and introducing a round-specific constant (`RCon`). The `SBox` and `RCon` are publicly known constant lookup tables [72], simply mapping one value to another (see Appendix A). The remaining words 1..3 of each round key are computed by applying a *simple* XOr expansion between the same word of the previous round key and the previous word of the same round key. Both expansions are bijective, which simply means that the computations can also be performed backwards: e.g. if word 3 and 1 are known then word 2 can be computed using backward expansion.

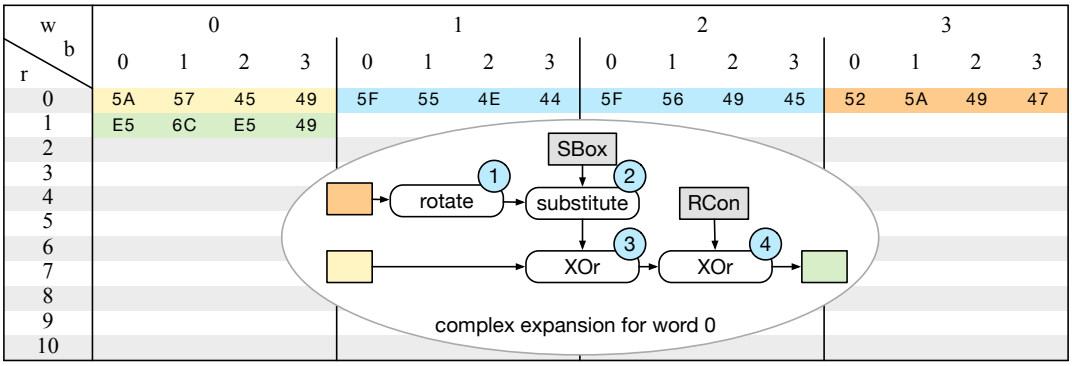



Figure 3.3: Complex AES key expansion operations to generate the second round key for word 0. The next rounds follow the same principle.

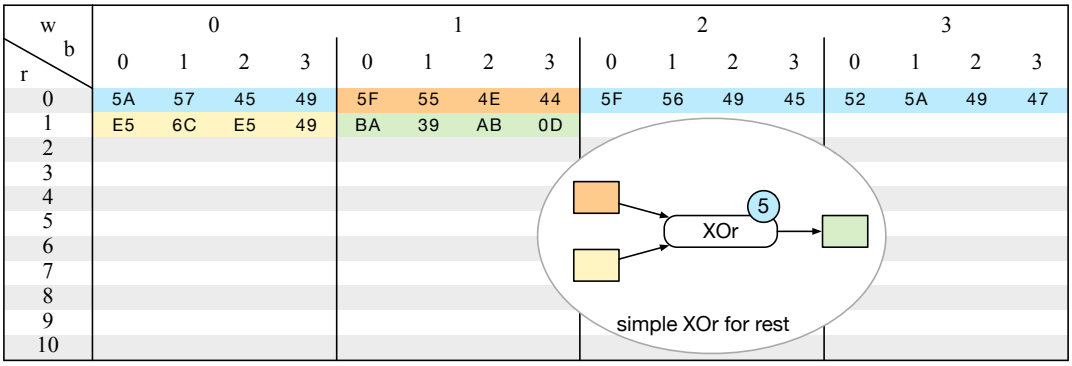



Figure 3.4: Simple XOr AES key expansion operations to generate the second round key for all remaining words of round 1. The next rounds follow the same principle.

**Example** Using the described operations we will expand a secret key to the full key schedule $KS$ to create a good understanding of the underlying operations. The secret key is:

$$key = [\texttt{5A}, \texttt{57}, \texttt{45}, \texttt{49}][\texttt{5F}, \texttt{55}, \texttt{4E}, \texttt{44}][\texttt{5F}, \texttt{56}, \texttt{49}, \texttt{45}][\texttt{52}, \texttt{5A}, \texttt{49}, \texttt{47}] = KS_0$$

The first round key $KS_0$ is the secret master key itself shown in Table 3.3. Each individual position designates a byte, therefore it is a 16-byte or 128-bit key (AES-128). If the key is too short, a padding symbol is appended at the end to fill the missing positions. The numbers used in this example are hexadecimal, although the leading prefix `0x` is mostly truncated.

| w | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b / r | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | 5A | 57 | 45 | 49 | 5F | 55 | 4E | 44 | 5F | 56 | 49 | 45 | 52 | 5A | 49 | 47 |

Table 3.3: Initial key schedule consisting of the secret master key $KS_0$ in round 0.

First, the second round key $KS_1$ is expanded in five steps from the secret key in round 0. The steps ① to ④ describe the complex expansion of the first word $KS_{0,0}$ (see Figure 3.4), while step ⑤ applies the simple expansion using the XOr operator (symbol $\oplus$) of the remaining words $KS_{0,1}$, $KS_{0,2}$ and $KS_{0,3}$ (see Figure 3.3).

① **complex: rotate** The last word $KS_{0,3}$ is taken from the previous round key and rotated. The cyclic permutation shifts all elements left and places the first byte at the end position:

$$r = rotate(KS_{0,3}) = rotate([\texttt{52}, \texttt{5A}, \texttt{49}, \texttt{47}]) = [\texttt{5A}, \texttt{49}, \texttt{47}, \texttt{52}] \tag{3.3}$$

② **complex: substitute** – Next, each byte must be replaced by the corresponding value in the SBox (see Appendix A). This operation is called `SubBytes`.

$$s = SubBytes(r) = SubBytes([\texttt{5A}, \texttt{49}, \texttt{47}, \texttt{52}]) = [\texttt{BE}, \texttt{3B}, \texttt{A0}, \texttt{00}] \tag{3.4}$$

③ **complex: XOr** The intermediate result $s$ from the previous step is now XOred to the first word $KS_{0,0}$. The XOr operation is very fast and requires a small number of gates in hardware. Basically it is a simple bit flipper with minimal hardware and no unwanted carry flag.

$$t = s \oplus KS_{0,0} = [\texttt{BE}, \texttt{3B}, \texttt{A0}, \texttt{00}] \oplus [\texttt{5A}, \texttt{57}, \texttt{45}, \texttt{49}] = [\texttt{E4}, \texttt{6C}, \texttt{E5}, \texttt{49}] \tag{3.5}$$

④ **complex: round constant** – The result $t$ is now XOred with the round constant in RCon. The constant varies from round to round. The complete lookup table of all round constants can also be found in Appendix A. The result is the first word of the next round key $KS_{1,0}$.

$$KS_{1,0} = t \oplus rcon(1) = [\texttt{E4}, \texttt{6C}, \texttt{E5}, \texttt{49}] \oplus [\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00}] = [\texttt{E5}, \texttt{6C}, \texttt{E5}, \texttt{49}] \tag{3.6}$$

⑤ **simple** – The remaining words $KS_{1,1}$, $KS_{1,2}$, $KS_{1,3}$ are much easier to calculate. To do this, the previous word is XOred to the same word of the previous round.

$$
\begin{aligned}
KS_{1,1} &= KS_{1,0} \oplus KS_{0,1} = [\texttt{E5},\texttt{6C},\texttt{E5},\texttt{49}] \oplus [\texttt{5F},\texttt{55},\texttt{4E},\texttt{44}] = [\texttt{BA},\texttt{39},\texttt{AB},\texttt{0D}] \\
KS_{1,2} &= KS_{1,1} \oplus KS_{0,2} = [\texttt{BA},\texttt{39},\texttt{AB},\texttt{0D}] \oplus [\texttt{5F},\texttt{56},\texttt{49},\texttt{45}] = [\texttt{E5},\texttt{6F},\texttt{E2},\texttt{48}] \\
KS_{1,3} &= KS_{1,2} \oplus KS_{0,3} = [\texttt{E5},\texttt{6F},\texttt{E2},\texttt{48}] \oplus [\texttt{52},\texttt{5A},\texttt{49},\texttt{47}] = [\texttt{B7},\texttt{35},\texttt{AB},\texttt{0F}]
\end{aligned} \tag{3.7}
$$

All five steps together result in the first round key $KS_1$. For the remaining rounds the same sequence of operations (steps ① to ⑤) takes place and results in the next round key and so on. The fully expanded key schedule $KS$ for the example is shown in the Table 3.4. The example has a 16-byte (128-bit) key and therefore 10 round keys.

| w | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b / r | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | 5A | 57 | 45 | 49 | 5F | 55 | 4E | 44 | 5F | 56 | 49 | 45 | 52 | 5A | 49 | 47 |
| 1 | E5 | 6C | E5 | 49 | BA | 39 | AB | 0D | E5 | 6F | E2 | 48 | B7 | 35 | AB | 0F |
| 2 | 71 | 0E | 93 | E0 | CB | 37 | 38 | ED | 2E | 58 | DA | A5 | 99 | 6D | 71 | AA |
| 3 | 49 | AD | 3F | 0E | 82 | 9A | 07 | E3 | AC | C2 | DD | 46 | 35 | AF | AC | EC |
| 4 | 38 | 3C | F1 | 98 | BA | A6 | F6 | 7B | 16 | 64 | 2B | 3D | 23 | CB | 87 | D1 |
| 5 | 37 | 2B | CF | BE | 8D | 8D | 39 | C5 | 9B | E9 | 12 | F8 | B8 | 22 | 95 | 29 |
| 6 | 84 | 01 | 6A | D2 | 09 | 8C | 53 | 17 | 92 | 65 | 41 | EF | 2A | 47 | D4 | C6 |
| 7 | 64 | 49 | DE | 37 | 6D | C5 | 8D | 20 | FF | A0 | CC | CF | D5 | E7 | 18 | 09 |
| 8 | 70 | E4 | DF | 34 | 1D | 21 | 52 | 14 | E2 | 81 | 9E | DB | 37 | 66 | 86 | D2 |
| 9 | 58 | A0 | 6A | AE | 45 | 81 | 38 | BA | A7 | 00 | A6 | 61 | 90 | 66 | 20 | B3 |
| 10 | 5D | 17 | 07 | CE | 18 | 96 | 3F | 74 | BF | 96 | 99 | 15 | 2F | F0 | B9 | A6 |

Table 3.4: All round keys for the example. The secret master key is in round 0 and the expanded round key is in round 1.

### 3.3.3 Fundamental Cryptographic Principles

This section describes the three basic ideas [236, 257, 118] behind cryptosystems and explains how they are manifested in AES by encrypting some input data with the key schedule generated in the previous section. AES works round-based for encryption and decryption. This means that the same operations are applied several times in a row to the data to be encrypted or decrypted. The following 16-byte input serves as an example that will be encrypted using the key schedule generated in the previous section:

$$
input = [\texttt{41},\texttt{4E},\texttt{47},\texttt{52}][\texttt{49},\texttt{46},\texttt{46},\texttt{5F}][\texttt{4D},\texttt{4F},\texttt{52},\texttt{47}][\texttt{45},\texttt{4E},\texttt{21},\texttt{2A}] \tag{3.8}
$$

If the input would not correspond to the full block size of 16 bytes, again a padding symbol is appended. We will encrypt the input with the first round key; the remaining rounds follow the same sequence. First, the input is XORed with the secret key in round 0 of the key schedule. The result is:

$$
\begin{aligned}
input_0 &= input \oplus KS_0 \\
&= [\texttt{41},\texttt{4E},\texttt{47},\texttt{52}][\texttt{49},\texttt{46},\texttt{46},\texttt{5F}][\texttt{4D},\texttt{4F},\texttt{52},\texttt{47}][\texttt{45},\texttt{4E},\texttt{21},\texttt{2A}] \\
&\quad \oplus [\texttt{5A},\texttt{57},\texttt{45},\texttt{49}][\texttt{5F},\texttt{55},\texttt{4E},\texttt{44}][\texttt{5F},\texttt{56},\texttt{49},\texttt{45}][\texttt{52},\texttt{5A},\texttt{49},\texttt{47}] \\
&= [\texttt{1B},\texttt{19},\texttt{02},\texttt{1B}][\texttt{16},\texttt{13},\texttt{08},\texttt{1B}][\texttt{12},\texttt{19},\texttt{1B},\texttt{02}][\texttt{17},\texttt{14},\texttt{68},\texttt{6D}]
\end{aligned} \tag{3.9}
$$

**Idea 1 – Confusion** The relationship between the encrypted message and the plain text has to be obfuscated. A simple example of confusion is provided by the Caesar cipher [63]. It maps the letters of the ordered alphabet to another letter. Each

letter is moved to the right by a certain number of places and substituted with the letter at this position. The number of shifts to the right is the secret key. The following example in Equation 3.10 shows a mapping with the secret key 5.

$$\begin{array}{cccccccc} A & B & C & D & E & \dots & Z & \\ F & G & H & I & J & \dots & D & \hookleftarrow \end{array} \tag{3.10}$$

In AES, the confusion property is ensured by the use of the Rijndael substitution box (SBox). To hide the relationship between individual bytes, each byte is mapped to a byte of the fixed SBox (see Appendix A).

$$\begin{aligned} t &= SubBytes(input_0) \\ &= [\texttt{AF},\texttt{D4},\texttt{77},\texttt{AF}][\texttt{47},\texttt{7D},\texttt{30},\texttt{AF}][\texttt{C9},\texttt{D4},\texttt{AF},\texttt{77}][\texttt{F0},\texttt{FA},\texttt{45},\texttt{3C}] \end{aligned} \tag{3.11}$$

**Idea 2 – Diffusion** Another principle is to change the positions of the letters in the secret text. Ideally, a small change in the secret text leads to many changes in the ciphertext. In AES the diffusion follows from the two operations `ShiftRows` and `MixColumns`. To apply these operations, the 16 bytes are aligned into a $(4 \times 4)$ matrix representation. `ShiftRows` moves the lines of the substituted input $t$ from the last Equation 3.11 ascending to the left and attaches the superimposed elements to the right side again.

$$u = ShiftRows(t) = ShiftRows\left(\begin{pmatrix} \texttt{AF} & \texttt{47} & \texttt{C9} & \texttt{F0} \\ \texttt{D4} & \texttt{7D} & \texttt{D4} & \texttt{FA} \\ \texttt{77} & \texttt{30} & \texttt{AF} & \texttt{45} \\ \texttt{AF} & \texttt{AF} & \texttt{77} & \texttt{3C} \end{pmatrix}\right)$$

$$\begin{aligned} &= \begin{matrix} \overset{0}{\leftarrow} \\ \overset{1}{\leftarrow} \\ \overset{2}{\leftarrow} \\ \overset{3}{\leftarrow} \end{matrix} \begin{pmatrix} & & & \texttt{AF} & \texttt{47} & \texttt{C9} & \texttt{F0} \\ & & [\texttt{D4}] & \texttt{7D} & \texttt{D4} & \texttt{FA} & \\ & [\texttt{77}] & [\texttt{30}] & \texttt{AF} & \texttt{45} & & \\ [\texttt{AF}] & [\texttt{AF}] & [\texttt{77}] & \texttt{3C} & & & \end{pmatrix} = \begin{pmatrix} \texttt{AF} & \texttt{47} & \texttt{C9} & \texttt{F0} \\ \texttt{7D} & \texttt{D4} & \texttt{FA} & [\texttt{D4}] \\ \texttt{AF} & \texttt{45} & [\texttt{77}] & [\texttt{30}] \\ \texttt{3C} & [\texttt{AF}] & [\texttt{AF}] & [\texttt{77}] \end{pmatrix} \\ &= [\texttt{AF},\texttt{7D},\texttt{AF},\texttt{3C}][\texttt{47},\texttt{D4},\texttt{45},\texttt{AF}][\texttt{C9},\texttt{FA},\texttt{77},\texttt{AF}][\texttt{F0},\texttt{D4},\texttt{30},\texttt{77}] \end{aligned} \tag{3.12}$$

This is followed by the second transposition using the function `MixColumns`. The function gets $u$ from Equation 3.12 in matrix notation and performs a matrix multiplication:

$$\begin{aligned} v &= \texttt{MixColumns}(u) \\ &= \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} \texttt{AF} & \texttt{4F} & \texttt{C9} & \texttt{F0} \\ \texttt{7D} & \texttt{D4} & \texttt{FA} & \texttt{D4} \\ \texttt{AF} & \texttt{45} & \texttt{77} & \texttt{30} \\ \texttt{3C} & \texttt{AF} & \texttt{AF} & \texttt{77} \end{pmatrix} = \begin{pmatrix} \texttt{51} & \texttt{03} & \texttt{44} & \texttt{DB} \\ \texttt{83} & \texttt{94} & \texttt{10} & \texttt{64} \\ \texttt{D3} & \texttt{F3} & \texttt{37} & \texttt{DD} \\ \texttt{40} & \texttt{1D} & \texttt{88} & \texttt{01} \end{pmatrix} \end{aligned} \tag{3.13}$$

The calculation rules of the matrix multiplication are over a Galois field [158]. The following rules are applied to multiply the elements $u_i \in u$:

$$\begin{cases} 1 \cdot u_i = u_i \\ 2 \cdot u_i = \begin{cases} 2 * u_i & \text{if } u_i < 128 \\ 2 * u_i \oplus \texttt{0x11B} & \text{else} \end{cases} \\ 3 \cdot u_i = (2 \cdot u_i) \oplus u_i \end{cases} \tag{3.14}$$

Here the $*$ denotes the normal multiplication with natural numbers; otherwise the matrix multiplication needs to applied. The calculation of the bracketed value DB is carried out as an example:

$$\begin{aligned}
\begin{pmatrix} 2 & 3 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \texttt{F0} & \texttt{D4} & 30 & 77 \end{pmatrix}^T &= (2 \cdot \texttt{F0}) \oplus (3 \cdot \texttt{D4}) \oplus (1 \cdot 30) \oplus (1 \cdot 77) \\
&= ((2 * \texttt{F0}) \oplus \texttt{11B}) \oplus ((\texttt{D4} \cdot 2) \oplus \texttt{D4}) \oplus 30 \oplus 77 \\
&= (\texttt{1E0} \oplus \texttt{11B}) \oplus (((2 * \texttt{D4}) \oplus \texttt{11B}) \oplus \texttt{D4}) \oplus 30 \oplus 77 \\
&= \texttt{FB} \oplus ((\texttt{1A8} \oplus \texttt{11B}) \oplus \texttt{D4}) \oplus 30 \oplus 77 \\
&= \texttt{FB} \oplus (\texttt{B3} \oplus \texttt{D4}) \oplus 30 \oplus 77 \\
&= \texttt{FB} \oplus 67 \oplus 30 \oplus 77 \\
&= \texttt{DB}
\end{aligned}$$

The result $v$ from Equation 3.13 is finally XOred with the second round key $KS_1$ of the key schedule to complete the first round of the encryption:

$$\begin{aligned}
input_1 &= v \oplus KS_1 \\
&= [51, 83, \texttt{D3}, 40][13, \texttt{9C}, \texttt{FB}, 05][44, 10, 37, 88][\texttt{DB}, 64, \texttt{DD}, 01] \\
&\oplus [\texttt{E5}, \texttt{6C}, \texttt{E5}, 49][\texttt{BA}, 39, \texttt{AB}, \texttt{0D}][\texttt{E5}, \texttt{6F}, \texttt{E2}, 48][\texttt{B7}, 35, \texttt{AB}, \texttt{0F}] \\
&= [\texttt{B4}, \texttt{EF}, 36, 09][\texttt{B9}, \texttt{AD}, 58, 10][\texttt{A1}, \texttt{7F}, \texttt{D5}, \texttt{C0}][\texttt{6C}, 51, 76, \texttt{0E}]
\end{aligned} \tag{3.15}$$

$input_1$ will then be transformed the same way as $input_0$ starting with Equation 3.9 XOred to the next next round key $KS_1$.

**Idea 3 – Kerckhoffs' Principle** The strength of the cryptosystem should not lie in the secrecy of the encryption process, but rather in the secrecy of the key [63]. The disclosure of the key expansion function and the encryption and decryption processes provides an additional public discussion platform to uncover and eliminate weaknesses of a cipher. As a result, the complete specification of the AES algorithm Rijndael is freely available [72]. There are also robust implementations in hardware and software.

## 3.4 Chapter Conclusion

In this chapter, we took an excursion to the cryptography and information security domain to lay the fundamentals required by the case study used in this thesis. We explained the basic ideas of side-channel attacks in general and cold-boot attacks in particular, which will be analyzed stepwise and accelerated in the next chapters using reconfigurable hardware.

In order to model the bit errors caused by a side-boot attack, we introduced two distinct error models: an idealized error model called perfect asymmetric decay (PAD), which considers only bit errors in one direction ($1 \rightsquigarrow 0$ *or* $0 \rightsquigarrow 1$), and the realistic error model called expected value as threshold (EVT), which supports bit errors in both directions ($1 \rightsquigarrow 0$ *and* $0 \rightsquigarrow 1$). The main difference between the models is that for PAD the compatibility can be computed using only local information, whereas for EVT the bounding function is more complex and requires global state space information that needs to be carried during all compatibility checks.

Finally, we explained the key expansion of the AES cipher and described the cryptographic principles behind the individual operations. We showed that AES has a very simple structure and uniform functionality. While this simple structure is often an expressed criticism, it is an essential basis for its efficiency and popularity.

# Chapter 4

# Intermediate Findings: Identification of Secret Key Material

We started our studies on side-channel attacks by accelerating the algorithm for searching secret key material in a possibly large stream of noisy data with reconfigurable hardware. In this chapter, we present our FPGA-based approach and architecture.

The key search algorithm serves as a good starting point for the rest of the studies presented in this thesis. The procedure gives a good intuition to hardware acceleration with FPGAs, but is algorithmically and computationally less complex than the branch-and-bound algorithmic pattern that is tackled in the following chapters. The approach and results discussed in this chapter are published and presented in a peer-reviewed conference article [3] in the International Conference on Field-Programmable Technology (ICFPT), the premier conference in the Asia-Pacific region on reconfigurable computing.

The remainder of this chapter is structured as follows. In the first Section 4.1, we present the basic ideas of a software implementation for searching secret key material in a data stream containing bit errors acquired by a side-channel attack. Then, in Section 4.2, we present our design and implementation for identifying AES key schedules with reconfigurable hardware. Afterwards, we compare the performance of our accelerator with a CPU implementation and discuss the results in Section 4.3. Finally, we draw a conclusion in Section 4.4.

## 4.1 Basic Idea and Software Approach

For the identification of secret key material in a data stream acquired from the first step of a side-channel attack (see Section 3.1.3) we use the method proposed by Halderman et al. [126]. The algorithm is designed for software and is based on the assumption that the secret key material, for example an AES key schedule, is located contiguously in the data stream. Figure 4.1 depicts the processing idea of the data steam divided into four distinct steps:

(1) In the first step, the method runs through the data stream byte by byte. For each input byte (`5A` at position 0 in the figure), a window of 176 continuous bytes is forming a decayed AES-128 key schedule $D = (D[0], D[1], \ldots, D[175])$. In the figure, $D_0$ (first 16 bytes of round 0, see addressing scheme in Section 3.3.1) represents a possible secret key, while the remaining 160 bytes of

the considered window are the round keys. For each decayed key schedule $D$ the procedure evaluates whether it could be a decayed version of an actual AES key schedule. The vast majority of the examined potentially decayed key schedules strongly differs from a consistent key schedule. Therefore the software variant proposed by Halderman et al. uses an early exit heuristic.

(2) The heuristic examines whether more than eight repetitions of any byte occur in $D$. If the test is positive, the procedure skips the position in question because such an event is very unlikely in a valid key schedule.

(3) Otherwise, this is followed by the second heuristical check of $D$, which is much more computationally intensive than the first one: Using the expansion rule of AES-128 (see Section 3.3.2), for each byte of $D$ a validation byte of a reference key schedule $R$ is calculated. Figure 4.2 illustrates this in detail for round 1 of $R$. Word 0 results from the complex key expansion (steps 1 to 4 of the key expansion function) and the remaining words are computed with a simple XOr expansion (step 5). $R$ is therefore the *expected* key schedule from the considered window $D$ if it would contain no errors.

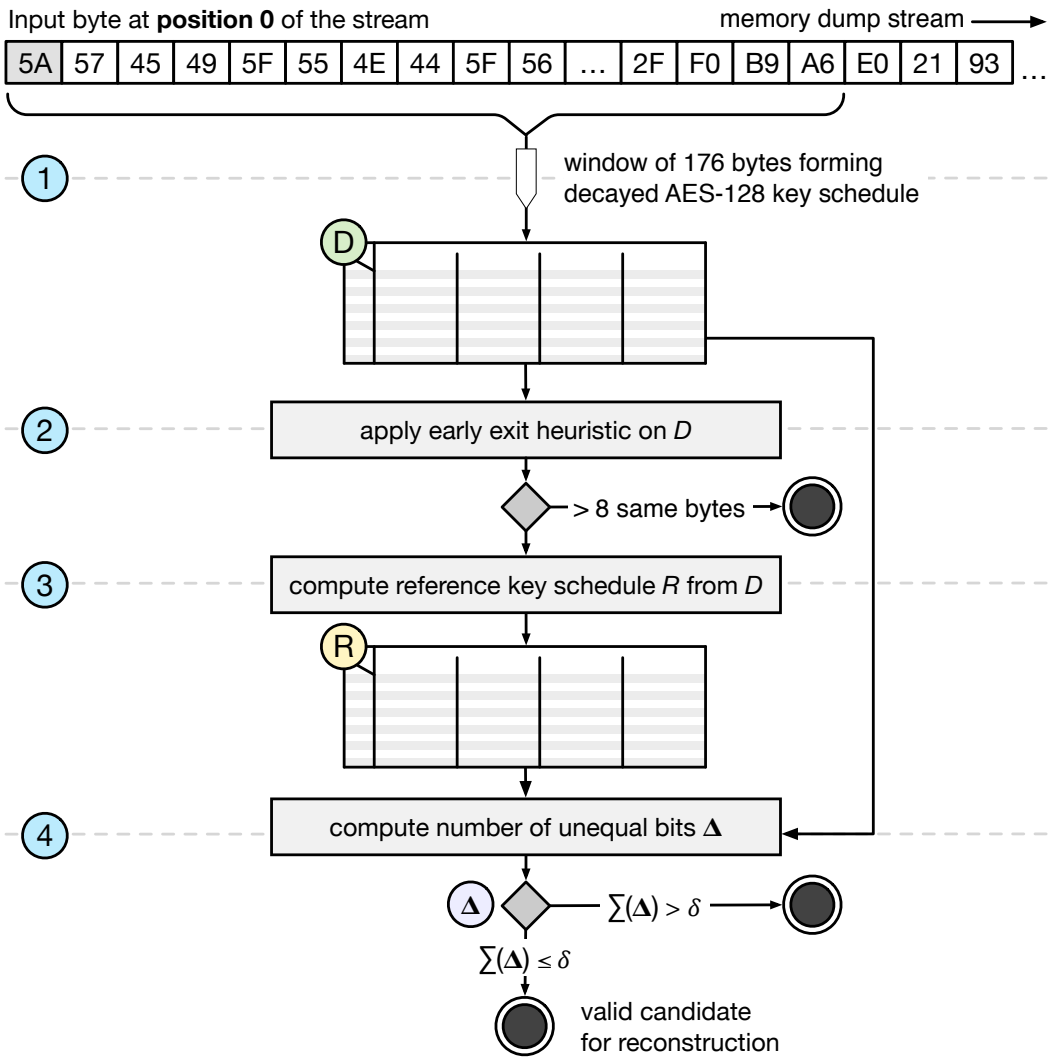
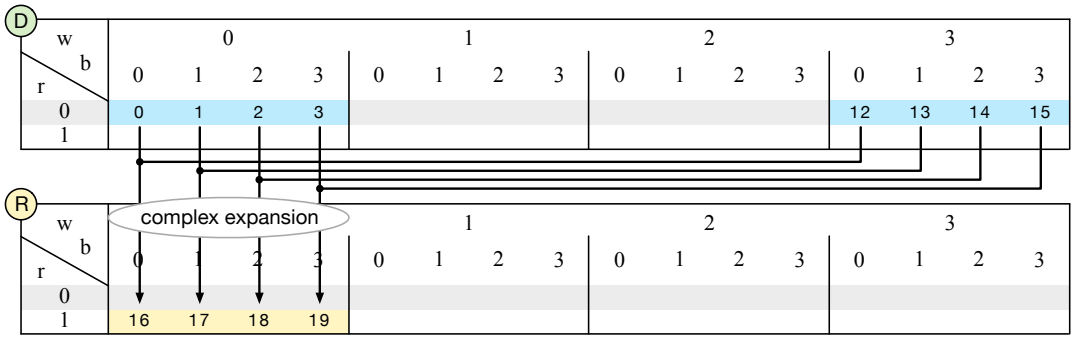


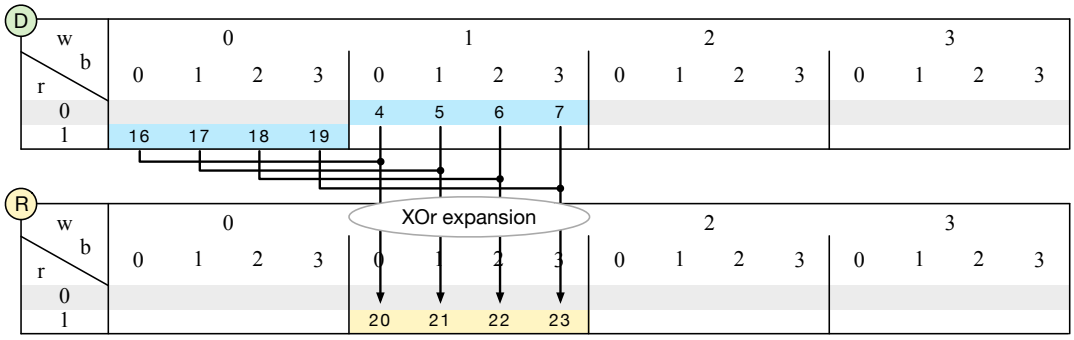

Figure 4.1: A continuous memory stream is processed to identify valid secret key material candidates for AES-128. Here, the first byte at position 0 is evaluated.

④ In the last step, a validation is performed by counting the number of unequal bits between $D$ and $R$. This is done by calculating the Hamming distance $\Delta(D,R) = (\Delta_0, \Delta_1, \ldots, \Delta_{175})$ between each byte of $D$ and $R$. The Hamming distance [36] serves as a measure for the difference between two elements and is defined by the number of unequal bits of a byte, e.g. $\Delta(\texttt{0xDB}, \texttt{0x9D}) = \Delta(1\underline{1}011\underline{0}11, 1\underline{0}011\underline{1}01) = 3$. Each individual Hamming distance corresponds to the number of bit errors between the found decayed byte from $D$ and the expected byte from the reference $R$. To get the number of all errors, all single Hamming distances are summed up $\sum \Delta(D,R)$. If the sum is below or equal a threshold value $\delta$, $D$ is accepted and output as a possible key schedule candidate containing bit errors that need to be corrected.

The threshold value $\delta$ determines the tolerance (in terms of number of bit errors) towards candidates that are accepted as possible key schedules and must therefore be defined in advance. The procedure returns all key schedule candidates that are valid under $\delta$ as output. If the value of $\delta$ is selected too high, too many false-positive candidates are detected. If the value of $\delta$ is selected too low, the right candidates might not be detected. The Hamming distance used in this procedure serves as an error model similar to the described errors models from Section 3.2. In contrast to the PAD and EVT model, the direction of a bit flip is not relevant in this part of the attack and therefore not reflected by this model.




(a) Complex expansion, when word 0 is involved.




(b) XOr expansion for words 1–3. Here word 0 is shown exemplarily.

Figure 4.2: Computation of the reference key schedule $R$ from bytes of the decayed key schedule $D$ using either the complex (in Figure 4.2a) or simple XOr (in Figure 4.2b) expansion function.

The efficiency of these heuristics depends on the structure of the input: If there are many structured areas, many unnecessary computations can be skipped and the speed of processing is increased. However, if the data is highly random, the application spends a lot of time computing the heuristic without excluding candidates. Our analysis of the runtime behavior has shown that the reference software implementation spends 20-40% in these tests.

The core algorithm only utilizes the operations from the AES key expansion, which are inherently suitable for a hardware implementation. The computation of $R$ can be fully parallelized, as well as the validation check with the Hamming distances. Finally, the algorithmic structure follows the streaming data flow model that should also favor FPGAs. Therefore, we designed and implemented this algorithm in reconfigurable hardware, which is described in the next section.

## 4.2 Hardware Implementation

The following sections take up the individual parts of the software approach and describe their conceptual adaption and transformation, such that they are suitable for an implementation on an FPGA and use the possible inherent parallelism of the algorithm and the hardware architecture. The examples and numbers in this section refer to the search for an AES-128 key schedule. We implemented and evaluated two separate kernels, one for AES-128 and one for AES-256. Even though they use a set of the same operations, we separated them in order to avoid configuration overheads inside the critical path of the kernels.

### 4.2.1 Input

The search for possible key schedules can be implemented as a streaming application in hardware, where every new input byte together with previous bytes in the buffer forms a new decayed key schedule $D$ (see Figure 4.1). Thus, even though the used input in every cycle is a data window of 176 bytes (480 bytes for AES-256), only one byte per cycle has to be transferred from the host. The calculation starts immediately after the transfer of the first data window is present on the device. After a relatively short latency, the pipeline is filled and the first value is in the output stream. If there are enough values, the return transfer takes place, while the calculation is continued with new values in parallel.

### 4.2.2 Heuristics

Since we aim for a throughput of one entire key schedule check per clock cycle in our hardware implementation, the described software optimizations (early exit heuristic and threshold $\delta$) are not required. Instead, we always compute the entire error value for each byte position in the data stream and transfer it back to the host. Thus, we can avoid searching with a too low or too high threshold $\delta$ and having to repeat the entire search. In contrast to the software solution, the pipelined search in hardware without heuristics is not data-sensitive and the computation of all error values causes no computational overheads.

### 4.2.3 Computation of Reference Key Schedule

For every byte in the decayed key schedule $D$, we know from the key expansion rule (see Section 3.3.1) how to compute it depending on two previous words. This principle is used to compute the reference words of $R$ from $D$. The expansion rule is applied to every word from round 1 to 10. In the existing reference implementations by Halderman et al. [126], the calculation of all bytes of $R$ is executed sequentially. However, as shown in Figure 4.2, all bytes of $R$ are expanded from already existing values in $D$. Hence, the computations are independent of each other and can be completely parallelized. As a result, the calculation takes only one step in hardware due to the fully spacial parallelization of all bytes of $R$. The upper part of Figure 4.3 schematically indicates the complete parallel calculation of all bytes $R[i]$ from $D$ in one step.

Most of the expansions (all bytes in word 1, 2 and 3) are computed by the simple expansion rule with the help of a XOr gate. In contrast, the bytes in word 0 are computed by the complex expansion rule comprising rotation, substitution and XOr operations (see Figure 3.4, page 21). The substitution operation is implemented as a lookup table in BRAM (LUT BRAM). To maximize throughput, a total of 40 substitution operations per candidate key schedule $R$ have to take place in parallel. Thus, we use 20 dual-ported BRAMs, all filled identically with the content of the substitution table, in order to support 40 parallel requests. Similarly, ten parallel lookups of round constants (RCon) are required as input to the XOr operations, which we implement with lookups to five dual-ported BRAMs.

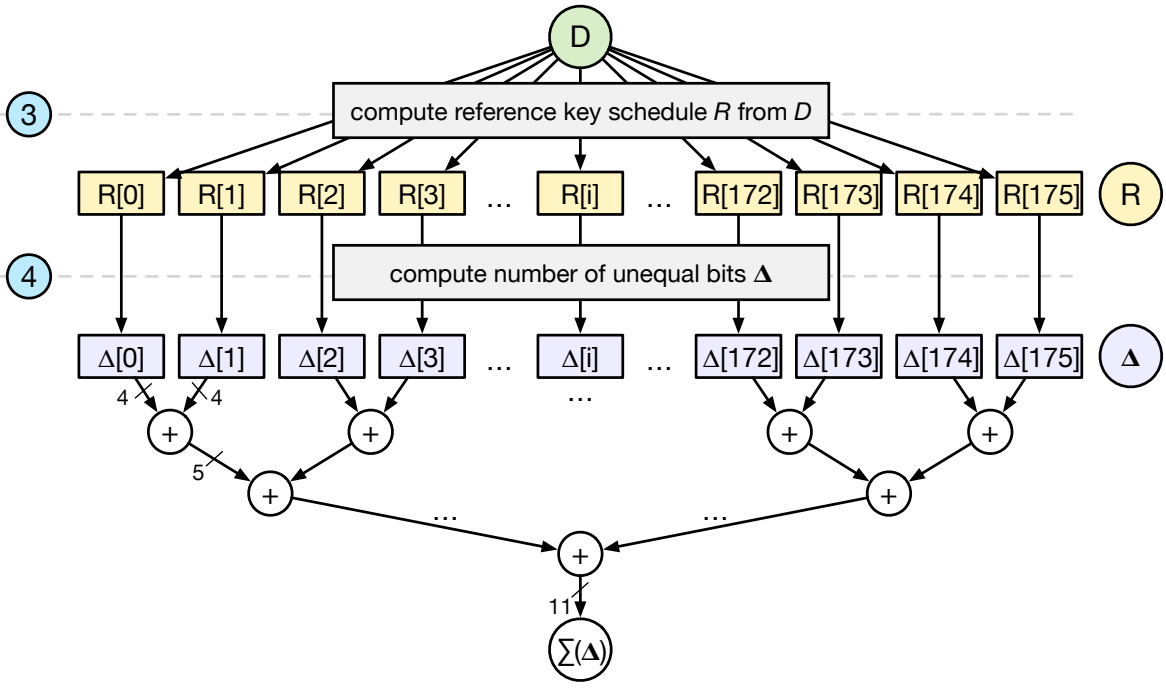



Figure 4.3: Complete parallelization of the computation of the individual bytes of $R$ and the corresponding Hamming distances $\Delta$. The Hamming distances are summed up with a balanced adder tree.

### 4.2.4 Computation of the Hamming Distances

In the next stage, we use the number of unequal bits as the basis of a simplified error model: i.e. we compute the Hamming distance between each pair of reference byte $R[i]$ and the decayed byte $D[i]$. This is also completely parallelizable due to the independence of the individual positions after all $R[i]$ are computed, as is illustrated in the lower part of Figure 4.3. The sum of all individual Hamming distances $\Delta[i]$ is then aggregated by a balanced adder tree, which enables us to use adders with the specific bit-widths required to represent the highest possible error value at each level. The complete adder for AES-128 to compute the aggregated sum $\sum_{i=0}^{176} \Delta(D[i], R[i])$ therefore requires $\lceil log_2(|\Delta|) \rceil = \lceil log_2(176) \rceil = 8$ levels. Each level adds two adjacent elements, so that the next level only has about half as many elements. If $x$ is the number of bytes in the full key schedule, the total number of adders required is $x - 1$: e.g. 175 adders for AES-128. For each decayed key schedule $D$, the resulting aggregated sum $\sum_{i=0}^{176} \Delta(D[i], R[i])$ is returned to the host via an output stream, representing the number of unequal bits. The whole computation is fully pipelined, so our *key search* kernel computes one error sum per position in each cycle after the pipeline is filled. Note that since the complex expansion rule for $R[i]$ takes several cycles, whereas the simple XOr can be performed in a single cycle, the pipeline needs to be balanced. However, the MaxCompiler performs these steps transparently to the developer.

## 4.3 Evaluation

In this section, we evaluate the performance of our hardware accelerator for searching secret key material by comparing the runtime with the software implementation of Halderman et al. [126]. We use both real contents of the main memory after cold-booting a running machine that uses encryption and synthetic random data. The real decayed memory content has been acquired using the tools provided by Halderman et al.[1] The random data has been generated by using the Linux random number generator, i.e., by reading from `/dev/urandom`. The random data is not as strongly affected by the early exit heuristics as real decayed data, because it has no inherent structure in the memory dump and serves for better repeatability of our results.

### 4.3.1 Software Reference

The software implementation provided by Halderman et al. is written in single-threaded C code. It discards candidates by the early exit heuristic or as soon as they exceed the given error threshold $\delta$. Therefore, we execute all tests with three different thresholds:

- No decay: $\delta = 0$. Accepts only candidates without any bit errors. The Hamming distance heuristic in software is maximally effective in this case, because most of the candidates can immediately be discarded.
- Small decay: $\delta = 100$. Accepts only candidates with at most 100 bit errors. For an AES-128 key schedule with a total of 1408 bits, $\delta = 100$ represents a decay of up to 7%.
- Large decay: $\delta = 500$. Accepts candidates with a decay of up to 36%. In this case, the Hamming distance heuristic will be less effective.

[1] https://citp.princeton.edu/research/memory/code/

For all configurations, result printing to standard output (`stdout`) is disabled, as it consumes a significant amount of time for high thresholds because a lot of candidates are detected. The software reference has been executed on the host of the Maxeler system described in detail in Section 5.5.1.

### 4.3.2 Kernel Replication

We synthesized the described hardware design with the MaxCompiler tool chain for Maxeler MPC-C system mentioned above. Even when a throughput of one candidate key schedule $D$ per cycle is reached, the key search can be further parallelized, since each validation of a key schedule is independent of all others. One way to do so is to divide the entire search space into $N$ chunks and let $N$ kernels work on those chunks in parallel. Since potential key schedules could also exist at the borders of chunks, the chunks have to be sufficiently overlapping. This additional parallelization approach is well suited not only for hardware kernels, but also for multi-core in software.

| Number of Parallel Kernels | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Used LUTs (%) | 4.03 | 5.97 | 11.67 | 20.61 |
| Used FFs (%) | 3.13 | 5.47 | 10.17 | 19.56 |
| Used BRAMs (%) | 2.07 | 3.48 | 6.30 | 11.94 |
| Used DSPs (%) | 0.00 | 0.00 | 0.00 | 0.00 |
| Achieved Frequency (MHz) | 250 | 240 | 210 | 170 |

Table 4.1: Synthesis results of replicated AES-128 key search kernels targeting a Virtex-6 SX475T FPGA.

In hardware, we implemented this high-level parallelization strategy and replicated up to eight parallel kernels on one FPGA. Table 4.1 shows that the kernel replication reduces the achievable clock frequencies, but resource utilization still permits higher replication factors. However, our analysis in the next section shows that the current implementation becomes bandwidth limited at this point.

### 4.3.3 Results

All software and hardware tests are performed on one Maxeler system: for the software implementations using its host CPU and for hardware tests using the host CPU plus one of its FPGA accelerator cards.

| threshold $\delta$ | real memory contents AES-128 and AES-256 | synthetic random data AES-128 and AES-256 |
|---|---|---|
| 0 | 215 | 540 |
| 100 | 230 | 605 |
| 500 | 420 | 1114 |

Table 4.2: Runtime in seconds of software key search for 2 GB of input data. The algorithm searches for AES-128 and AES-256 keys with a single run.

Without modifications, the software reference code of Halderman et al. can process at most 2 GB of input data. If the main memory dump is larger, it needs to be manually partitioned into overlapping chunks and executed several times. The measured software runtimes are reported in Table 4.2. Since the execution time for key identification scales almost exactly linearly with input size, we report only the results on 2 GB of data here. The software checks for AES-128 and AES-256 key schedules in one run. Since we implemented AES-128 and AES-256 in different hardware kernels,

we perform two subsequent hardware calls and sum up their runtimes to mimic the functionality of the software implementation. As our hardware implementation computes the entire error value for all candidates, its runtime is independent of the threshold $\delta$.

| | real memory contents | | | synthetic random data | | |
|---|---|---|---|---|---|---|
| **number of kernels** | AES-128 | AES-256 | Σ | AES-128 | AES-256 | Σ |
| 1 | 10.60 | 10.66 | 21.26 | 10.62 | 10.59 | 21.21 |
| 8 | 2.73 | 2.83 | 5.56 | 2.67 | 2.76 | 5.43 |

Table 4.3: Runtime in seconds of hardware key search for 2 GB of input data. The search is separated into two kernels, one for AES-128 and one for AES-256, in order to avoid configuration overheads inside the critical path of the kernels.

The hardware results on 2 GB of input data are summarized in Table 4.3 and contain a design with only one kernel to mimic the single threaded software reference and a design with eight kernels working on different chunks of the entire data. Again, the runtimes scale almost linearly with input size. Our implementation was tested with up to 8 GB of input data. We compute the speedups in terms of improvement in speed of execution compared to the software implementation using the sum of AES-128 and AES-256 execution times and summarize the speedups in Table 4.4.

| | real memory contents | | synthetic random data | |
|---|---|---|---|---|
| **threshold $\delta$** | 1 kernel | 8 kernels | 1 kernel | 8 kernels |
| 0 | 10.1 | 38.7 | 25.5 | 99.4 |
| 100 | 10.8 | 41.4 | 28.5 | 111.4 |
| 500 | 19.8 | 75.5 | 52.5 | 205.2 |

Table 4.4: Improvement in speed of execution (speedup) of hardware key search over software implementation for 2 GB data.

### 4.3.4 Discussion

The single kernel hardware implementation performs between 10x and 52x faster than the software version, depending on the input data and threshold parameters. Notably, this variability comes purely from the software implementation. Since the hardware uses a throughput-optimized deterministic design, its performance is independent of the input data and any error threshold. Actually, checking close to 2 billion candidate key schedules in 2 GB of data at a clock frequency of 250 MHz and a throughput of one candidate per cycle let us expect a runtime of 8s in the best case. Our measured runtimes of 10.6s comes close and includes the runtimes of the host code and for streaming the data between CPU and FPGA, where it is not perfectly overlapped with the computation times.

Replicating the kernel eight times yields an additional speedup of 3.85x over the single kernel hardware implementation. Considering the replication factor and clock frequencies, we would have expected a speedup of 5.4x in compute throughput. Inspecting the discrepancy, we note that the result output for 2 GB of input data (one byte per position) is 4 GB (two-byte short integer for the error sum). At a total runtime of 2.75s, this corresponds to a bandwidth of 1.45 GB/s, which approaches the 2 GB/s bandwidth that the PCIe 2.0 connection to the FPGA card can deliver. Thus, for further acceleration, we need to overcome this bandwidth limitation, for

example by filtering results on the FPGA or by encoding them more efficiently, possibly with some form of compression.

The high-level parallelization strategy applied to hardware can of course also be used in software. Preliminary tests show that on physical CPU cores almost perfect scaling is possible, whereas when using simultaneous multithreading hardly any additional speedup is achieved without further optimizations. However, even when assuming perfect scaling on all 12 physical cores of our test machine, still 4 to 17 such high-end server systems would be required to match the performance of a single FPGA accelerator card.

## 4.4 Chapter Conclusion

In this chapter, we have presented our intermediate findings on the acceleration of the identification of secret key material in a stream of erroneous data using an FPGA. Our hardware implementation achieves speedups of up to 205x compared to the software reference. In order to make the data flow of our hardware implementation regular and to decouple the calculations from the user-defined threshold $\delta$, we changed the design to a streaming application. The new design provides the total error sum $\sum \Delta(D, R)$ for every position in the data stream. The advantage of this redesign is that the application is independent of $\delta$ and that the size of the input corresponds to the size of the output times two. The host application can optionally analyze the complete output after the calculation and, depending on the actual output, filter the most promising candidates.

Once a potentially decayed key schedule is identified as a promising candidate, the actual secret key needs to be reconstructed by correcting the bit errors caused by the side-channel attack. While searching secret key material served as a good starting point to introduce the case study and hardware acceleration with FPGAs, key reconstruction is essentially a branch-and-bound tree search procedure. The computation is highly irregular and thus not a natural fit for FPGAs. This problem will be addressed and accelerated stepwise in the following chapters.

# Chapter 5

# Branch-and-Bound with Reconfigurable Hardware

In this chapter, we present the main concepts for a general hardware design for using branch-and-bound (B&B) with FPGAs. Following our cross-cutting case study of side-channel attacks, we discussed in Chapter 3 that the key material acquired during an attack probably contains bit errors, e.g. due to memory decay. Given the possibility of bit errors, cryptographic keys may not conform exactly to the expected values. Hence, the secret key cannot be obtained immediately and reconstruction techniques are required to compensate the bit errors and get the secret key. The key reconstruction is based on traversing a large, highly unbalanced and dynamically growing search tree in order to find a feasible or optimal solution. The algorithmic idea forms a perfect example to study the branch-and-bound design flow for reconfigurable hardware. Nonetheless, it is a very unusual and insufficiently understood problem for FPGAs, because the computation and algorithmic structure is very irregular.

The approach and results discussed in this and the following chapters are published and presented in a peer-reviewed conference article [2] in the International Symposium on Field-Programmable Custom Computing Machines (FCCM), the premier A-ranked conference on capabilities of FPGAs and other reconfigurable hardware. Furthermore, an in-depth version has been peer-reviewed and published as a journal article [1] in the ACM Transactions on Reconfigurable Technology and Systems (TRETS), a highly relevant journal focusing on research with reconfigurable systems.

The remainder of this chapter is structured as follows: In Section 5.1, we define the basic terms and traversal methods for a tree data structure. Then we give a description of the branch-and-bound algorithmic pattern in Section 5.2. We explain and highlight the building blocks and variants of B&B that are important for a general realization and specialties for hardware realizations. In Section 5.3, we introduce the AES key reconstruction as the last and most difficult algorithmic challenge of our case study. On the basis of the AES key reconstruction we develop a stepwise process in Section 5.4, revealing how a B&B algorithm can be transformed and translated into a form suitable for an efficient implementation on an FPGA. We present the application-specific details of the AES key reconstruction, but also lift the concepts onto a more general level towards common B&B problems. Then we evaluate our hardware design and compare the results to a software reference in Section 5.5. Finally, we conclude this chapter in Section 5.6.

## 5.1 Basics and Common Terminology

In this section, we give a basic introduction to the common terminology used in the rest of this thesis. Afterwards, the branch-and-bound algorithmic pattern and its main building blocks are described.

### 5.1.1 Tree Data Structure

A tree is a well-known and widely used data structure to organize hierarchical data or to model optimization problems [65]. A search tree [67] is a special, typically very large subtype used for locating values or optimal solutions. The leaves of a search tree represent potential solutions, while nodes represent intermediate steps towards these solutions using edges that connect nodes. For many problems, the size of the search space is tremendous, resulting in very wide and deep tree structures with exponential growth.

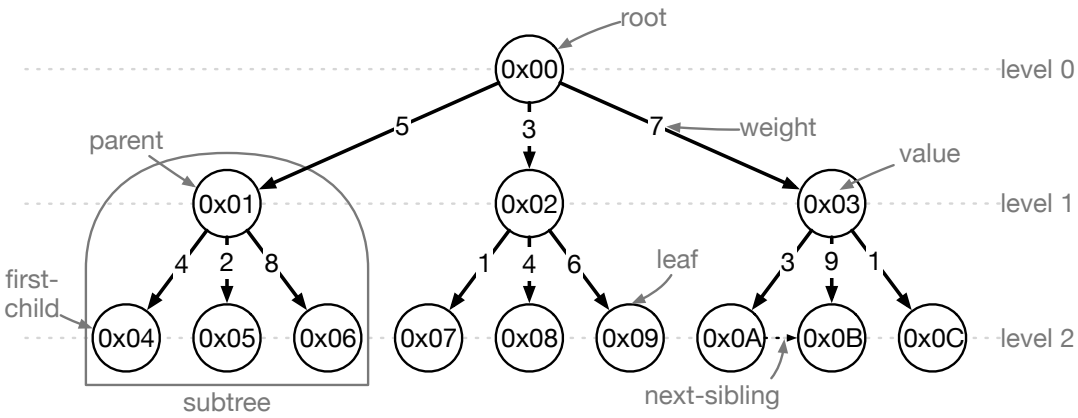



Figure 5.1: Visualization of a 3-ary search tree $T_3$ of depth $d = 3$.

***k*-ary Search Tree** A $k$-ary search tree $T_k$ [239] is organized, as the name suggests, in a tree where each node has a ***branching factor*** or degree of maximally $k$ children or is a leaf node without children. Figure 5.1 exemplary shows a 3-ary tree $T_3$ and labels the most important notations. Each node is associated with a value and each edge has a ***weight***, reflecting a cost or reward using the edge or playing a certain move. In general, a full $k$-ary search tree $T_k$ with ***depth*** $d$ (number of levels) has a total number of nodes of

$$|T_k| = k^0 + k^1 + k^2 + \cdots + k^d = \sum_{i=0}^{d} k^i = \frac{k^{d+1} - 1}{k - 1}, \tag{5.1}$$

indicating the exponential increase of nodes on each level. To give an intuitive interpretation, Chess has an average branching factor $k$ of about 35-38 [246], which means that there are 35-38 legal moves on average at each turn. With an average game length of about 40 turns per player, the full search space is already massive. In perspective, the game Go has an average branching factor $k$ of about 250 and the average game length is doubled compared to Chess [17].

In addition to a value, each node contains edges at least to its first-child, next-sibling and parent node to be traversable. If an edge is missing, it is set to `NULL`: e.g. the root node of a tree has no parent. In practice, typically more edges are introduced to make the tree traversal more efficient using shortcuts, e.g. the parent can have edges to all his direct children as depicted in Figure 5.1.

### 5.1.2 Traversal Strategies: Tree Structure and Search Path

There are different strategies to traverse a search tree towards a seeked or optimal value. Each traversal starts at the root node and proceeds towards a leaf node, building a sequence of the used nodes called a ***search path***. The most important strategies for tree traversals are the breadth-first search (BFS), depth-first search (DFS) and best-first search (BeFS) [97].

**Breadth-First Search (BFS)** BFS first explores all siblings on the present tree level $l$ before moving to all nodes on a higher level$(l+1)$. The traversal strategy is typically implemented as a queue data structure using the first-in-first-out (FIFO) principle to manage current search state. However, as the number of nodes on each level increases exponentially by a factor of $k$, this strategy may become quickly infeasible for larger search trees due to memory limitations of real systems.

**Depth-First Search (DFS)** In contrast, DFS builds a path that only explores one node per level and tries to traverse as fast as possible towards a leaf node where feasible solutions are located. If the leaf node is not the seeked or optimal value, backtracking to lower levels is used to continue the search. DFS algorithms are typically recursively specified, because the single path of interest is built in an incremental way. Therefore, the total memory requirements can be bound by the depth of the search tree (depth $d = log_k(|T_k|)$ for a $k$-ary tree, see Equation 5.1) multiplied by the branching factor $k$. Practical implementations use a stack data structure following the last-in-first-out (LIFO) principle to manage the search state.

**Best-First Search (BeFS)** BeFS is considered an informed search, because it explores the most promising nodes first according to a specific rule, e.g. minimizing or maximizing the weights of the edges used on the path. The actual level of a node is of minor importance. The traversal strategy is typically implemented as a priority queue, serving elements with lowest/highest weights first.

The strategy for traversing a search tree reflects a trade-off between maximizing the knowledge about the search space while staying within the memory bounds of a real computer system. In practice, strategies are also combined: e.g. starting the search with a depth-first search (DFS) and switching to a breadth-first search (BFS) on nodes close to leaf nodes. For the example 3-ary tree in Figure 5.1, a possible sequence in which the nodes are explored by each strategy is the following:

$$\text{BFS} = (\texttt{0x00}, \texttt{0x01}, \texttt{0x02}, \texttt{0x03}, \texttt{0x04}, \texttt{0x05}, \texttt{0x06}, \texttt{0x07}, \texttt{0x08}, \texttt{0x09}, \texttt{0x0A}, \texttt{0x0B}, \texttt{0x0C})$$
$$\text{DFS} = (\texttt{0x00}, \texttt{0x01}, \texttt{0x04}, \texttt{0x05}, \texttt{0x06}, \texttt{0x02}, \texttt{0x07}, \texttt{0x08}, \texttt{0x09}, \texttt{0x03}, \texttt{0x0A}, \texttt{0x0B}, \texttt{0x0C})$$
$$\text{BeFS} = (\texttt{0x00}, \texttt{0x02}, \texttt{0x07}, \texttt{0x08}, \texttt{0x01}, \texttt{0x05}, \texttt{0x04}, \texttt{0x09}, \texttt{0x03}, \texttt{0x0C}, \texttt{0x0A}, \texttt{0x06}, \texttt{0x0B})$$

Besides the search path, which decides the order in which branches are taken, the second most important property of trees for this thesis is the ***tree structure***. The tree structure defines the actual hierarchy of a tree, beginning with the root node and followed by its children. Figure 5.1 shows one concrete tree structure, starting at the node `0x00`. Accordingly, all paths will start at this node, independently of the traversal strategy. By defining another node as the root node, the search tree and the possible path to traverse it would be completely different. By analyzing an actual search problem instance, an optimized tree structure can be constructed to accelerate the search procedure.

However, the exponential increase in the total number of nodes leads to a combinatorial explosion, making a plain brute-force search impractical with any of the presented strategies and therefore quickly infeasible for larger search trees due to memory and time limitations of real systems. The only way to face the combinatorial explosion is to reduce the search space at each tree level by discarding unpromising subtrees that have a very low probability to lead to the seeked value or an optimal solution. In this thesis we focus on three techniques to improve the processing of large search trees: The first one, *branch-and-bound*, is the main focus of this chapter and will be explained next. The second, *work stealing* is used for dynamic distribution of the work to several workers and will be the focus of Chapter 6; finally we utilize *instance-specific information* to improve the performance for very hard problem instances in Chapter 7.

## 5.2 Branch-and-Bound: General Idea

The branch-and-bound (B&B) search is the most commonly used algorithmic pattern to solve combinatorial optimization or planning problems [169]. Branch-and-bound appears in a variety of $\mathcal{NP}$-hard real-world problem domains, such as logistics [265, 44], scheduling [137, 80], combinatorics [43, 212], decision processes [61], planning [124, 125], and many others (see related work in Chapter 8).

Formally, the tackled problems can be defined as minimization or maximization problems[1] of an ***objective function*** $obj(x)$, where $x = (x_0, x_1, \ldots, x_{d-1})$ are $d$ variable constraints that need to be found. Intuitively, each $(x_i)_{i=0,\ldots,d-1}$ corresponds to the node on the $i$-th level in the search tree that is reached through a specific, unique search path from the root node using the nodes $(x_0, x_1, \ldots, x_{i-1})$. In the case of a $k$-ary tree (see Section 5.1.1), each $(x_i)_{i=0,\ldots,d-1}$ has $k$ possible branches and $d$ corresponds to the depth of the search tree. For $\mathcal{NP}$-hard branch-and-bound problems, an exponential number of potential solutions are located on leaves of the search tree, making a complete, brute-force search impractical.

The main idea behind the B&B pattern is to avoid an explicit enumeration through all possible paths in the search tree by discarding unpromising subproblems early on. This idea is illustrated in Figure 5.2. In (a), the total search space $S$ including the set of all feasible solutions is represented by the root node of the corresponding 3-ary search tree. In (b), the set of feasible solutions derived from the root node is partitioned into smaller, usually disjoint subproblems $S_0, S_1$ and $S_2$ through imposition of a new constraint for the first variable $x_0$. These smaller subproblems $S_0, S_1$ and

[1] To keep the following description compact and without loss of generality, we focus only on minimization problems.

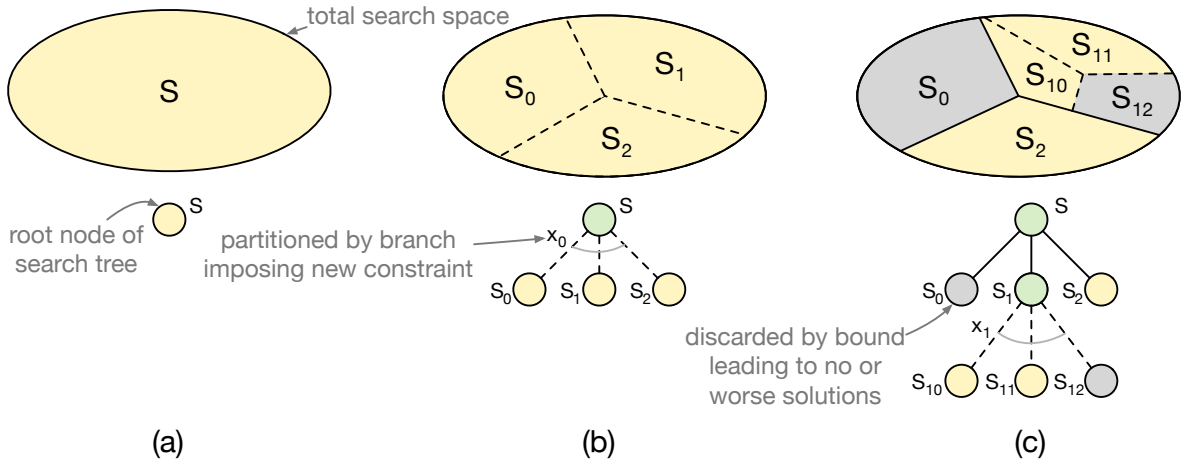



Figure 5.2: Main idea of the branch-and-bound principle.

$S_2$ satisfy the same constraints as $S$ and are additionally constrained by the value of $x_0$ in each branch. Then each subproblem can be evaluated separately and systematically with the same principle of partitioning into smaller subproblems with additional constraints for their descendants. The status of the search is described by a pool of live or unexplored nodes (colored in yellow in Figure 5.2) and optionally an intermediate best solution, if reached. Depending on the tree traversal strategy (see Section 5.1.2) the next node to expand is selected from the pool. In (c), the subproblem $S_1$ is selected and then further partitioned into the subproblems $S_{10}$, $S_{11}$ and $S_{12}$ by imposing a new constraint for the second variable $x_1$.

The search process is continued until the seeked value or a provable optimal solution is found, or the search process contains no live nodes in the pool. The termination and convergence towards a feasible solution can be guaranteed if the size of each subproblem is getting smaller by adding more constraints and the number of feasible solutions in the search space is limited.

The final key element of the B&B approach is the ***bounding function*** $bnd(y)$ that computes for any node $y \in S$ a value representing a lower or upper bound of a potential solution including $y$. With the help of the bounding function $bnd(y)$ and an intermediate best solution found so far, large parts of the total search space can be directly discarded [50, 206, 168, 28]. In Figure 5.2 (c) the subproblems $S_0$ and $S_{12}$ (in gray) are discarded by the bounding function, while the root node $S$ and the inner node $S_1$ (in green) are compatible to the bounding function and added to the pool of live nodes.

### 5.2.1 Algorithmic Pattern

Listing 5.1 shows the described branch-and-bound algorithmic pattern in pseudocode. Before the search process begins, an initial feasible solution can be optionally computed by the function `COMPUTE_INITIAL_BOUND()` with the help of a heuristic to serve as an intermediate bound (line 2). In line 3, the pool of live, yet unexplored nodes is populated with the root node, where the search always starts.

```
1  # Initialization.
2  bound ← COMPUTE_INITIAL_BOUND( root )
3  pool ← { root }
4  solution ← NULL
5
6  # Main loop.
7  while pool ≠ ∅ :
8    # Select node to process from pool.
9    cur_node ← COMPUTE_BRANCH( pool )
10   pool_remove( cur_node  )
11
12   # Compute additional information inferred by selected node.
13   INFER_KNOWLEDGE( cur_node )
14
15   # Compute bound inferred by selected node.
16   cur_bound ← COMPUTE_BOUND( cur_node )
17
18   # Validate, if current bound is valid.
19   if check_bound(cur_bound, bound) == VALID :
20     # Check, if the seeked value or better solution is found.
21     # Update bound and store solution accordingly.
22     if solution_found(cur_node, cur_bound, bound, solution) :
23       continue
24
25     # Otherwise, generate next nodes to process and add to pool.
26     cur_node_children ← GENERATE_LIVE_NODES( cur_node )
27     pool_add( cur_node_children )
28   else:
29     # Node and subtree cannot lead to a better solution.
30     discard( cur_node )
```

Listing 5.1: General algorithm for (lazy) branch-and-bound with the five essential operations highlighted.

The main loop in line 7 iterates over the total search space as long as the pool of unexplored nodes is not empty. In each iteration step, the next node to process (`cur_node`) is selected from the pool by the function `COMPUTE_BRANCH` and the selected node is removed from the pool (lines 9–10). Then the additional constraints inferred by the selected node are computed in line 13 by the `INFER_KNOWLEDGE` function. Afterwards, the bounding function is computed for `cur_node` by the `COMPUTE_BOUND` function. If the computed bound exceeds the initial bound or the current best solution, the selected node and all descendants are discarded from the search, because they cannot lead to a feasible solution (lines 28–30). In the previous example in Figure 5.2 (c), the subproblems $S_0$ and $S_{12}$ colored in gray are discarded by the bounding function. If, on the other hand, the computed bound value is valid, the algorithm checks if a better solution or the seeked value is found. If this is the case, the bound is updated, the found solution is stored and the search continues (lines 22–23). Otherwise, the `GENERATE_LIVE_NODES` operation in lines 26–27 generates the next nodes to process and adds them to the pool of live nodes. The partitioning of the search space into multiple subproblems is represented by the branches in the search tree (see Figure 5.2 (b) with branches to $S_0$, $S_1$ and $S_2$).

The presented variant in Listing 5.1 is named lazy branch-and-bound, because the generation of the next nodes to process (line 26) is performed after the bound operation [65]. This variant is used when the next node to be processed should be of

the highest depth in the search tree (following the DFS principle, see Section 5.1.2). In contrast, the eager branch-and-bound variant calculates the bounds as soon as possible without selecting a specific node to process. Choosing between the variants creates an interesting trade-off between the number and type of live nodes in the pool. The nodes are either raw (without a computed bound, leaving additional computational efforts) or already bounded and ready for further expansion. Selecting the right strategy is especially important for the parallelization of branch-and-bound to generate a good granularity of work items to distribute, which is the focus of Chapter 6.

Depending on the problem domain and instance, one can implement numerous custom variants for the five essential operations highlighted in Listing 5.1 to construct the most promising algorithmic search pattern dynamically with lazy or eager variants. These essential operations are discussed briefly with more details in the next paragraphs and afterwards applied to the case study accompanying this thesis.

1. `COMPUTE_INITIAL_BOUND( root ):` **Compute Initial Bound** An intermediate feasible solution is computed to receive an initial bound in order to discard unpromising subproblems as early as possible and reduce the search space. Besides very problem-specific heuristics, general approximation techniques [33] such as simulated annealing [273], tabu search [109] or genetic algorithms [77] are typically used. If no heuristic exists, the initial bound is set to $+\infty$ or $-\infty$.

2. `COMPUTE_BRANCH( pool ):` **Select Branch** The selection of the next node to process from the pool of live nodes follows the tree traversal strategies discussed in Section 5.1.2. The depth-first search (DFS) variant is useful if no initial bound is available, because with DFS the search process reaches leaf nodes as fast as possible and therefore produces a feasible solution and an initial bound. Otherwise, a combination of the search strategies is useful to reflect the trade-off between the number of live nodes in the pool and the memory capacities of the actual system. The selection of a branch comes with the addition of constraints, often in form of assigning values to variables.

3. `INFER_KNOWLEDGE( cur_node ):` **Compute Further Information** Based on the constraints added by selecting a particular branch in the previous step, additional information can be inferred.

4. `COMPUTE_BOUND( cur_node ):` **Discard Infeasible Subproblems** In order to avoid the combinatorial explosion during the search process, the bounding function $bnd($`cur_node`$)$ is the key component. The function tries to bound each selected node as close as possible to the optimal value (called a tight bound). However, computing the correct or very tight value is usually $\mathcal{NP}$-hard itself and therefore $bnd($`cur_node`$)$ is usually also a heuristic and obliges the trade-off between quality (tight bound) and time (efficient estimation). The bound operation is highly application-specific and could be an upper or lower threshold. To be useful in practice, the bounding function typically has the following properties:

- If $\texttt{cur\_node}_p$ is the parent node of `cur_node`, then the bounding function is monotonic: $bnd(\texttt{cur\_node}_p) \leq bnd(\texttt{cur\_node})$. The bound value increases (or does not get smaller) by adding more information through additional constraints. The bound value of the parent holds as a lower bound for the value of *any* solution implied by the descendants of the node. For maximization problems, the inequality symbols are inverted.
- For all leaves, the potential solution derived from the bounding function leads into a feasible solution of the original problem: $bnd(x_d) = obj(x_d)$, where $x$ is a leaf node and $d$ the depth of the search tree.

(5) `GENERATE_LIVE_NODES( cur_node ):` **Subdivide the Search Space** If a solution is not found yet, the next live nodes to process are generated and added to the pool. Depending on the tree traversal strategies the nodes can be on the same level as the current node, on the next higher level towards a leaf node or on any other level that is promising.

### 5.2.2 State Machine Design for Reconfigurable Hardware

The first challenge is to translate the branch-and-bound (B&B) algorithmic pattern with the five essential operations described above into a hardware design. As B&B algorithms always consist of the same general elements, we are able to develop a general finite state machine (FSM, see Figure 5.3) for the main loop that represents each operation in an FSM state.

The first operation `COMPUTE_INITIAL_BOUND` (1) is optional and only executed at most once to compute the initial bound by a heuristic and therefore located on the host CPU, while the other, computationally intensive operations (2) - (5) are located in reconfigurable hardware.

The superstate `PROCESS` corresponds approximately to the `while` loop in Listing 5.1. It contains several substates: The state `COMPUTE_BRANCH` (2) is responsible for selecting the most promising live node from all possibilities in the pool. Afterwards the further knowledge is inferred by the `INFER_KNOWLEDGE` (3) state. Next, the state `COMPUTE_BOUND` (4) is responsible for pruning the search space, going back to the `COMPUTE_BRANCH` (2) state if the currently considered branch with the corresponding bound cannot lead to a feasible solution. If `COMPUTE_BOUND` (4) assesses the currently considered branch as valid, but not a feasible solution yet, the state `GENERATE_LIVE_NODES` (5) is entered. This state generates the next live nodes to process and adds them together with the current search context into the data structure of the pool. The states responsible for operations on the pool data structure are condensed under `CHECKPOINTING`. The checkpointing mechanism keeps track of the current state of the search. A checkpoint added by `POOL_ADD` needs to contain all information that is required by the `COMPUTE_BRANCH` (2) state to fully define the state of the FSM, i.e. the used search path to the selected node of the search tree and the corresponding bound.

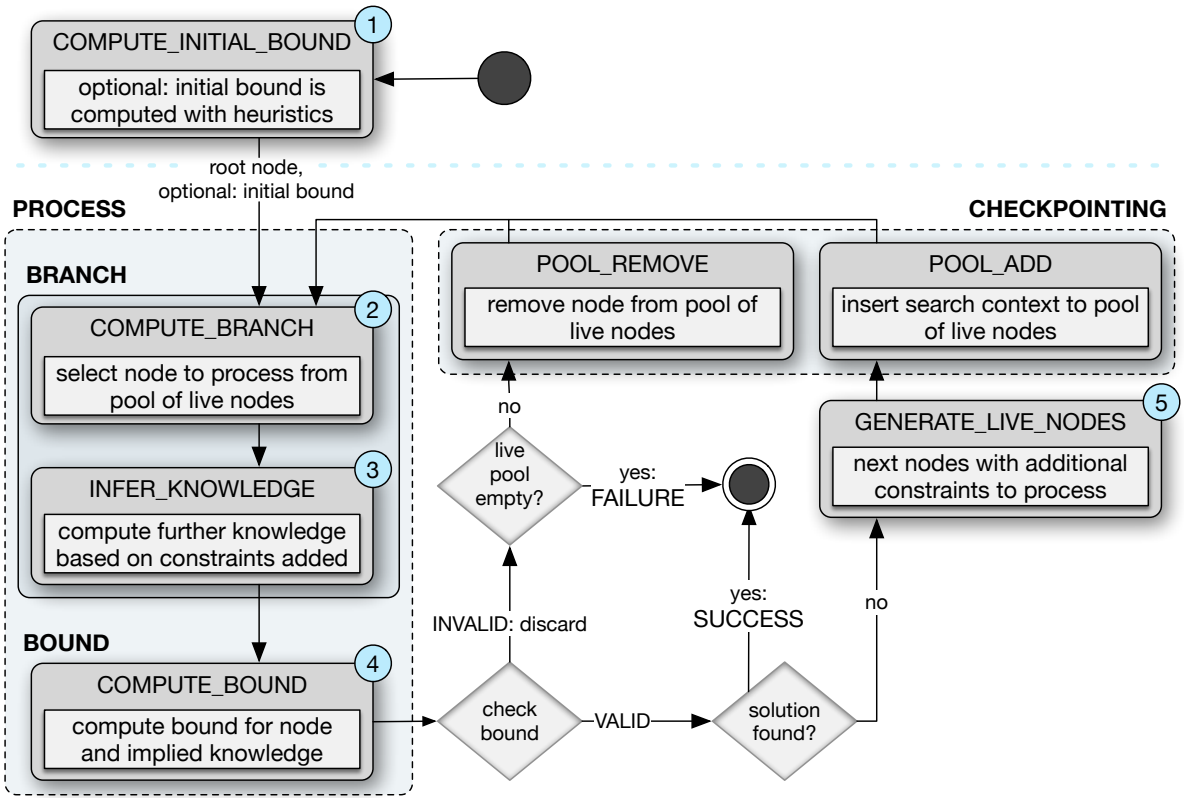



Figure 5.3: General elements of a FSM that implements the main loop of the B&B design paradigm (see Listing 5.1).

## 5.3 Case Study: Secret Key Reconstruction

The secret key reconstruction provides a perfect example to study the aforementioned concepts of the branch-and-bound algorithmic pattern in reconfigurable hardware on a concrete application and to generalize the lessons learned to other problems. Reconstructing corrupted keys is the last necessary step in a side-channel attack. The main idea of secret key reconstruction was proposed by Halderman et. al. [126]. In this thesis, we use the improved variant proposed by Tsow [269] for AES key schedules, because it is algorithmically closer to the branch-and-bound algorithmic pattern and offers more optimization opportunities. Although the following concepts can be applied for all AES key sizes with minor adaptations, we will describe them for AES-128 for simplicity and comprehensibility. We will describe the software approach proposed by Tsow and explicitly point out and refer to the general branch-and-bound algorithmic pattern described in the previous Section 5.2.

### 5.3.1 Basic Idea

The reconstruction algorithm receives an AES key schedule identified by the methods presented in Chapter 4. From the key schedule, the secret key cannot directly be obtained due to bit errors caused by the side-channel attack, e.g. memory decay. The main algorithmic idea is to exploit the bijective AES key expansion function (see Section 3.3.2) that is used to compute the round keys derived from the master key. The expansion function provides important redundancies that can be used as a bounding function for the branch-and-bound algorithmic pattern.

Formally, the tackled reconstruction problem can be defined as a minimization problem of the objective function $obj(x)$, where $x = (g_0, g \in \{0, 1, \ldots, 15\})$ are 16 variable constraints of specific key schedule values that need to be found. Table 5.1 shows one possible allocation of the 16 byte positions (emphasized in bold) presented by Tsow [269]. Values for all remaining byte positions $g_i$ for $i > 0$ can be derived from already fixed bytes following *implication chains* (described in next section). The 16 byte positions for $g_0$ are chosen such that they, together with the implications, determine a complete key schedule.

| w / b / r | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | $\mathbf{0_0}$ | $14_{10}$ | | | | $13_{10}$ | | | | $12_{10}$ | | | | $1_1$ | $14_9$ | |
| 1 | $\mathbf{1_0}$ | $13_9$ | | | | $12_9$ | | | | $2_2$ | $14_8$ | | | $2_1$ | $13_8$ | |
| 2 | $\mathbf{2_0}$ | $12_8$ | | | | $3_3$ | $14_7$ | | | $3_2$ | $13_7$ | | | $3_1$ | $12_7$ | |
| 3 | $\mathbf{3_0}$ | $4_4$ | $14_6$ | | | $4_3$ | $13_6$ | | | $4_2$ | $12_6$ | | | $4_1$ | $5_5$ | $14_5$ |
| 4 | $\mathbf{4_0}$ | $5_4$ | $13_5$ | | | $5_3$ | $12_5$ | | | $5_2$ | $6_6$ | $14_4$ | | $5_1$ | $6_5$ | $13_4$ |
| 5 | $\mathbf{5_0}$ | $6_4$ | $12_4$ | | | $6_3$ | $7_7$ | $14_3$ | | $6_2$ | $7_6$ | $13_3$ | | $6_1$ | $7_5$ | $12_3$ |
| 6 | $\mathbf{6_0}$ | $7_4$ | $8_8$ | $14_2$ | | $7_3$ | $8_7$ | $13_2$ | | $7_2$ | $8_6$ | $12_2$ | $14_1$ | $7_1$ | $8_5$ | $9_9$ |
| 7 | $\mathbf{7_0}$ | $8_4$ | $9_8$ | $13_1$ | | $8_3$ | $9_7$ | $12_1$ | $\mathbf{14_0}$ | $8_2$ | $9_6$ | $10_{10}$ | $\mathbf{13_0}$ | $8_1$ | $9_5$ | $10_9$ |
| 8 | $\mathbf{8_0}$ | $9_4$ | $10_8$ | $\mathbf{12_0}$ | $15_{10}$ | $9_3$ | $10_7$ | $11_{10}$ | $15_9$ | $9_2$ | $10_6$ | $11_9$ | $15_8$ | $9_1$ | $10_5$ | $11_8$ |
| 9 | $\mathbf{9_0}$ | $10_4$ | $11_7$ | $15_7$ | | $10_3$ | $11_6$ | $15_6$ | | $10_2$ | $11_5$ | $15_5$ | | $10_1$ | $11_4$ | $15_4$ |
| 10 | $\mathbf{10_0}$ | $11_3$ | $15_3$ | | | $11_2$ | $15_2$ | | | $11_1$ | $15_1$ | | | $\mathbf{11_0}$ | $\mathbf{15_0}$ | |

w: word b: byte r: round

Table 5.1: One possible allocation for the position of the 16 byte values $g_0$, $g \in \{0, 1, \ldots, 15\}$ (emphasized in bold) for AES-128. The remaining values $g_i$ for $i > 0$ are derived from implication chains to complete round 8.

To achieve this, the reconstruction algorithm proposed by Tsow [269] resembles a depth-first search (DFS) and traverses the search space as shown in the left part of Figure 5.4. The search space is a 256-ary search tree and has a fixed maximum depth of $d = 16$ levels. The search space is partitioned into 256 possible branches in each level $g \in \{0, 1, \ldots, 15\}$ of the tree by sequentially assigning (or *guessing*) a value for $g_0$ from all 256 possible values of a byte (`0x00`, `0x01`, ..., `0xFF`) that is compatible with a bounding function.

Reusing the terminology defined in Section 5.1.2, we want to emphasize two important connections: different positions for $g_0$ in the key schedule result in different *tree structures*, while the guessing order for the actual values for each $g_0$ results in different *search paths*. Accordingly, Table 5.1 shows one possible allocation for all positions of $g_0$, $g \in \{0, 1, \ldots, 15\}$, while the sequential guessing of values for each $g_0$ from all 256 possible values of a byte (`0x00`, `0x01`, ..., `0xFF`) is one possible search path. We call a search using the allocation in Table 5.1 and incremental guessing a **static tree structure** and **static incremental search path**. In this chapter, only the static variants are used in software and hardware. In the following chapters we will analyze concrete problem instances to further improve the search by building dynamic tree structures and dynamic search paths.

Each guessed value $g_0$, $g \in \{0, 1, \ldots, 15\}$ corresponds to a specific position in the key schedule and guessing a value corresponds to imposing a new constraint in the search tree. If a valid value for $g_0$ is found that is compatible to a bounding function, the search proceeds on the next search tree level, sequentially guessing a value for $(g + 1)_0$. The compatible guessed bytes for the example in Figure 5.4 are colored in green. For the first byte position $0_0$ in Figure 5.4, the possible values are tried

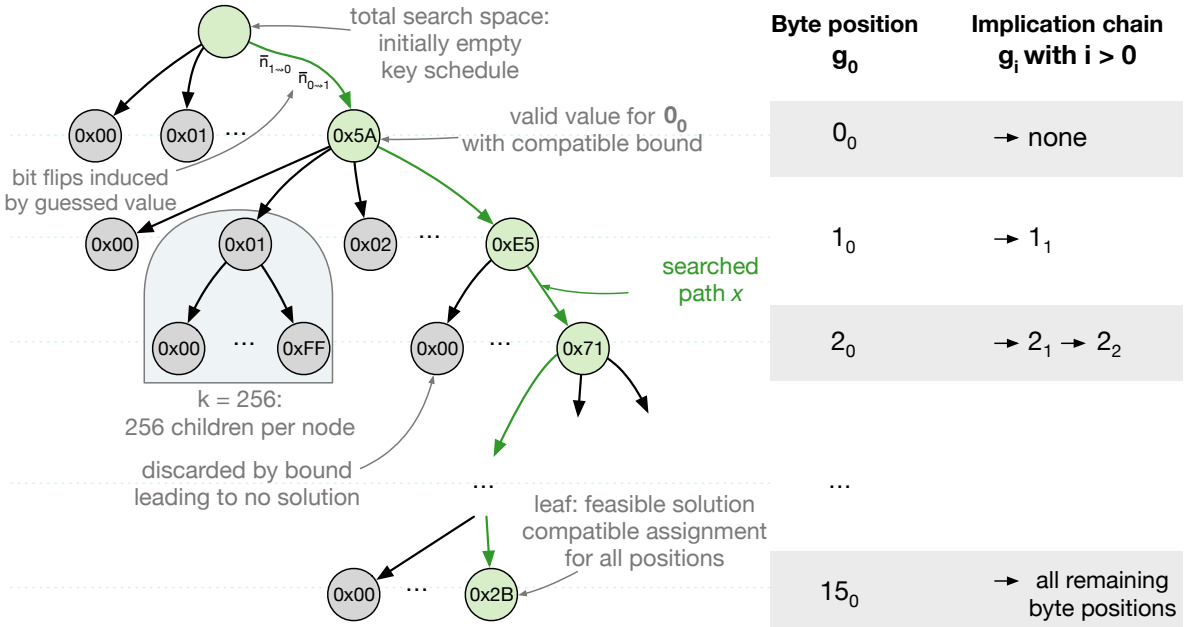



Figure 5.4: Search tree for the key reconstruction, starting at the root node and sequentially *guessing* compatible byte values in each level.

consecutively, starting with `0x00` and continued until the compatible value `0x5A` is guessed. Then the search proceeds with the next byte position $1_0$.

**Implication Chains** All guessed values for byte positions $g_0, g \in \{0, 1, \ldots, 15\}$ are progressively combined using the inherent structure of a key schedule (the key expansion function of AES depicted in Figures 3.4 in Section 3.3.1, page 21) to deduce the values of other byte positions. Those *implied* byte positions, denoted as $g_i$ with $g \in \{0, 1, \ldots, 15\}$ and $i > 0$ in Table 5.1, form the implication chains. As shown in the right part of Figure 5.4, the guessed value for the first position $0_0$ is only constrained by the corresponding decayed byte value at the same position. While the guessed value for the second position $1_0$ is constrained by the decayed byte at the same position and another implied value $1_1$. The value for the byte position $1_1$ can be implied using the complex key expansion rule (see Figure 3.4) after the values for byte positions $0_0$ and $1_0$ are guessed. After additionally guessing the value for the third byte position $2_0$, implication of the byte position $2_1$ (combining $2_0$ and $1_0$) and subsequently $2_2$ (combining $2_1$ and $1_1$) is possible. Note that for some implications also the inverse variants of the expansion rules are required, resulting in a backward calculation inside the key schedule. The number of possible implications depends on the byte position $g_0$ and increases with each byte:

$$\texttt{number\_of\_implications\_for\_position(}\ g_0\ \texttt{)} = \begin{cases} g & \text{if } g < 11 \\ 10 & \text{if } 11 \leq g < 15 \\ 65 & \text{if } g = 15 \end{cases} \quad (5.2)$$

**Recover Secret Key from Round 8** After all implications of the last byte position $15_0$ are computed, the entire data for round 8 (highlighted in Table 5.1) of the key schedule is known. From a single complete round key, the entire key schedule and thus the actual secret key located in round 0 can be recovered. At this point 111

of the 176 bytes of the key schedule for AES-128 are determined. Table 5.2 shows an optimal and maximally parallel ordering to compute the missing 55 bytes. The idea to find such an ordering is to compute the missing values in parallel upwards of round 8 and downwards of round 8. The empty positions represent the already determined values. The other byte positions are labeled with letters from `A` till `J`. The alphabetic order defines the sequence of computations. Positions with identical letters are independent and can be computed in parallel. The letters highlighted in bold (word 0) use the complex key expansion rule, which requires a memory access. This constraint needs to be considered in the further study.

| w | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b / r | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | | | **J** | **I** | H | | J | I | H | | J | I | H | | | I |
| 1 | | | **I** | **H** | G | | I | H | G | | | H | G | | | H |
| 2 | | | **H** | **G** | F | | | G | F | | | G | F | | | G |
| 3 | | | | **F** | E | | | F | E | | | F | E | | | |
| 4 | | | | **E** | D | | | E | D | | | | D | | | |
| 5 | | | | **D** | C | | | | C | | | | C | | | |
| 6 | | | | | B | | | | B | | | | | | | |
| 7 | | | | | A | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | | |
| 9 | | | | | A | | | | B | | | | D | | | |
| 10 | | | | **E** | B | | | F | D | | | G | E | | | H |

Table 5.2: Optimal sequence to complete missing values for the static allocation of Table 5.1.

### 5.3.2 Software Approach

Listing 5.2 outlines the main function of the algorithm in pseudocode with the five essential branch-and-bound operations highlighted. For the decayed key schedule `ks_Decayed` in line 3 that was obtained from memory and contains erroneous bits, a candidate schedule `ks_Candidate` is built up recursively calling the `recoverKeyRec` function. In every recursive call, a value for the next byte $g_0$ is computed in line 14 by `getNextGuessedByte` and all bytes that can be derived from the guessed byte are computed in line 17 by `computeImpliedBytes`. If the extended candidate key schedule still passes the bounding function `isCompatible` in line 20, the next recursive call descends further into the search tree. Otherwise, or if no feasible key schedule is found further down the tree, the next possible value for $g_0$ is tried (loop in line 12). Computing the implied bytes and checking the compatibility consume the most runtime in software and thus are crucial in order to gain performance [207]. The positions of the guessed bytes in the key schedule are chosen according to the static allocation in Table 5.1. After 16 bytes are guessed and checked for compatibility along with all implied bytes, the candidate schedule `ks_Candidate` corresponds to a feasible solution and can be returned as a valid AES key schedule in line 9. If the master key in the first round key of the found solution does not decrypt the secret data, the search can be continued to find the next feasible solution.

This idea of guessing and implying byte values to sequentially complete the key schedule forms a 256-ary search tree, with exactly the properties introduced at the beginning of this chapter. The algorithm is computation-bound. When guessing a single value for one of these byte positions $0_0, 1_0, \ldots, 15_0$, all $k = 256$ possibilities are tested in the worst case. The large branching factor on each level makes it necessary to prune parts of the search tree to avoid a combinatorial explosion and to allow

finding a solution in an acceptable amount of time. We use an error model as the bounding function $bnd(x)$ to prune the search tree, eliminating or bounding guesses for bytes (or subtrees of the search space) that cannot lead to a feasible AES key schedule. The details of the bounding function will be discussed in the next section.

```
1  # Key schedule obtained from memory
2  #   contains erroneous bits caused by memory decay.
3  KeySchedule ks_Decayed
4  # Optional: COMPUTE_INITIAL_BOUND
5
6  recoverKeyRec( KeySchedule ks_Candidate ):
7    # After 16 guesses a feasible solution is found.
8    if ( ks_Candidate.getNumberOfGuessedBytes() == 16 ):
9      return ks_Candidate.getKey();
10
11   # Impose next constraint.
12   for ( i = 0 to 255 ):
13     # Select next branch to process: COMPUTE_BRANCH
14     g_0 = ks_Candidate.getNextGuessedByte( i )
15
16     # Set guessed byte and compute implication chain: INFER_KNOWLEDGE
17     ks_Candidate.computeImpliedBytes( g_0 );
18
19     # Check bounding function: COMPUTE_BOUND
20     if( ks_Decayed.isCompatible( ks_Candidate ) ):
21       # If bound holds, continue down the tree: GENERATE_LIVE_NODES
22       key = recoverKeyRec( ks_Candidate )
23       if ( key != NULL ):
24         return key;
25     else:
26       # Otherwise, discard selected node and whole subtree.
27   return NULL;
```

Listing 5.2: Recursive algorithm for key reconstruction of AES-128 keys with the five essential branch-and-bound operations highlighted.

### 5.3.3 Bounding the Search Space: Error Model

At a basic abstraction, the bounding function `isCompatible` in line 20 of Listing 5.2 reflects the probability that a guessed value (a *candidate* for the correct value) has decayed into the *observed* value (in the memory image). Therefore, the goal of the bounding function $bnd(x)$, together with the objective function $obj(x)$, is to minimize the difference in terms of number of bit errors between the guessed and the decayed key schedule. This increases the likelihood to discard unpromising candidates and to find a feasible solution in a reasonable time. For our case study, the bounding function that is applied by the `isCompatible` function is also called an error model.

The error model for our case study receives the two key schedules as input: the found, decayed key schedule `ks_Decayed` and a possible candidate key schedule `ks_Candidate`. The error model uses the defined rules from Section 3.2 to determine whether the two key schedules are compatible with each other or not. The compatibility in the error model under consideration determines whether all bytes of the candidate byte can decay at all to the found bytes from the decayed memory image. Figure 5.4 shows how different bit flips are induced by assigning different values for each each byte. In this way, invalid candidates can be immediately determined and

excluded. If a guess for a byte is compatible, the search proceeds in the next tree level. This reduces the search space considerably and only then can the problem be calculated in a practicable way without a combinatorial explosion.

Note that for better performance and in contrast to the original proposal by Tsow [269], in practice we only check the compatibility of the guessed and all implied values in each level. All remaining values are already checked and reflected by the bound values.

## 5.4 Branch-and-Bound in Hardware

In the beginning of this chapter, we introduced the B&B principle and emphasized that our case study can be described as a B&B problem. In this section, we bring both concepts together and describe the implementation of a B&B algorithm in hardware. We use the AES key reconstruction as our case study to show the relationship between the general branch-and-bound operations and the application-specific realization in reconfigurable hardware. More details on the AES-specific implementation can be found in the original papers and our previous publications on AES key reconstruction [2, 1].

### 5.4.1 Software Translation: Concrete Finite State Machine

The first challenge is to translate the recursive branch-and-bound algorithm from Listing 5.2 into a form suitable for reconfigurable hardware. Combining the general state machine design for branch-and-bound developed in Section 5.2.2 and having the application-specific branch-and-bound elements identified in Listing 5.2, we are able to create a concrete FSM that implements the recursive AES key reconstruction as depicted in Figure 5.5.

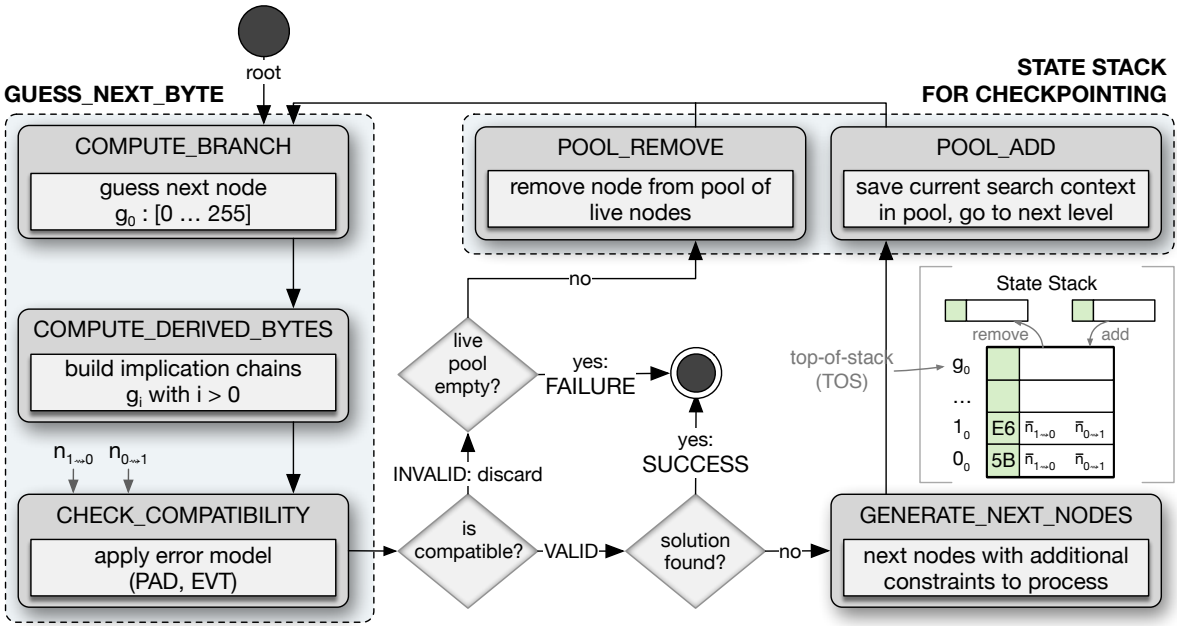



Figure 5.5: Concrete FSM that implements the recursive AES key reconstruction based on the branch-and-bound design paradigm.

### 5.4.2 Selecting Branches

The superstate `GUESS_NEXT_BYTE` of the FSM corresponds approximately to the `for` loop of the recursive algorithm that tries all possible values for the guessed byte $g_0 \in \{0, 1, \ldots, 255\}$ in sequential order. The superstate keeps track of the current level of the search tree and determines which byte is to be guessed next according to the search path. It contains substates to set the guessed byte (`COMPUTE_BRANCH`), to compute all implied bytes and to perform the compatibility check. The latter two consume the most runtime in software and thus are crucial in order to gain performance. The tasks inside those substates `COMPUTE_DERIVED_BYTES` and `CHECK_COMPATIBILITY` differ in each level not only by their inputs, but also in the type and number of implication steps to apply for the former state and in the number of new bytes to check for the latter level (see implication chains in Section 5.3.1).

### 5.4.3 Computing Inferred Knowledge: Implication Chains

For the reason that all levels differ, we decided to implement the mentioned two states of the FSM, `COMPUTE_DERIVED_BYTES` and `CHECK_COMPATIBILITY`, as separate subcircuit for each of the 16 different reconstruction levels, one for each level. This saves the latencies for selecting the inputs and enables optimization of their datapaths for their specific tasks. Additionally, we gathered statistics about how often each level is reached depending on the bit flip rate (see Figure 5.6) and found that levels 8 and higher are typically reached at least three orders of magnitude less frequently than the most frequent ones (levels 2 to 4). Therefore the more frequent levels are optimized with elaborate combinational datapaths, whereas multi-cycle implementations are chosen for levels 8 and higher in order to minimize their impact on the clock frequency.

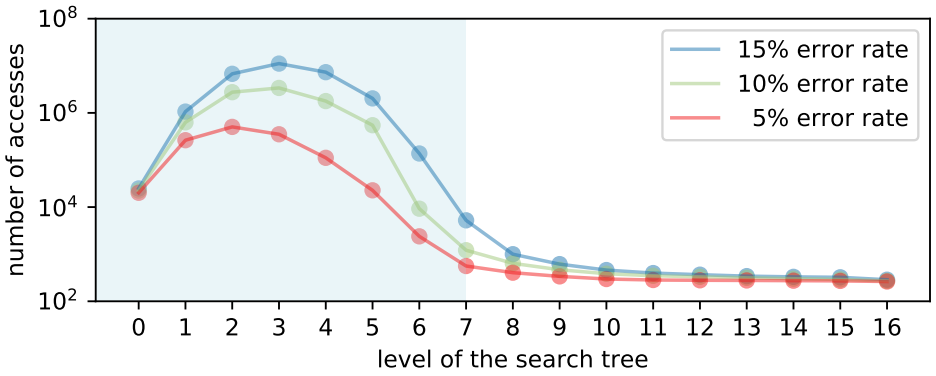



Figure 5.6: Number of times each level is reached for 256 test cases.

Computing the implied bytes follows the static allocation shown in Table 5.1. As motivated before, for each level we hardcoded all its implications into one subcircuits. This also allowed us to hardcode the round constants RCon (required in the complex key expansion rule) into the subcircuits to save accesses to BRAM, since in each level at most one known round constant is used. Note that all implied bytes of each level depend on each other, so for the example allocation in Table 5.1 after guessing $\mathbf{15_0}$, first $15_1$ needs to be derived from $\mathbf{15_0}$ and $11_4$, before $15_2$ can be derived from $15_1$ and $11_5$. All derivation operations that only apply the simple XOr operation are easily combined into a single-cycle combinational path, for example from $\mathbf{15_0}$ all the

way left to $15_3$. After hardcoding the round constants RCon for each level, the operations of the complex expansion rule can also be merged into this combinational path. However, this additionally requires us to store the data for the substitution operation SBox in distributed LUT RAM that can be accessed without a latency cycle. The result of an SBox-LUT operation can immediately be used for the adjacent computation in the same cycle and does not require a waiting cycle, in contrast to the BRAM accesses. With 256 bytes, the SBox table is slightly larger than a typical use case for LUT RAM, but easily tolerable for the short and frequent levels of the `COMPUTE_DERIVED_BYTES` state. With this optimization, we were able to implement single-cycle combinatorial datapaths for the most frequent levels 0 to 7. For the longer and conveniently less frequent levels 8 and higher, we instead employ BRAM for the SBox lookup, splitting the state into a chain of substates, each one ending with a read request SBox-Req and starting with the corresponding read in the next cycle, indicated by the clock signal between the sub-levels. In the upper part of Figure 5.7, we illustrate level 3 as an example for a single-cycle datapath, starting with the complex expansion rule to compute $3_1$ followed by simple XOrs to compute $3_2$ and $3_3$ in a single cycle. In the lower part of the same figure, level 15 is computed as an example for a multi-cycle datapath. In the first cycle, the implications $15_1$-$15_3$ are computed and the SBox-LUT read request is issued. Then, in the next cycle, the value is available and $15_4$ is computed as the first byte of the next substate, along with $15_5$ to $15_7$ which follow combinatorially. Afterwards, the substate ends with the next read request to compute $15_8$.

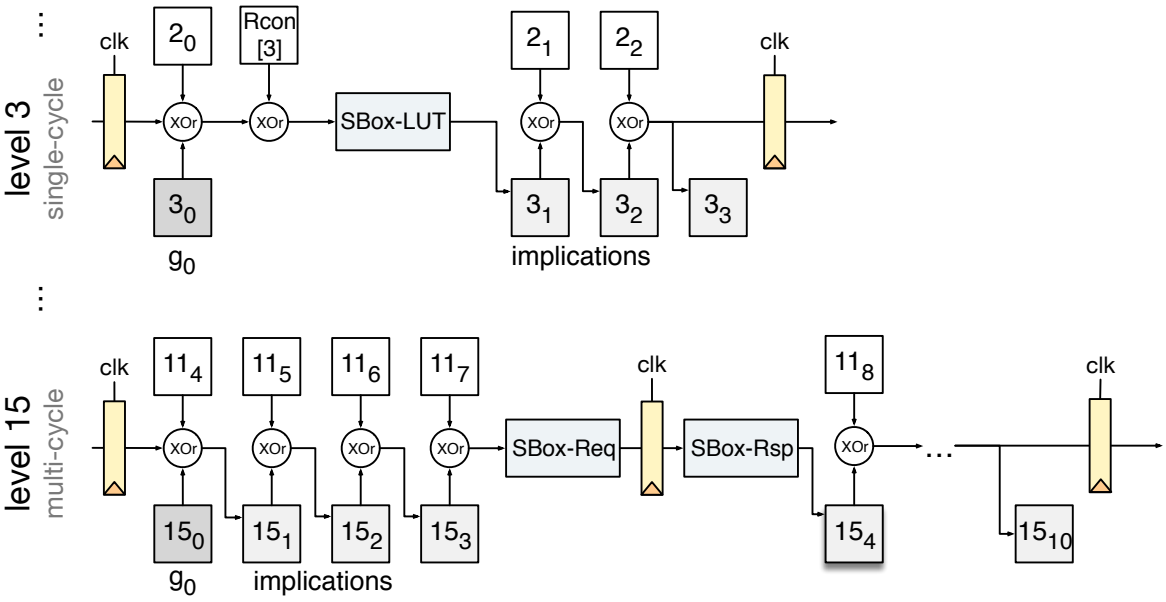



Figure 5.7: Illustration of datapaths for level 3 (single-cycle combinational path using SBox-LUT) and level 15 (multi-cycle combinational path using BRAM) of state `COMPUTE_DERIVED_BYTES`.

### 5.4.4 Checkpointing Tree Traversal

Since for our case study the recursion depth is limited to 16 levels, we are able to design a finite state machine where the current search tree node, which is implicitly represented by the call stack of the recursive function in software (see Listing 5.2), is translated into explicit tree nodes that are pushed to and popped from a 16-entry

stack in the fast local memory on the FPGA (BRAM). Each of those tree nodes on the stack stores the position inside the search tree along with all already guessed and implied bytes of the candidate key schedule, as illustrated in Figure 5.5. For the EVT error model we additionally need to store the sums of previous (consumed) bit errors $\bar{n}_{1\rightsquigarrow 0}$ and $\bar{n}_{0\rightsquigarrow 1}$ as global information to correctly evaluate the bounding function. In contrast to an execution in software, we would not gain performance by using update and rollback mechanisms to save computations, because our stack has a sufficient bitwidth to access an entire tree node in parallel.

In the DFS-style branch-and-bound search of our case study, the checkpointed nodes correspond to an ascent or descent in the tree search (see example in Figure 5.4), keeping track of the current state: i.e. the current level of the search tree ($l \in \{0, 1, \ldots, 15\}$) and the next branch to consider ($g_0 \in \{0, 1, \ldots, 255\}$). The top-of-stack (TOS) pointer holds the current level of the search, highlighted in Figure 5.5. The checkpoints created by the state `GENERATE_LIVE_NODES` on the stack after a successful compatibility check are used to generate the next live nodes to process, which are used for backtracking as soon as all candidates for $g_0$ on a level have been tried and the considered branch is exhausted. A checkpoint needs to contain all information that is required by the `GUESS_NEXT_BYTE` superstate. This is the current position in the search tree (in our case all previously guessed byte values) as well as the next branch that would be taken. To avoid repetition of calculations, we also include all previously implied byte values and the current error counts in our checkpoints. Since a checkpoint is created for each movement down along the search tree, the number of elements on the stack implicitly represents the current search tree level. It is important to understand that each checkpoint completely defines the current state of the FSM. We will refer to the stack of checkpoints as *state stack* from now on, see Figure 5.5.

### 5.4.5 Maintaining the Bound: Applying Error Model

The bound is used to prune the search tree. In our application it is defined by checking the compatibility of the guessed and implied byte values against the decayed key schedule: i.e. if the byte values could have decayed to the observed key schedule in memory according to the error model. The state `CHECK_COMPATIBILITY` of the FSM in Figure 5.5 is responsible to check this property after each guess and its corresponding implications. We implemented the `CHECK_COMPATIBILITY` state in two variants, one reflecting the PAD error model and the other one implementing the EVT error model (see Section 3.2).

For the PAD error model, each byte can be checked independently for its compatibility. It is compatible if the two bytes match, or if bits are only flipped towards the ground state of the memory cell (see Lemma 3.2). In the `CHECK_COMPATIBILITY` state, this can be checked in parallel in a single clock cycle for the guessed and all implied bytes. For the EVT error model, at each guess the bit flips of $g_0$ and all implied bytes need to be summed up and compared to the expected values. The expected values are computed on the host CPU and added as parameters $n_{0\rightsquigarrow 1}$ and $n_{1\rightsquigarrow 0}$ to the search. This summation of bit flips is shown in Figure 5.8 and is similar to a balanced adder tree. We need to compute two separate sums for bits flipped in either direction $0 \rightsquigarrow 1$ or $1 \rightsquigarrow 0$, respectively, which is done in parallel. We use adders with the specific bit-widths required to represent the highest possible bit flip value at each level. Afterwards, we add the sums from the previous search level

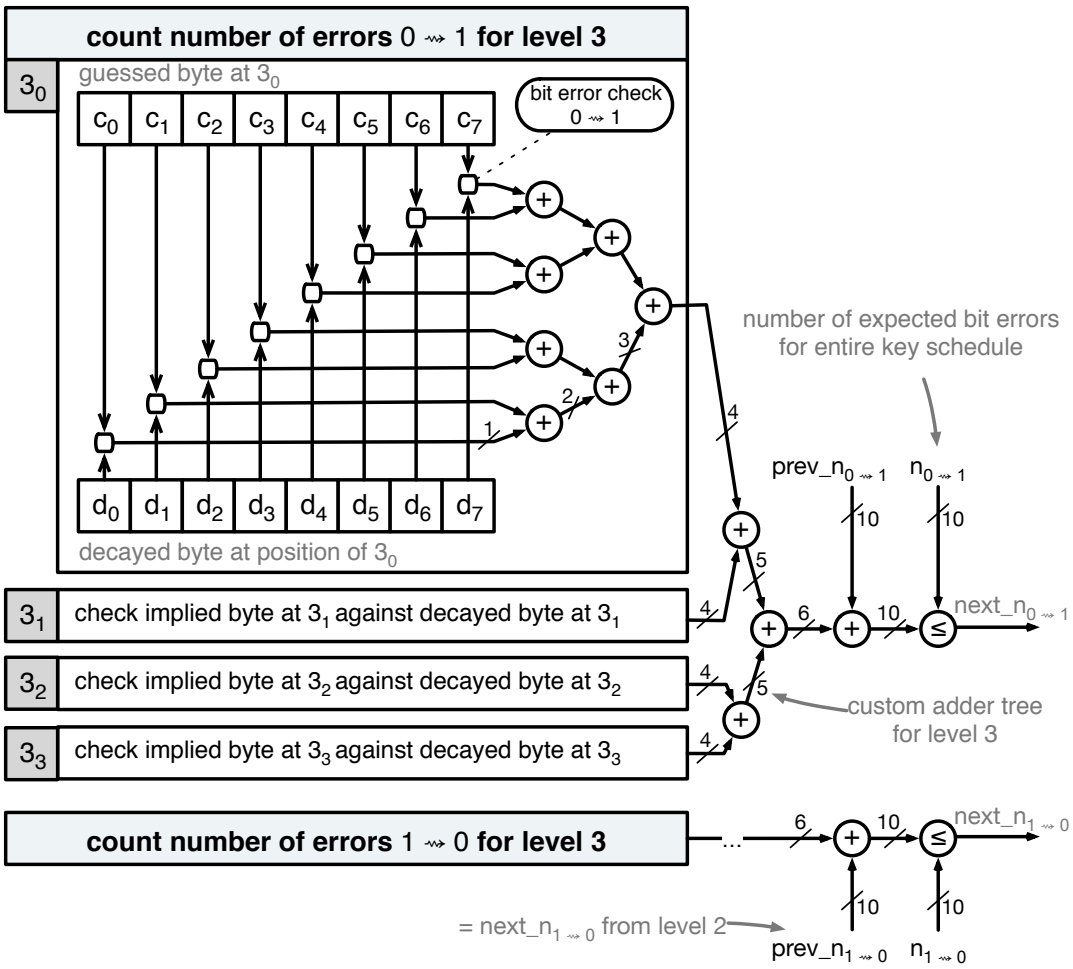



Figure 5.8: Checking compatibility for level 3 with the EVT error model for bit flip direction $0 \rightsquigarrow 1$. The values are stored in registers.

$\texttt{prev\_n}_{0\rightsquigarrow 1}$ ($\texttt{prev\_n}_{1\rightsquigarrow 0}$, respectively) to the bit flips of those bytes that were guessed or implied in the current level. If each value is less or equal to the expected values for the entire key schedule $n_{0\rightsquigarrow 1}$ and $n_{1\rightsquigarrow 0}$, the guessed value for $3_0$ along with the implied bytes are compatible and passed to the next level.

The size of the required adder tree reflects the number of derived bytes in each level (see Equation 5.2). Using this information, we added additional substates into the `CHECK_COMPATIBILITY` state of our FSM to split the summation for levels 8 and higher over multiple cycles. Otherwise, those levels become the critical path of our design and reduce the maximal achievable clock frequency. Nevertheless, the EVT error model still has a strong impact on our achieved clock frequencies, as shown in the synthesis results of our reconstruction kernels for the two error models in Table 5.3. Our implementation achieves 175 MHz on a Virtex-6 FPGA for the PAD error model, but only 90 MHz for the EVT model.

| | AESKeyFixPAD | AESKeyFixEVT |
|---|---|---|
| Used LUTs (%) | 6.65 | 8.74 |
| Used FFs (%) | 5.06 | 5.04 |
| Used BRAMs (%) | 1.41 | 1.41 |
| Used DSPs (%) | 0.00 | 0.00 |
| Achieved Frequency (MHz) | 175 | 90 |

Table 5.3: Synthesis results of two key reconstruction kernels targeting a Virtex-6 SX475T FPGA.

## 5.5 Evaluation

In this section, we evaluate our basic hardware design and implementation developed in this chapter, which only uses the static reconstruction allocation from Table 5.1 (page 46) and static incremental byte guesses. We show the raw overall potential offered by FPGAs for a branch-and-bound problem by comparing the results to a software implementation performing the same computations. Please note that the results discussed in this section are based on our conference article [2]. The main goal of the evaluation is to put our results in perspective to the state-of-the-art technique in software at the time of publication and to create a meaningful baseline for the following chapters, where we apply advanced techniques to further accelerate B&B problems with FPGAs.

### 5.5.1 Target Platforms

We executed the designs and implementations on two different platforms. The hardware designs are executed on the Maxeler system introduced in Chapter 2. The software baseline is mainly executed on the host processors of the same system. However, some very hard test cases would have exceeded the practicable runtime for solving them on one single system. Therefore, we used also used a cluster of CPU nodes. Results obtained by the CPU cluster are marked with **. In this section the specification of both systems is described. The same target platforms are also used for all further results from the following chapters. Therefore, we will refer back to this section, if required.

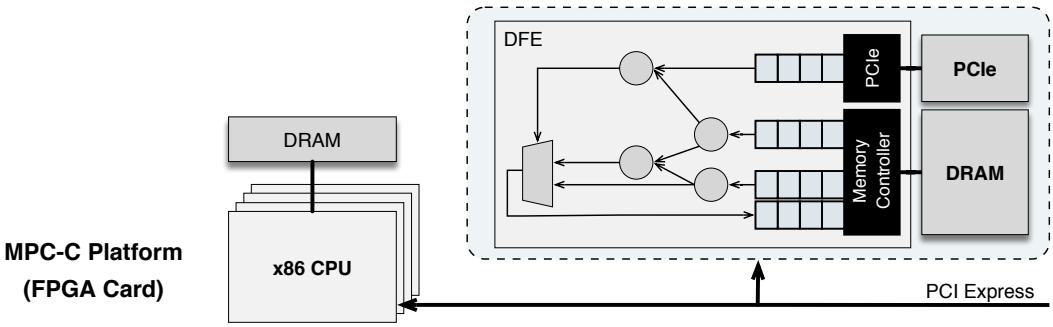



Figure 5.9: Maxeler MPC-C platform architecture.

#### Maxeler Data Flow Computer

The target platform for implementing our hardware design is the FPGA-based Maxeler MPC-C data flow computer system. This system is sketched in Figure 5.9 and the corresponding programming model is described in Section 2.3. The system is

a two-socket Intel Xeon server system with 48GB of DDR-3 SDRAM with up to four FPGA accelerator cards connected via PCIe. These FPGA cards feature a Xilinx Virtex-6 FPGA (XC6VSX475T) and 24GB of on-board DDR-3 SDRAM memory. The host uses two 6-core X5650 CPUs supporting hyper-threading and running at 2.67GHz.

#### CPU Cluster

The target platform to execute parts of the software reference is a CPU cluster called OCuLUS consisting of several hundreds of server nodes. We only use the CPU node declared as *small*. Each of these nodes consists of two Intel Xeon E5-2670 CPU with 8 cores each running at 2.6GHz and provides 64GB of main memory. Hyper-threading is supported, but disabled.

### 5.5.2 Error Metrics

In order to systematically evaluate the different designs and variants, a number of test cases with varying error rates is required.

Independently of the concrete error model that we discussed in Section 3.2, there are two different metrics in literature to describe the error rate of a found decayed key schedule. Tsow's [269] tests, which are limited to the perfect asymmetric decay (PAD) model, are designed on the basis of an error rate $d_{\text{Tsow}}$ that specifies for each bit in the key schedule the probability to switch to its ground state. Let the ground state be 0 throughout this section without loss of generality. Then an error rate of e.g. $d_{\text{Tsow}} = 60\%$ means that *around* 60% of all bits are decayed, so if they have been 1 before they became 0 and if they have been 0 before, they remained 0. The latter are clearly no bit errors, because a decay from $0 \rightsquigarrow 0$ preserves the correct value and all remaining ones can also be treated as correct exploiting the known bit lemma (see Section 3.2.1). Due to the cryptographic properties of AES, a key schedule has on average an equal amount of ones and zeros (see Appendix A). Thus, at an error rate of $d_{\text{Tsow}} = 60\%$, on average 50% of all bits have been zeros in the first place and remained zeros, either after decay or not. Further 30% of all bits are flipped $1 \rightsquigarrow 0$ and the last 20% remain ones. However, this rate of on average 30% bit flips $1 \rightsquigarrow 0$ varies a lot depending on how the actual distribution of ones and zeros in each concrete key schedule is and how they are struck by the random decay process. These considerations may affect the difficulty of the actual reconstruction task a lot with Tsow's decay metric, even for the same error rate.

The risk of having the same labeled tasks (e.g. a set of key schedules to reconstruct with 60% error rate) with different difficulties can have an impact on the comparability of different approaches and may be misleading. Furthermore, the error metric of Tsow only supports bit errors in one direction (e.g. $1 \rightsquigarrow 0$), which excludes the more realistic expected value as threshold (EVT) error model (see Section 3.2). Therefore, we use a second error metric proposed by Wang [281]. The error rate of Wang is defined as $d_{\text{Wang}} = d_{1 \rightsquigarrow 0} + d_{0 \rightsquigarrow 1}$. $d_{\text{Wang}}$ describes the total *exact* rate of bit flips (e.g. $d_{\text{Wang}} = 30\%$ means that *exactly* 30% of all bits are flipped). The single terms $d_{0 \rightsquigarrow 1}$ and $d_{1 \rightsquigarrow 0}$ describe the individual rates of zeros that became ones and ones that became zeros, respectively. This allows to easily support the EVT error model. To realize this error metric, we first determine the positions of the ones (zeros) in the key schedule and then invert them according to the desired individual error rates

$d_{0\rightsquigarrow1}$ ($d_{1\rightsquigarrow0}$). Therefore we consider bit errors in both directions and the result differs fundamentally for the same error rate compared to Tsow's metric. When using this definition of bit errors, the maximum error rate has to be much lower than the rates used by Tsow, since the expected number of ones (zeros) in the key schedule is 50% respectively, as argued above. Conceptually, Tsow's metric is closer to the actual decay process, whereas Wang's model better captures the bit flip rates of decayed key schedules retrieved from actual key search in memory. We use Tsow's error rate $d$ only for the comparisons with Tsow's results and otherwise utilize Wang's bit flip rates.

### 5.5.3 Evaluation Scenario

All key schedules used in this evaluation were created by first generating a key from 16 random bytes by reading from `/dev/urandom`, expanding it to a full 176 bytes key schedule using the AES-128 key expansion as described in Section 3.3.2 and finally randomly flipping single bits to simulate a memory decay according to one of the discussed metrics by Tsow or by Wang with the required error rates.

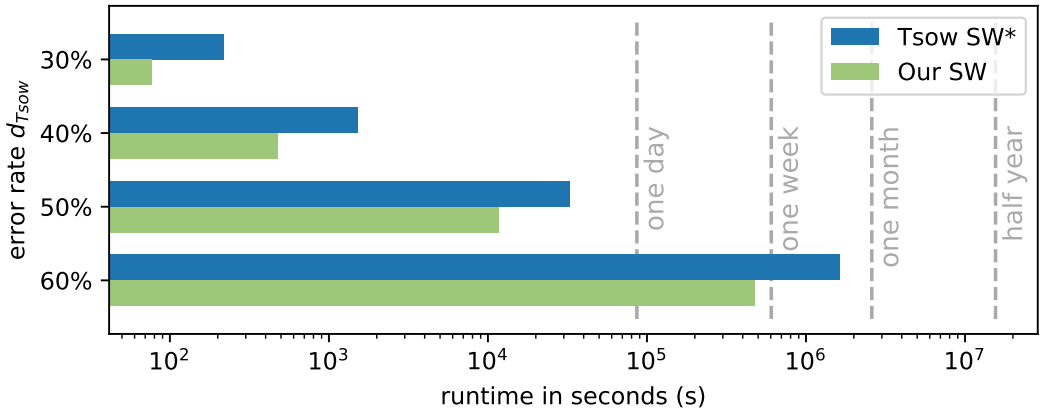
Since our basic accelerator developed in this chapter follows only the static allocation of Table 5.1 with incremental byte guesses, we always need to determine a compatible byte at the same position for $\mathbf{0_0}$ of the key schedule (top left at address 0). The time needed for reconstruction depends on the value of the searched compatible value. If the searched compatible value to guess is large, the reconstruction needs more effort because our state machine incrementally tries every possible value (0–255) and reaches the correct value very late. Hence, we need to average over a sufficiently large number of test cases to measure representative runtimes. Therefore, we generate 10,000 random AES-128 keys for our performance tests with the method described in the previous section. In Appendix A we show that the value for $0_0$ is equally distributed in the test cases. 10,000 decayed key schedules are also used in related work [270, 141] and represent a realistic workload for real cold-boot attacks. At a biflip rate of 30%, 422 bits of an AES-128 key schedule flip. In a concrete test, searching 1 GB of real memory contents for decayed key schedules with up to 422 bit flips ($\delta = 422$, see Section 4.1) resulted in 554 candidates. Extrapolating this to 16 GB of RAM, 8864 candidates might occur in this example, which would be approximated quite well by 10,000 key schedules. However, these numbers should only be considered as possible reference points and may greatly vary with error rates, memory type, memory size and the software stack of the system under attack.

### 5.5.4 Software Implementation

Since reference software was only partially available and had serious shortcomings with regard to our tests, we implemented our own software reference. Tsow's [269] software is written in C, but only supports the PAD error model. Wang's [281] software supports both error models, but is written in Java and around one order of magnitude slower than Tsow's implementation. In order to achieve good performance for our software, we implemented the reconstruction as an iterative algorithm to avoid copy operations on the call stack during recursion steps. Additionally, we introduced an improved guess procedure that only needs one instance of a candidate key schedule (array) to perform the guessing and implying of bytes. As mentioned

before, we extend the candidate key schedule in-place according to the static allocation from Table 5.1 on a valid assignment of $g_0$ (and its derived bytes) and use a rollback function to undo the changes respectively, if the computation on some level is exhausted.

Figure 5.10: Visualization of the comparison of our C implementation to numbers presented in the publication of Tsow (see Table 5.4).




| error rate $d_{\text{Tsow}}$ | | Sum | Average | St.Dev. | Speedup |
|---|---|---|---|---|---|
| 30% | Tsow* SW | 219.204 | 0.022 | 0.140 | |
| | Our SW | 76.454 | 0.008 | 0.124 | 2.9 |
| 40% | Tsow* SW | 1,526.308 | 0.153 | 2.994 | |
| | Our SW | 474.249 | 0.047 | 0.471 | 3.2 |
| 50% | Tsow* SW | 32,551.469 | 3.255 | 55.563 | |
| | Our SW | 11,602.430 | 1.160 | 25.752 | 2.8 |
| 60% | Tsow* SW | 1,638,788.166 | 163.879 | 3,753.608 | |
| | Our SW | 472,374.406 | 47.237 | 850.187 | 3.5 |

Runtimes in seconds (s)

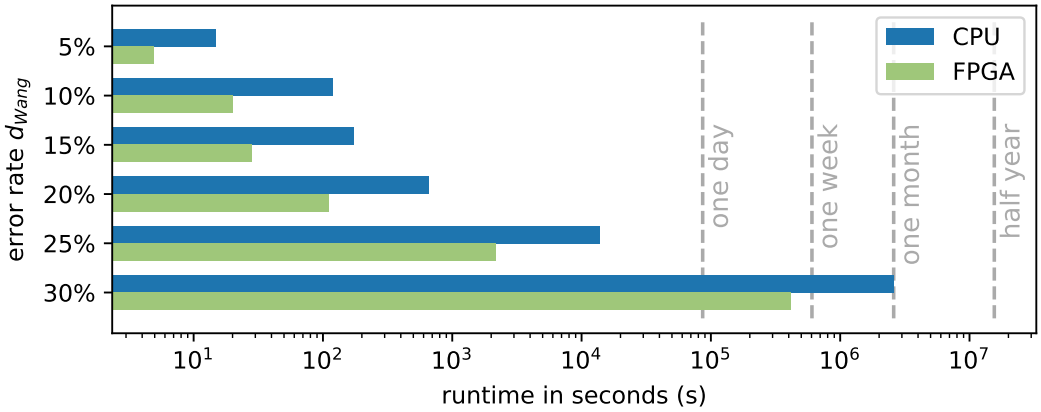
Table 5.4: Our C implementation executed on the host of the Maxeler system compared to numbers presented in the publication of Tsow [269] (marked as Tsow*).

Our software implementation supports both PAD and EVT error models. It is compiled with gcc-4.4.7 at the highest optimization level. We compared its PAD variant with Tsow's software numbers. Tests are executed with the static allocation of Table 5.1 for 10,000 test cases for each error rate $d_{\text{Tsow}} = \{30\%, 40\%, 50\%, 60\%\}$, same as in the article of Tsow and according to his definition of an error rate $d_{\text{Tsow}}$. Table 5.4 summarizes the results. A visual representation is depicted in Figure 5.10. Note that the listed values (marked with Tsow*) are obtained from the publication of Tsow and only serve for a rough comparison, since his test cases are not available and the used platform and compiler differs. So both computations are performed on different test cases, but with a sufficiently large and equal number of tests and the same error metric. Due to our code optimization and probably to some degree also due to our faster CPU, our software implementation achieves an overall speedup of around 3x over Tsow's implementation and is thus to the best of our knowledge the fastest one using the presented reconstruction technique at the time of publication.

### 5.5.5 Performance Comparison of Software to Hardware

We compare our software implementation to our presented hardware accelerator, also supporting both error models.

Figure 5.11: Visualization of the comparison of software versus hardware for the PAD error model with 10,000 test cases each (see Table 5.5).




| error rate $d_{\text{Wang}}$ | | Sum | Average | St.Dev. | Speedup |
|---|---|---|---|---|---|
| 5% | CPU | 14.901 | 0.001 | 0.007 | |
| | FPGA | 4.956 | 0.000 | 0.001 | 3.0 |
| 10% | CPU | 118.731 | 0.012 | 0.032 | |
| | FPGA | 20.095 | 0.002 | 0.008 | 5.9 |
| 15% | CPU | 174.841 | 0.017 | 0.080 | |
| | FPGA | 28.437 | 0.003 | 0.015 | 6.1 |
| 20% | CPU | 659.96 | 0.066 | 0.772 | |
| | FPGA | 110.835 | 0.011 | 0.136 | 6.0 |
| 25% | CPU | 13,895.451 | 1.390 | 21.399 | |
| | FPGA | 2,187.083 | 0.219 | 3.179 | 6.4 |
| 30% | CPU** | 2,599,493.566 | 259.949 | 3,912.294 | |
| | FPGA | 418,751.305 | 41.875 | 602.368 | 6.2 |

CPU** computed on cluster — Runtimes in seconds (s)

Table 5.5: PAD error model for 10,000 test cases each. The error rate corresponds to the metric of Wang ($d_{\text{Wang}} = d_{1\rightsquigarrow 0} + d_{0\rightsquigarrow 1}$ with $d_{1\rightsquigarrow 0} = \{5\%, \ldots, 30\%\}$ and $d_{0\rightsquigarrow 1} = 0$).

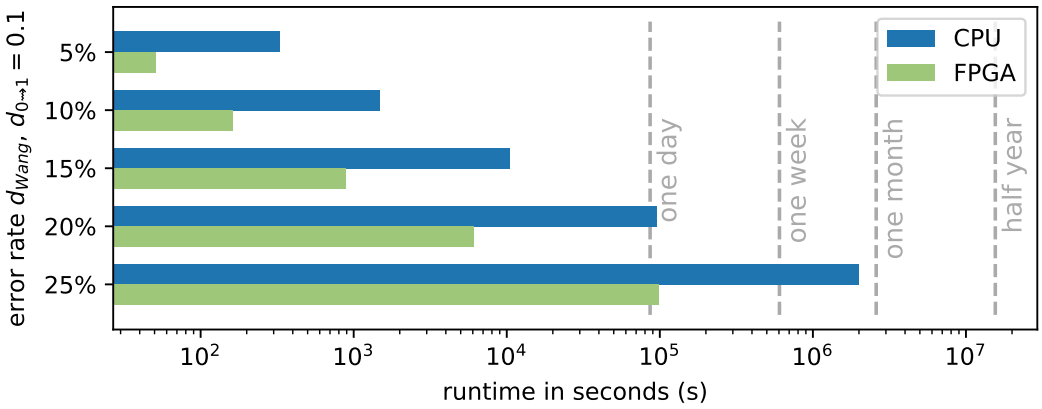
The results for the perfect asymmetric decay (PAD) error model are summarized in Table 5.5 with memory error rates varying from 5% to 30% according to the presented error metric by Wang. The visualization of the data is depicted in Figure 5.5. The hardware achieves a speedup of around 6× for all bit flip rates except for the lowest tested bit flip rate, where the individual runtimes are small enough to be affected by call overheads to the hardware, which limits the speedups to 3× in this single case. We consider this case as an outlier caused by small overall runtimes. For this error rate, the transfer time of the data to and from the accelerator dominates the overall computation time to recover the key schedule (495 ms on average). Our accelerator successfully recovered all keys for all error rates. With an overall average speedup of 5.6x (6.1x without the smallest error rate) we expect for the highest error rate $d_{\text{Wang}} = 30\%$ a runtime of around 27 days for the C implementation, if it is performed sequentially on one CPU. Since this would exceed the practicability of a

real attack, we used a cluster of hundreds of CPU nodes (all equipped with the CPU mentioned above). Each of the 10,000 test cases for this error rate was allocated to a central scheduler, which assigned them to a free node. If the job gets a free resource, the computation is performed exclusively without any concurrent task. Note that the presented runtime (marked with **) for this error rate is the pure sum of each runtime, without the allocation time to the central scheduler and the waiting for a free resource. With this approach, the software implementation has a sequential runtime of about 30 days. In comparison our hardware implementation solved all test cases for all error rates within five days on a single FPGA card.

For expected value as threshold (EVT), we performed two test series, one with a bit flip rate into the opposite direction of the ground state of $d_{0\rightsquigarrow 1} = 0.1\%$ and one with $d_{0\rightsquigarrow 1} = 0.2\%$ with total memory error rates varying from 5% to 25%. The results are summarized in Table 5.6 and Table 5.7. The visual representations are in Figure 5.12 and Figure 5.13. The overall runtimes increase with higher bit flip rates both for CPU and FPGA. However, the FPGA implementation apparently scales better, so the speedups increase from 6.5× for $d_{1\rightsquigarrow 0} = 4.9\%$ and $d_{0\rightsquigarrow 1} = 0.1\%$ (first row in Table 5.6) up to 27× for $d_{1\rightsquigarrow 0} = 24.8\%$ and $d_{0\rightsquigarrow 1} = 0.2\%$ (last row in Table 5.7). On the basis of this observation we expect for the highest error rate of 25% a sequential runtime of about 20 days for $d_{0\rightsquigarrow 1} = 0.1\%$ and about 120 days for $d_{0\rightsquigarrow 1} = 0.2\%$ in software on one CPU. Similar to our approach for PAD model, we used the cluster of 900 CPU nodes to distribute this case and present the sum of each discrete runtime (marked with **). The overall sequential execution time for the software implementation is over 23 days for $d_{0\rightsquigarrow 1} = 0.1\%$ and over 123 days for $d_{0\rightsquigarrow 1} = 0.2\%$, whereas the overall execution time on one FPGA card for all test cases and all error rates is 29 hours for $d_{0\rightsquigarrow 1} = 0.1\%$ and four and a half days for $d_{0\rightsquigarrow 1} = 0.2\%$, respectively.

As discussed before, the reconstruction difficulty of the various decayed key schedules differs a lot even for the same error rates. This is also reflected in the high standard deviations in all result tables. Therefore, even when large amounts of parallel resources are available, in our case of several hundreds software nodes, the achievable runtimes for an entire workload of 10,000 candidates to reconstruct have a lower limit given by the longest running individual reconstruction. In our experiments, this was about three days for $d_{0\rightsquigarrow 1} = 0.1\%$ and twelve days for $d_{0\rightsquigarrow 1} = 0.2\%$ in software, which is almost three times more than the entire runtime for all 10,000 reconstructions on one single FPGA accelerator. So in this regard, a single FPGA system outperforms the utilized CPU cluster not only by orders of magnitude in asset cost and energy consumption, but still delivers more practical usefulness. As a result, the advantage of the FPGA implementation grows in particular for difficult problem instances when the runtimes become a practical issue.

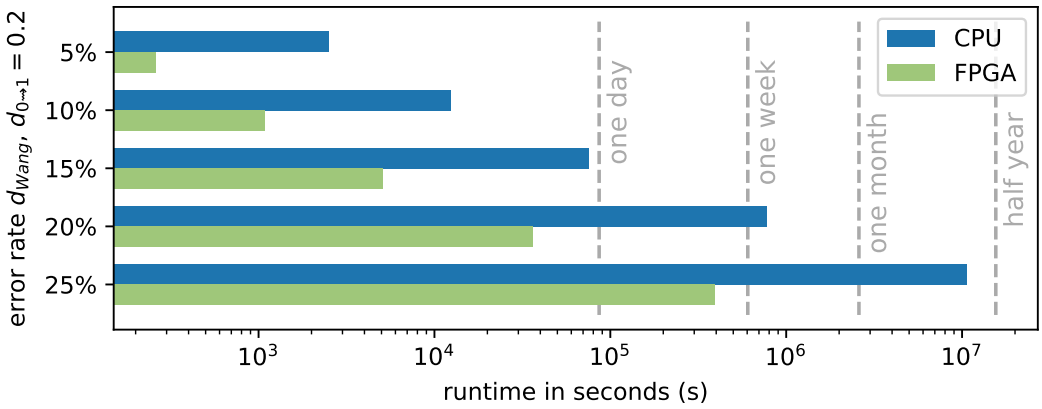
Figure 5.12: Visualization of the comparison of software versus hardware for the EVT error model with $d_{0\rightsquigarrow 1} = 0.1\%$ (see Table 5.6).




| error rate $d_{\text{Wang}}$ | | Sum | Average | St.Dev. | Speedup |
|---|---|---|---|---|---|
| 5% | CPU | 329.714 | 0.033 | 0.728 | |
| | FPGA | 50.499 | 0.005 | 0.027 | 6.5 |
| 10% | CPU | 1,483.592 | 0.148 | 1.132 | |
| | FPGA | 161.574 | 0.016 | 0.109 | 9.2 |
| 15% | CPU | 10,428.147 | 1.043 | 10.305 | |
| | FPGA | 881.636 | 0.088 | 0.832 | 11.8 |
| 20% | CPU | 95,249.41 | 9.525 | 142.731 | |
| | FPGA | 6,030.007 | 0.603 | 8.397 | 15.8 |
| 25% | CPU** | 1,996,589.189 | 199.659 | 3,071.691 | |
| | FPGA | 98,269.133 | 9.827 | 117.238 | 20.3 |
| $d_{0\rightsquigarrow 1} = 0.1\%$ | | Runtimes in seconds (s) | | | |

Table 5.6: EVT error model with decay opposite direction of the ground state $d_{0\rightsquigarrow 1} = 0.1\%$ and varying total error rate from 5% to 30%.

Figure 5.13: Visualization of the comparison of software versus hardware for the EVT error model with $d_{0\rightsquigarrow 1} = 0.2\%$ (see Table 5.7).




| error rate $d_{\text{Wang}}$ | | Sum | Average | St.Dev. | Speedup |
|---|---|---|---|---|---|
| 5% | CPU | 2,501.573 | 0.250 | 2.845 | |
| | FPGA | 260.108 | 0.026 | 0.257 | 9.6 |
| 10% | CPU | 12,335.190 | 1.234 | 21.820 | |
| | FPGA | 1,089.177 | 0.109 | 1.776 | 11.3 |
| 15% | CPU | 75,171.739 | 7.517 | 83.372 | |
| | FPGA | 5,095.195 | 0.510 | 6.530 | 14.8 |
| 20% | CPU | 773,118.366 | 77.312 | 909.462 | |
| | FPGA | 36,250.274 | 3.625 | 30.053 | 21.3 |
| 25% | CPU** | 10,636,553.810 | 1,063.655 | 13,973.626 | |
| | FPGA | 392,347.162 | 39.235 | 335.880 | 27.1 |

$d_{0\rightsquigarrow 1} = 0.2\%$ Runtimes in seconds (s)

Table 5.7: EVT error model with decay opposite direction of the ground state $d_{0\rightsquigarrow 1} = 0.2\%$ and varying total error rate from 5% to 30%.

## 5.6 Chapter Conclusion

In this chapter, we started with the basic ideas of the branch-and-bound algorithmic pattern. We presented the general principles by defining five main operations and discussed required transformations to translate the operations into a finite state machine suitable for FPGAs.

We used the reconstruction of erroneous AES keys as our case study to show our concepts on a real and relevant application problem. The reconstruction of erroneous keys is a crucial operation in cryptography and very time consuming when trying to break encryption systems with side-channel attacks (for example, through cold-boot attacks). In software, the reconstruction of the AES master key can be performed using a recursive branch-and-bound tree search algorithm that exploits redundancies in the key schedule for constraining the search space. In this chapter, we investigated how this branch-and-bound algorithm can be implemented on an FPGA. We showed how a recursive search procedure in software can be translated step-by-step to general finite state machine states with an explicit stack that stores the context for each recursion level.

For the designed FSM architecture, we showed where and how highly optimized combinational datapaths can be created to accelerate in particular the processing of the most frequently accessed tree levels, which is crucial to gain performance. For the less frequent and more complex lower levels we showed a more resource-efficient pipelined design.

Finally, we evaluated the basic design hardware design in different scenarios and compared it to a competitive software reference. Even though both implementations perform algorithmically exactly the same operations, our FPGA implementation is able to utilize the low-level parallelization and spatial processing to outperform the software reference.

Although this design is not instance-specific and processes the search tree sequentially with only one worker, it was the state-of-the-art in performance for AES key reconstruction at the time of the publication [1]. However, the evaluation also revealed that the computation time is mainly dominated by a few especially difficult problem instances. Even after applying the branch-and-bound algorithmic pattern on an FPGA with a very tight and efficient bounding function, the search can still require a significant amount of time. The main reason is that it is unknown in advance how much computation time each generated subproblem needs. On the other hand, the processing of the individual subproblems during the search with the branch-and-bound principle is independent of each other. Hence, a method to efficiently parallelize, distribute and process the computation should improve the performance. Therefore, we investigate in the next Chapter 6 how a parallelization strategy for B&B problems can be applied.

# Chapter 6

# Work Stealing with Reconfigurable Hardware

Even after applying the branch-and-bound (B&B) algorithmic pattern with an efficient bounding function, the search trees are typically very large and the sequential exploration of all promising subproblems with one single worker is very time consuming. Hence, a method to efficiently distribute and process the computation is required. In this chapter we describe the extension of our general, but sequential branch-and-bound design developed in Chapter 5 to allow parallelization of the work in reconfigurable hardware. We describe the necessary changes and considerations in detail to parallelize the branch-and-bound algorithmic pattern using work stealing (WS) over several hardware workers. Work stealing is a well-known parallelization strategy for parallel computers [34], but has hardly been studied for FPGAs.

In Section 6.1, we give an introduction to the parallelization of a task using work stealing and motivate why this strategy is very promising for branch-and-bound problems. Then, in Section 6.2, we design and implement the required extensions to the sequential state machine from the previous chapter to allow parallelization. Besides the required architectural changes, we will focus on the coordination and synchronization of the stealing. In Section 6.3, we evaluate the performance of our design compared to a software implementation and finally, we draw a conclusion in Section 6.4.

## 6.1 Motivation and General Description

Besides the trivial parallelization of branch-and-bound by starting another instance of the search on another device or context such as described for password breaking in Section 8.1.3, Lai et al. [166] discuss essentially three ways to introduce parallelism into B&B operations:

1. Select more than one node to process from the pool of live nodes.
2. Compute additional knowledge and bound values inferred by the selected node in parallel.
3. Use parallelism to generate promising live nodes.

The parallelism of type 1 corresponds to task-level parallelism, where the search tree is explored in parallel. In Listing 6.1 we recall the general algorithm structure and highlight the operations where parallelism can be introduced. The parallelism of type 2 corresponds to a form of data-level parallelism [101], where operations

on subproblems (e.g. inferring knowledge and bounding the selected node) are executed in parallel, while the exploration of the search tree is sequential. In software, there are several techniques available to implement and accelerate branch-and-bound algorithms. One technique for acceleration is to parallelize the execution on multiple CPUs. This can be done by statically dividing the tree into multiple subtrees, each processed by a different worker (work sharing [145]). As it cannot be known in advance how long a traversal of each subtree will take, this may lead to workers running idle before a solution is found. Therefore a dynamic load balancing using *work stealing* [34, 84, 13, 58] is more powerful for the parallelization of B&B algorithms. When a worker runs idle, it autonomously competes for new work items by stealing them from others. A centralized distribution of work items is not required.

In every parallelization strategy, instead of one there are, $N_w$ workers. As a basic abstraction of work stealing, every worker maintains an own data structure of work packages. A work package is the smallest amount of work that can be addressed. At the beginning, the processing can either start at one worker and is then further distributed to the others or every worker receives some work packages to start with. If a worker has processed all its work packages and becomes idle, it tries to gather or *steal* a work package from another worker. Starting with a single package, the execution may cause it to be split into two separate work packages: the *continuation*

```
1  # Initialization.
2  bound ← COMPUTE_INITIAL_BOUND( root )
3  pool ← { root }
4  solution ← NULL
5
6  # Main loop.
7  while pool ≠ ∅ :
8    # Select more than one node to process from the pool of live nodes.
9    cur_node ← COMPUTE_BRANCH( pool )
10   pool_remove( cur_node  )
11
12   # Compute additional knowledge and bound values inferred by
13   #   the selected node in parallel.
14   INFER_KNOWLEDGE( cur_node )
15   cur_bound ← COMPUTE_BOUND( cur_node )
16
17   # Validate, if current bound is valid.
18   if check_bound(cur_bound, bound) == VALID :
19     # Check, if the seeked value or better solution is found.
20     # Update bound and store solution accordingly.
21     if solution_found(cur_node, cur_bound, bound, solution) :
22       continue
23
24     # Otherwise, use parallelism to generate next nodes
25     #   to process and add to pool.
26     cur_node_children ← GENERATE_LIVE_NODES( cur_node )
27     pool_add( cur_node_children )
28   else:
29     # Node and subtree cannot lead to a feasible solution.
30     discard( cur_node )
```

Listing 6.1: General algorithm with operations highlighted where parallelism can be introduced.

of the original work package and a new *child* work package. Each of the packages may be split again recursively. Depending on the chosen strategy, the current worker starts processing the child package or the continuation package. Other idle workers may then steal the other work package correspondingly. After all the work is done, the results are collected and the execution is complete. Work stealing is a common load balancing technique when static division of labor (work sharing) is not feasible (e.g. because work packages are of different sizes or the duration of their execution cannot be estimated in advance, which is typically the case for branch-and-bound problems). Hence, work stealing is a well-suited strategy for this kind of application.

In the next section, we describe the required transformations and modifications to parallelize the finite state machine (FSM) discussed in the last chapter, starting with Section 5.2.2 to use all three types of parallelism.

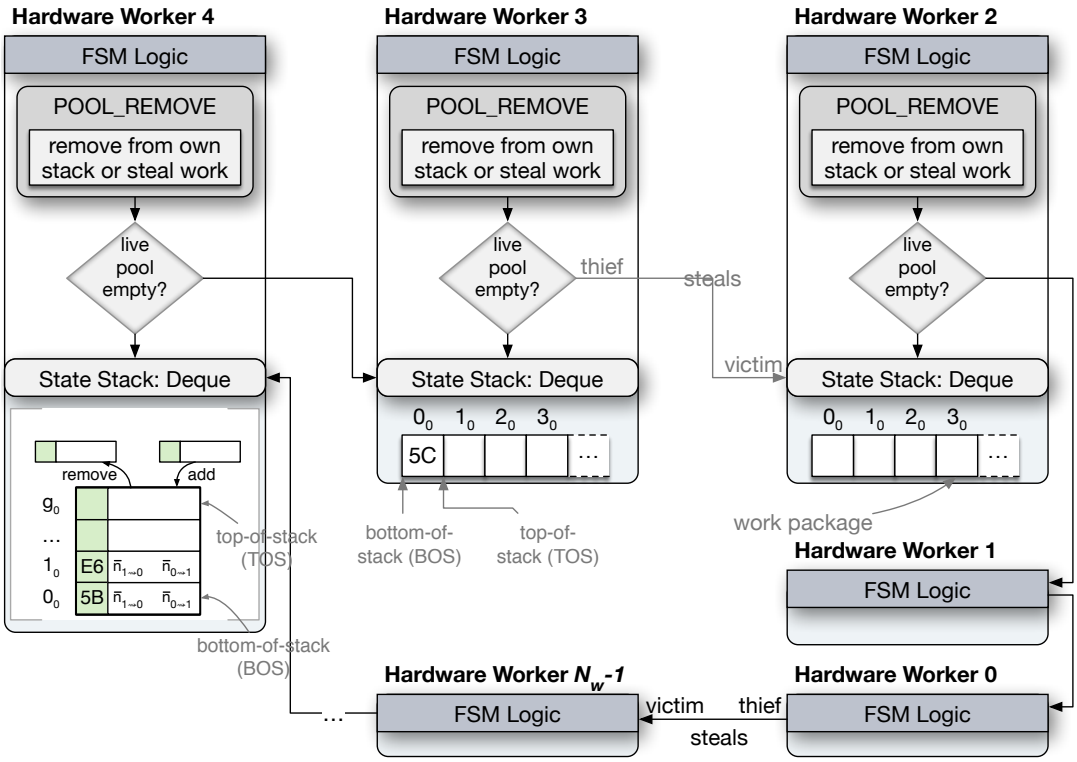



Figure 6.1: The original FSM is duplicated $N_w$ times to create the required number of hardware workers. The hardware workers expose their state stacks storing the checkpoints to share the work items.

## 6.2 Extensions of the General State Machine

The first step to support work stealing in hardware is to create the hardware workers that are able to maintain their data structure of work packages. In our work stealing design, the original FSM developed in the last chapter as well as its state stack (pool of live checkpoints) are duplicated $N_w$-times so that each FSM acts as one hardware worker. Figure 6.1 shows schematically the replication of the FSMs and indicates the communication that is required to handle the transfers to share and steal work. To enable this communication, especially extensions to the accesses

through the POOL_REMOVE state are required.

The state stack from Section 5.4.4 in its previous version for one sequential worker contained information about the parts of the search space that were already visited or discarded. The position of the top-of-stack (TOS) in the state stack of the FSM determined in which level of the tree the worker is currently located. Work packages located low on the state stack represent large subtrees and therefore tend to contain more work than packages located higher on the state stack (see statistics in Figure 5.6). To coordinate tree traversal between different workers an additional bottom-of-stack (BOS) pointer is introduced that marks the level of the tree up to which workers will backtrack when no solution in the considered branch is found. This prevents workers from entering parts of the search tree that are examined by other workers. By introducing this BOS pointer, the previous stack data structure to handle the pool of live nodes becomes a double-ended queue (deque), see bottom left of Figure 6.1. We define each entry on the deque between BOS and TOS as a *work package*, which can either be processed by the worker itself or stolen by other workers. Entering a deeper level of the tree by pushing the state to the deque is equivalent to creating a new child work package and beginning to process it.

```
1  # Check if own deque self.deque contains work. If so,
2  #   load a work package from top-of-stack (TOS) and process it.
3  if COUNT( self.deque ) > 0 :
4    PROCESS ( self.deque.TOS )
5  else :
6    # Otherwise, try to acquire work from another workers.
7
8    # Check if work is available and if stealing
9    #   would interfere with victim's deque access.
10   if ( COUNT( victim.deque ) == 1 and victim.state == POOL_REMOVE )
11          or COUNT( victim.deque ) == 0 :
12     # Stealing is unsafe. Idle and try later.
13     idleCannotSteal ← true
14   else :
15     # Stealing is safe. Acquire and process
16     #   work package from victim's bottom-of-stack (BOS).
17     idleCannotSteal ← false
18     # Create space on own deque.
19     self.deque.BOS ← victim.deque.BOS
20     self.deque.TOS ← victim.deque.BOS + 1
21     # Copy work package from victim's to own deque.
22     copy_work_package( victim.deque.BOS, self.deque.BOS )
23     # Remove work package from victim's deque.
24     victim.deque.BOS ← victim.deque.BOS + 1
25     # Load and process stolen work package.
26     PROCESS ( self.deque.TOS )
```

Listing 6.2: Work stealing extension in POOL_REMOVE state. If the FSM has no elements on own deque, it steals from a victim.

Listing 6.2 shows the work stealing extension of a worker entering the state POOL_REMOVE of the FSM. If the bottom-of-stack and top-of-stack pointers of its own deque are not identical (COUNT( self.deque ) > 0, lines 3–4), the worker still has work packages left on its own deque and continues with the processing at the TOS position. Otherwise, the whole search subspace is explored and the worker needs to steal packages from another worker (lines 6–26). Stealing work from a victim FSM

by a thief FSM is realized by copying the bottommost element from the victim's state stack to the corresponding position on the thief's state stack (line 22) and incrementing the victim's BOS by one (line 24). The BOS and TOS of the thief are set to the recently stolen work package (lines 19–20). Moving the BOS of the victim ensures that it will not enter the search subspace that is now explored by the thief.

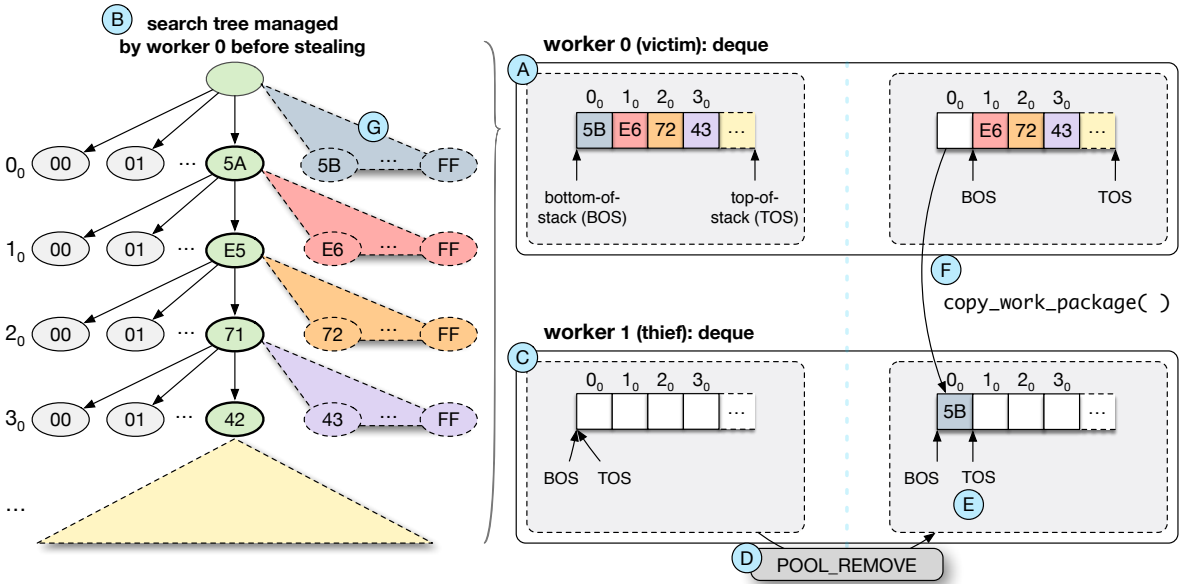



Figure 6.2: Stealing work by copying the bottommost stack entry.

The whole functionality of stealing work is shown in Figure 6.2 with the help of an example of two workers. Before the stealing, worker 0 (the victim) has several work packages in its deque (A). The worker processes its own work packages from the top-of-stack (TOS) pointer (line 4 in Listing 6.2). Each work package in the deque has a distinct color in Figure 6.2 that represents a specific part in the search tree (B) that is managed by this worker. The second worker 1 (the thief) reaches its `POOL_REMOVE` state with an empty deque (C). The thief tries to acquire a work package by stealing it from the victim's deque (D). Therefore, the thief allocates space on its own deque (E) and copies the work package located at the bottom-of-stack (BOS) pointer of the victim (F). After stealing, the thief is responsible for the processing of the part of the search tree labeled with (G).

There are two major challenges in this approach: the choice of a victim FSM to steal work from, and the safe synchronization between accesses to all the state stacks. Both challenges are tackled in the next section.

### 6.2.1 Coordination and Synchronization of Stealing

A victim for stealing can be chosen uniformly at random among all available FSMs, which is done in most software implementations and is known to be efficient [34]. Realizing a randomized approach in hardware requires access to all state stacks by all workers, an arbitration mechanism and a pseudo-random number generator. The practicability of this approach depends on the number of workers and the amount of data stored on the state stacks.

Allowing all workers to steal from all other workers would require a large number of multiplexers. The number increases quadratically with the number of workers $N_w$. For our work stealing implementation on FPGAs, we therefore do not use a randomized approach but make the following simplification to reduce the hardware footprint: The hardware worker $x$ only steals work packages from its direct neighbor $(((x-1)+N_w) \mod N_w)$, i.e. workers are arranged in a ring topology as depicted in Figure 6.1. This restriction simplifies synchronization of state stack accesses between different workers, as only the worker itself and one of its direct neighbors can access its state stack. During normal operation a worker only changes the TOS of its own stack (line 4 of Listing 6.2) and a thief only the BOS of its victim's stack (line 24); therefore concurrent access is possible and safe in general. However, if the victim itself is in the `POOL_REMOVE` state to fetch a work package and it is the only one left on the deque, stealing would lead to a conflict. In this case, stealing is postponed until the next clock cycle by raising the `idleCannotSteal` signal (lines 10–13).

Apart from the conserved synchronization effort, the ring structure allows to efficiently spread workers over a large area of the FPGA as long as they are placed close to their two neighbors. At the same time, it is ensured that all workers are eventually able to steal work packages. In our evaluation we show that our simplified stealing approach shows negligible impact on efficiency while allowing to add additional workers with only constant additional resource costs.

### 6.2.2 Initialization and Termination

The initialization and termination of our extended work stealing FSMs require further considerations. On initialization the whole search space (root node of the search tree) is assigned to the first hardware worker 0. During processing the worker eventually finds compatible assignments confirming the bounding function and spawns new work packages by pushing its state to its own state stack. If this is done, its neighbor can steal the continuation of the first work package. Following this process, eventually all workers have a neighbor with work packages to steal from. The search space is dynamically distributed and work balancing is ensured by the work stealing principle.

For the termination there are two possible conditions: either one worker has found a feasible solution or all workers are idle and none of them has found a feasible solution. In the first case the corresponding FSM that found a feasible solution enters the `SUCCESS` state (see Figure 5.5) and its result gets transfered back to the host computer. In the unlikely case that more than one FSM wants to enter this state in the same clock cycle, the one with the lower ID gets priority. For the second case, if the search is exhausted without any feasible solution, each FSM is extended by an additional `idleCannotSteal` signal, which is asserted when the corresponding worker becomes idle, no work packages can be stolen and no result has been found (line 13 of Listing 6.2). As soon as this signal is set by all hardware workers, worker 0 enters the `FAILURE` state and the host computer is informed that no solution was found and terminates the search.

With the described changes, we are able to extend our original hardware design from Chapter 5 to use work stealing with several finite state machines. Each FSM is

able to work in parallel on different parts of the search space and dynamically obtain work packages from other hardware workers.

## 6.3 Evaluation

In this section, we systematically evaluate our parallelization strategy using work stealing and compare it with the static baseline implementation in soft- and hardware from Chapter 5.

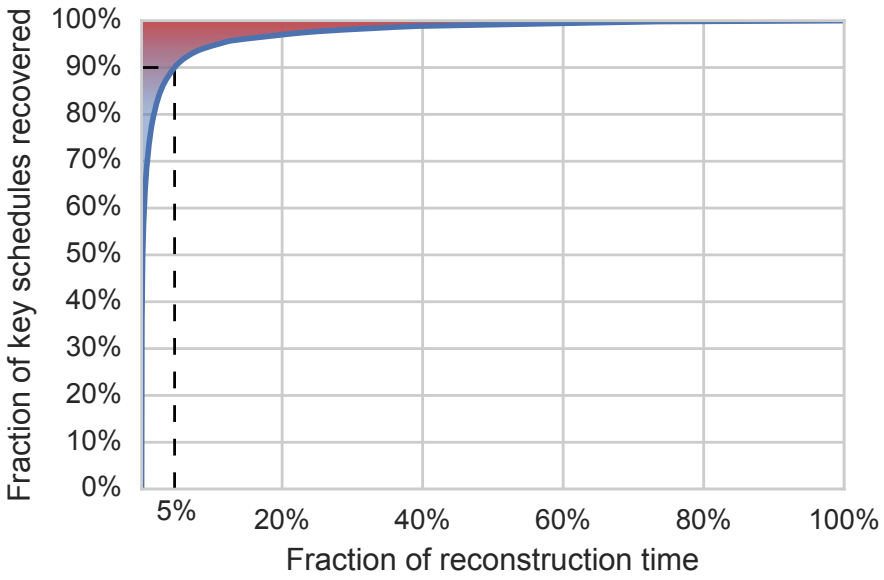



Figure 6.3: Few key schedules dominate the runtime.

### 6.3.1 Evaluation Scenario

We modify our evaluation scenario in comparison to Chapter 5 and focus only on the hardest problem instances. In this and the following chapters, we look only at the EVT error model and assume an error rate of $r_{1\rightsquigarrow 0} = 29.9\%$ and $r_{0\rightsquigarrow 1} = 0.1\%$, leading to a total error rate of $d_{\text{Wang}} = 30\%$. This total error rate is higher than all cases studied but still allows extensive evaluation. We focus on particularly hard reconstruction problems to show the potential of our different advanced strategies. Choosing even higher error rates would render the comparison to our baseline in software impracticable, because reconstruction of some key schedules would take several weeks.

The runtime required for the reconstruction of a large number of key schedules is very unevenly spread over the single key schedules. Figure 6.3 shows how the total reconstruction time develops when the key schedules are processed in increasing order of their difficulty[1]: 90% of all key schedules can be reconstructed in under 5% of the total required time, while the other 10% are accountable for 95% of the required time.

[1]Note that the runtime is not known a priori and the key schedules were reordered after execution.

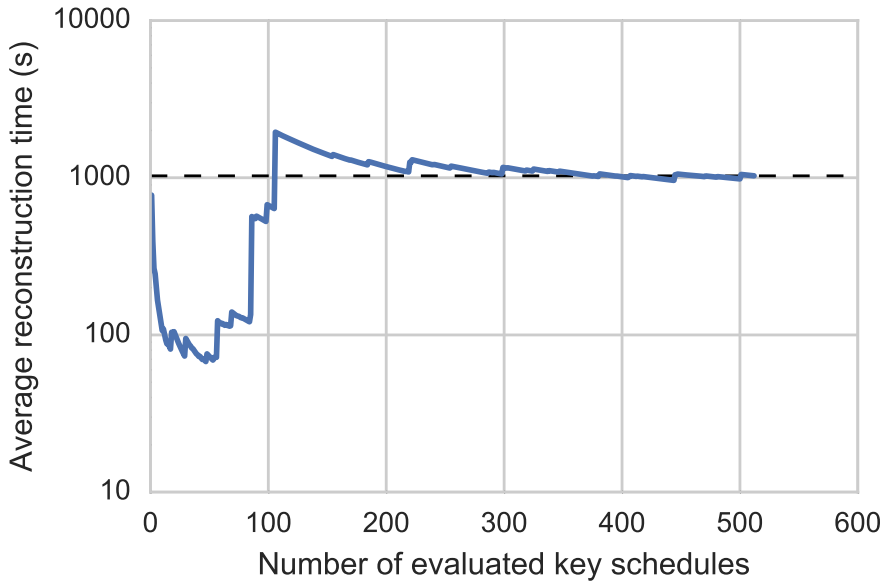


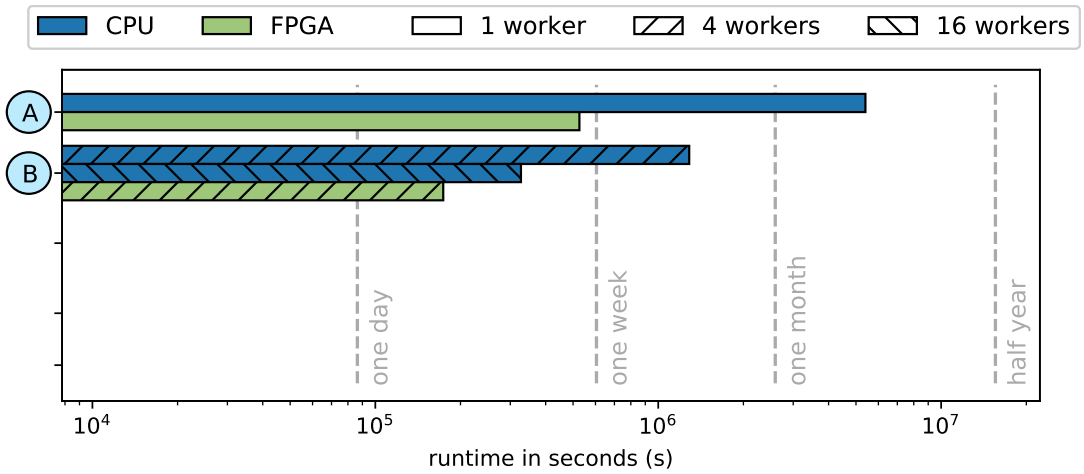

Figure 6.4: Evaluating 512 key schedules provides stable results.

Due to the high error rate, an evaluation based on 10,000 key schedules like in the previous section would have been impractical. Using the baseline hardware implementation for the reconstruction of 10,000 keys would take 10,280,000s, which corresponds to nearly four months. According to the results presented in Chapter 5, the software solution is at least a magnitude slower and therefore would take more than three years. Still, it is important to base the evaluation on a sufficient number of key schedules to make sure that not only relatively easy but also very hard reconstruction problems are considered. To determine how many key schedules should be used in this evaluation of very hard problem instances, we observed the average reconstruction time for a single key schedule depending on the number of evaluated key schedules. Figure 6.4 depicts the results for 512 different key schedules. It shows that the average reconstruction time varies significantly for the first 200 key schedules, due to a rapid increase in the average reconstruction time as soon as a particularly hard key schedule has to be reconstructed. Then the curve flattens out, because even for hard key schedules the reconstruction time of a single key schedule becomes negligible compared to the total runtime. For 512 key schedules the variance shows to be insignificant. Hence, we use the generated set of 512 key schedules throughout the evaluation of the advanced techniques.

### 6.3.2 Results

The measured runtimes to reconstruct all 512 key schedules using work stealing are shown in Table 6.1. The visualization of the data is depicted in Figure 6.5. To verify the observed speedups we repeated the measurements using software implementations. As an evaluation of the software implementation on the host platform of the Maxeler machine would have taken too long, we performed it completely on the OCuLUS computing cluster described in Section 5.5.1. Figure 6.6 shows a visualization of the observed speedups. We will use this visualization to show the impact of our incrementally added features for the rest of this thesis.

Figure 6.5: Visualization of the reconstruction of 512 key schedules using work stealing with a varying number of workers (see Table 6.1).




| | Feature Set | Number of Workers | Sum | Average | Standard Deviation | Total Speedup |
|---|---|---|---|---|---|---|
| CPU | (A) Baseline Software | 1 | 5,399,100 | 10,545 | 65,056 | 1.0 |
| | (B) Parallelization using work stealing | 4 | 1,286,770 | 2,513 | 15,630 | 4.20 |
| | —— " —— | 16 | 327,164 | 639 | 3,988 | 16.50 |
| FPGA | (A) Baseline Hardware | 1 | 526,453 | 1,028 | 7,271 | 1.0 |
| | (B) Parallelization using work stealing | 4 | 173,806 | 340 | 2,414 | 3.02 |

Runtimes in seconds (s)

Table 6.1: Reconstruction of 512 key schedules using work stealing with a varying number of workers.

Our work stealing design can be used for an arbitrary number of workers. Due to the increase of utilized chip area on the FPGA and overall complexity, routing these designs becomes harder for higher numbers of workers, resulting in lower achievable clock rates. We determined utilization and the highest achievable clock rate for different numbers of workers, shown in Table 6.2. Assuming that $N_w$ workers achieve an ideal speedup of $N_w$ times over a single worker, we calculated an equivalent clock rate for a single worker for each design. For the shown values, the highest overall performance is achieved using six workers. The resource utilization for different numbers of workers shows that after a large rise in utilization for the introduction of work stealing, only constant additional resources (around 12%) are required for each additional worker.

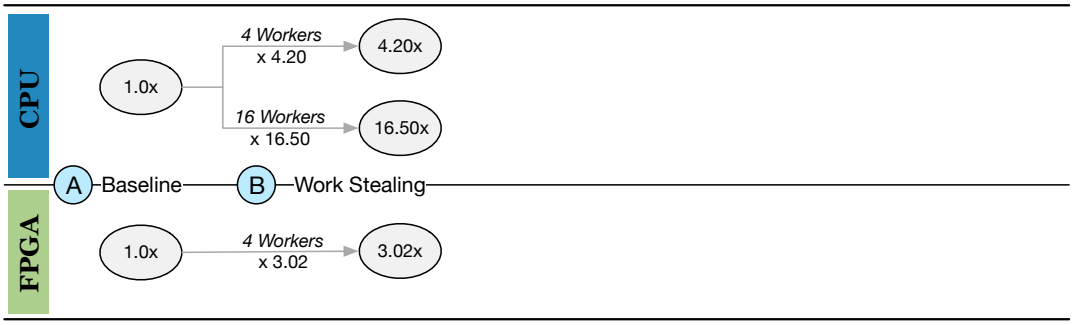



Figure 6.6: Visualization of the incremental speedups achieved using work stealing.

| Number of Workers $N_w$ | Utilization*** [%] | Clock [MHz] Max | Clock [MHz] Used | Single worker equiv. Max | Single worker equiv. Used | Used in evaluation |
|---|---|---|---|---|---|---|
| 1 | 5.9 | 85 | 80 | 85 | 80 | ✓ |
| 2 | 27.8 | 75 | | 150 | | |
| 3 | 40.7 | 75 | | 225 | | |
| 4 | 52.2 | 70 | 60 | 280 | 240 | ✓ |
| 5 | 65.2 | 55 | | 275 | | |
| 6 | 76.2 | 50 | | 300 | | |

***Fraction of LUT-FF pairs utilized on a Xilinx XC6VSX475T

Table 6.2: Synthesis results for different numbers of hardware workers $N_w$.

As synthesis time significantly increases with the number of workers and the target clock rate, we limited our design to four workers. For the same reasons, we did not target the maximum achievable clock rates but chose 80 MHz for our single-worker design and 60 MHz for our four-worker design. With these clock rates synthesis times are below 2.5 hours per design.

Corresponding to the reduction in clock rate, we expect a 3× speedup for our design using four hardware workers. We observed a speedup of 3.02× over a single worker using work stealing, which perfectly fits our model.

The software implementation used as a reference utilizes the Intel Cilk Plus [102] framework for implementation of the work stealing approach. During execution the process is bound to four CPU cores using the `taskset` utility. The 4.20× speedup observed in software is slightly above the expected speedup of 4× that could be reached using four CPU cores.

We use the software implementation to determine the scalability of work stealing with respect to our hardware implementation. Using all the available 16 CPU cores, we achieve a speedup of 16.50× compared to the single-threaded implementation. This shows the high potential of using work stealing for B&B applications. The comparison to software also shows that our deliberate restriction of work stealing onto direct neighboring workers has little or no negative impact in our use case in terms of scaling.

## 6.4 Chapter Conclusion

This chapter presented the extensions of our basic sequential branch-and-bound hardware design in order to allow parallelization of work over several hardware workers using work stealing. We transformed our existing hardware design to use a work stealing approach where several finite state machines (FSMs) work in parallel on different parts of the search tree and autonomously synchronize to obtain work packages from other workers when becoming idle. We described the required changes on the state stack to enable concurrent access to the work packages stored. Furthermore, this chapter discussed the fundamental principles to coordinate and synchronize the stealing process across several workers including the initialization and termination phases.

In comparison to the sequential design from Chapter 5, we achieved a speedup of about 3.02× using four hardware workers on one FPGA card. Our evaluation showed that speedups proportional to the number of workers can be expected if the

clock rate can be kept constant. The number of workers was bounded by synthesis times and achievable clock rates and should scale with upcoming technology.

The combination of our basic branch-and-bound hardware design with the work stealing extension enabled us to distribute the work to hardware workers. Another opportunity for acceleration is the fact that the organization of the search (i.e. the starting point and order of branching) and the computations within each level can heavily impact the runtime. This effect can be amplified for a particular problem instance such that a specific algorithm may be efficient to solve some instances but may be inefficient for others. All our previous designs use the static tree structure and static search path by incrementally guessing compatible values (see Section 5.3.1). However, a custom tailored, *instance-specific* tree structure for a given problem instance might lead to a significant speedup compared to a static, general solution. In the next chapter we investigate how instance-specific computing (ISC) can be applied to improve the performance for especially hard problem instances by utilizing instance-specific information.

# Chapter 7

# Instance-Specific Computing with Reconfigurable Hardware

The structure of a search tree representing the search space and the order in which branches are explored can heavily impact the time required to find a feasible solution. Depending on the concrete realization of the branch-and-bound (B&B) operations discussed in Section 5.2.1, different subtrees may contain different elements, subtrees may be arranged differently, or the order of live nodes on the pool may vary. These variations may cause a particularly long runtime for some cases, where the bounding function cannot exclude most of the unpromising parts of the search tree early on. This has also a direct impact on the state stack that is the key element for the parallelization with work stealing. In this chapter, we describe how different search trees can be dynamically constructed by utilizing instance-specific information to improve the search process with reconfigurable hardware.

In contrast to the extensions that we discussed in Chapter 6, e.g., the parallelization of branch-and-bound in hardware using work stealing, the concepts presented in this chapter are by their very nature tightly bound to a specific application and its instances. Therefore, we recall our case study, the AES key reconstruction, from Section 5.3. We start with concepts implied from our problem domain and describe methods of how the most promising branching order and tree structure can be chosen for a particular problem instance and how we automatically adapt our hardware design to any of those search trees, resulting in instance-specific hardware designs.

In the first Section 7.1, we give a motivation for using instance-specific computing and describe the three main methods to customize a design for a particular problem instance. Then, in Section 7.2, we apply these concepts to the branch-and-bound problem tackled in this thesis, the reconstruction of secret keys. We show heuristics of how valid and efficient search tree structures can be generated and how the most-promising branching order can be selected. In Section 7.3 we present our automated toolflow that we developed to generate instance-specific hardware designs utilizing the aforementioned methods. Finally, we evaluate the new designs in Section 7.4 and conclude this chapter in Section 7.5.

## 7.1 Motivation and General Description

Instance-specific computing (ISC) or sometimes instance-specific design (ISD) is possible in software and hardware. The general idea is to generate for each problem *instance* a distinct program heavily specialized and tailored to the concrete characteristics of that instance[129]. The generated result is usually only executed once. By

sacrificing the flexibility, the one-time execution is typically faster and the resulting binary or circuit can be smaller, because the generality of a solution is omitted. On the basis of this ideas, the instance-specific optimizations must provide enough improvements to overcome the overheads of analyzing and generating the single-use executable.

### 7.1.1 Methods for Customization

The three most important methods to customize for a particular problem instance are:

**Constant Folding and Constant Propagation** Constant folding is a well-known optimization performed by compilers [99, 283, 45, 241]. The idea is to detect constant expressions or input variables with values known at compile time and directly simplify them rather than perform additional computations at runtime. The required information to apply the simplifications originate from the compilation process itself, by resolving macros or system parameters or by the constraints implied by the input data. If, for example, the value of one input to an XOr gate is known, the gate can be replaced by an inverter or an wire. Constant propagation on the other hand is the process of gradually replacing known constants in expressions. In combination with constant folding, whole parts of the control flow of an application can be simplified or omitted if conditional branches can be determined to have only one possible outcome.

In the case of ISC even the actual input data is assumed to be constant. This opens a vector of optimizations beyond the transformations a compiler can usually apply. Using constant folding and constant propagation, the same output can be computed using less instructions or less logic resulting in a smaller executable or circuit.

**Custom-Precision Data Types and Functional Adaptation** A concrete functionality is usually implemented for the general case. This is especially important for functions operating on numbers. If the data types are not chosen carefully, overflows could occur, leading to wrong results and/or unintended behavior [68, 69]. However, if the actual input numbers for a specific problem instance are known, the data type and number of bits to represent the data can be of custom precision tailored to that input [128, 122, 205]. This optimization usually does not change the result, but allows area savings. However, in some cases the number of bits to represent data can be reduced to still achieve a certain quality of result. The result is guaranteed to be of a certain accuracy, but is not exact. The process of adding or removing parts of the functionality to adapt to a certain input data with or without allowing inaccuracies is called functional adaption or approximate computing.

**Architectural Adaption** In contrast to the other two methods, architectural adaption [15, 264, 252] tries to alter the way a result is computed on circuit level. One example is the introduction of custom instructions executed by a custom execution unit or the introduction of pipeline parallelism to increase throughput. The overall function and results typically stay the same.

### 7.1.2 Generation of Instance-Specific Designs

The generation of instance-specific designs in software is the just the recompilation of a program with hardcoded input data. Using the described methods and additional compiler optimizations (dead code elimination, etc.), a one-time executable can be generated within seconds. However, as CPUs have a general control and data path anyway, the more natural way to exploit instance-specific optimizations is the use of libraries that adapt the called function depending on runtime parameters. As an example, a certain problem instance can be analyzed to automatically configure an algorithm with good parameters at runtime [140, 157, 216].

Unlike for CPUs, integrated circuits are designed for a particular use, rather than for the general purpose. Instead of having a flexible control and data path, the specific required control path is unrolled spatially on the circuit. Application-specific integrated circuits (ASICs) offer a lot of benefits that are well aligned with the instance-specific computing principle [162]. However, they are custom-designed and hardwired for a particular application. The cutting edge technology for instance-specific computing are therefore FPGAs. FPGAs can also be customized to a particular application; but unlike ASICs, FPGAs can be reprogrammed after fabrication. Hence, for a given FPGA design the methods described above can be used to reduce the general design to perfectly fit only a particular instance of the problem. However, the generation of an instance-specific design (the actual hardware synthesis) for FPGAs needs a lot of time (in the order of hours to days) – in contrast to the compilation of seconds for software. As the resulting circuit is expected to be only a single-use design, the amortization for instance-specific designs for FPGAs is very challenging and only viable for problem instances that are guaranteed to be hard and have long execution times.

Branch-and-bound problems offer a perfect example to study instance-specific computing on FPGAs, because the search trees are highly irregular and therefore hard instances occur that dominate the overall runtime. A custom tailored, *instance-specific* tree structure for a given problem can lead to a significant speedup in comparison to a static solution. In contrast to existing work, we do not just use the methods described above to reduce a general application design down to a particular instance, but combine our general B&B hardware design with the work stealing extensions and instance-specific computing. In the next section, we apply the described methods to our case study, the secret key reconstruction, and describe how we adapt our hardware design to the instance-specific concepts automatically.

## 7.2 Instance-Specific Branch-and-Bound Search Trees

The exploration of the search tree that is spawned by a branch-and-bound (B&B) problem depends on the structure of the tree (which node or constraint is selected first) and the path that is followed during each branch-and-bound operation (the value that imposed for that node constraint). In the AES key reconstruction the structure of the search tree is determined by the selection of byte positions to guess values for and the path is determined by the order in which values are guessed in each level. The static allocation showed earlier in Table 5.1 of Section 5.3 on page 46 is one of many possible ways to select the byte positions to reconstruct a key schedule that gives a particular tree structure, while the sequential guessing of a compatible

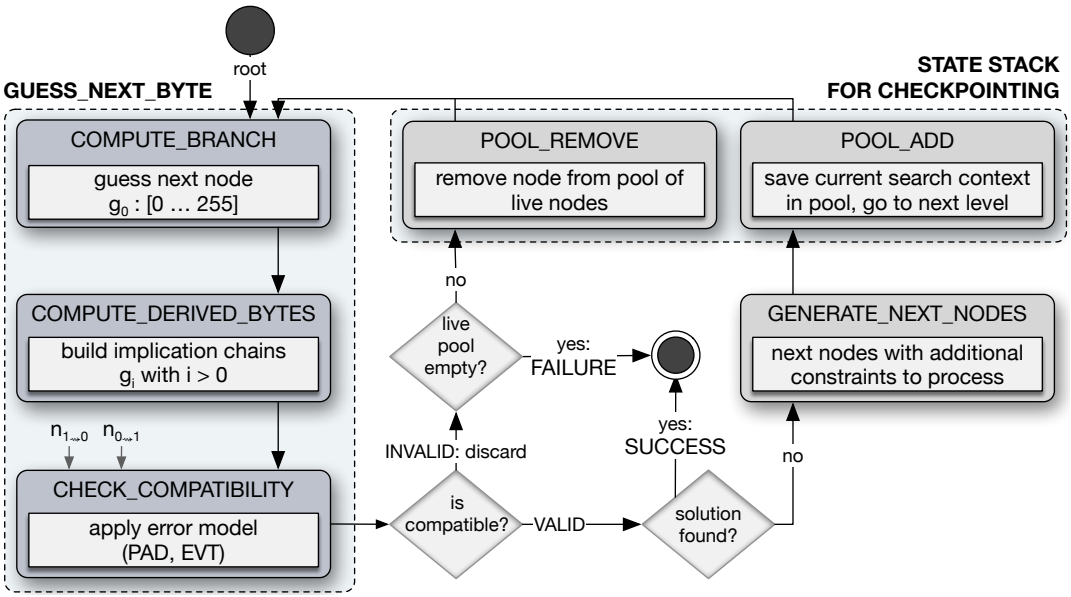



Figure 7.1: Finite state machine designed for secret key reconstruction with states highlighted that profit from instance-specific computing.

value is one of many possible ways to chose a value.

In the following, we present how the branching order can be improved by efficiently exploiting the information given by the actual problem instance. The branching order creates the path for how a search tree structure is explored. Then we describe how different valid and optimal tree structures for a search tree depending on the problem instance can be built and estimate the total number of possible tree structures. The selection of the most promising tree structure among all possibilities may effect the runtime dramatically. Therefore, we also describe how a tree structure can be chosen for a particular problem instance to solve it efficiently.

### 7.2.1 Instance-Specific Branching Order

Branch-and-bound search algorithms can traverse the created search trees in various orders (see Section 5.1.2), e.g. using a static order like in breadth-first search (BFS) or depth-first search (DFS) or some heuristical ordering like in best-first search (BeFS). Optimally, a heuristic should first branch to subtrees that promise the best solution or the highest chance to efficiently find a feasible solution. The state `COMPUTE_BRANCH` of the finite state machine (FSM) in Figure 7.1 selects and sets the next value and therefore determines the taken branch. In comparison to the static branch selection strategy from Chapter 5, in this section we introduce an optimized, instance-specific heuristic for the branching order. Instead of statically incrementing the guessed value for a byte from 0 to 255, we dynamically reorder the values individually on each position to visit the most promising branches of our search tree first.

The guessed values have different probabilities to decay to the value observed in the memory image. These probabilities can be computed in advance as the decayed value and assumptions about the decay probabilities per bit are known for a particular problem instance. In our error model, the error rates $r_{1\rightsquigarrow 0}$ and $r_{0\rightsquigarrow 1}$ determine the total fraction of bits that have flipped in one direction. Because in a key

| | from candidate byte 0x9C | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **flip type $\gamma$** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | **count** | **probability** |
| $0 \rightsquigarrow 0$ | | ✓ | ✓ | | | | | ✓ | 3 | $1 - p_{0\rightsquigarrow1}$ |
| $0 \rightsquigarrow 1$ | | | | | | | ✓ | | 1 | $p_{0\rightsquigarrow1}$ |
| $1 \rightsquigarrow 0$ | | | | ✓ | | ✓ | | | 2 | $p_{1\rightsquigarrow0}$ |
| $1 \rightsquigarrow 1$ | ✓ | | | | ✓ | | | | 2 | $1 - p_{1\rightsquigarrow0}$ |
| | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | |
| | to decayed byte 0x8A | | | | | | | | | |

Table 7.1: Calculation of the decay probability from a candidate byte 0x9C to the decayed 0x8A for all possible flip types.

schedule of $N$ bits without errors ones and zeros are expected to be equally frequent and represent half of the bits each, we can assess the probability of a 1 bit flipping towards a 0 bit (denoted with $p_{1\rightsquigarrow0}$) and the probability of a 0 bit flipping towards a 1 bit (denoted with $p_{0\rightsquigarrow1}$) as:

$$r_{1\rightsquigarrow0} \cdot N = \frac{1}{2} N \cdot p_{1\rightsquigarrow0} \qquad p_{1\rightsquigarrow0} = 2 \cdot r_{1\rightsquigarrow0} \tag{7.1}$$

$$r_{0\rightsquigarrow1} \cdot N = \frac{1}{2} N \cdot p_{0\rightsquigarrow1} \qquad p_{0\rightsquigarrow1} = 2 \cdot r_{0\rightsquigarrow1} \tag{7.2}$$

**Branching Heuristic Applied to Examples** We apply the branching heuristic to some example guessed values to emphasize the potential of the new branching order that exploits instance-specific information over a static incremental method.

In our example, the decayed byte value $d = \texttt{0x8A} = 10001010_2$ is received from memory and a promising value should be guessed that is compatible to the bounding function (error model). Assuming the probabilities $p_{0\rightsquigarrow1} = 0.002$ and $p_{1\rightsquigarrow0} = 0.598$[1] are given, we examine two different candidates:

1. $g_0 = \texttt{0x9C} = 10011100_2$. As shown in Table 7.1, for such a guessed value three bits would have stayed at 0, two bits would have flipped from 1 to 0, one bit would have flipped from 0 to 1 and two bits would have stayed at 1. We can assess the probability of 0x9C decaying to 0x8A as:

$$\begin{aligned}\texttt{probability\_of\_decay}(\texttt{0x9C}, \texttt{0x8A}) &= \prod_{\text{flip type } \gamma} \texttt{probability}(\gamma)^{\texttt{count}(\gamma)} \\ &= (1 - p_{0\rightsquigarrow1})^3 \cdot {p_{0\rightsquigarrow1}}^1 \cdot {p_{1\rightsquigarrow0}}^2 \cdot (1 - p_{1\rightsquigarrow0})^2 \\ &= 0.998^3 \cdot 0.002^1 \cdot 0.598^2 \cdot 0.402^2 \\ &= 1.149 \cdot 10^{-4}\end{aligned}$$

2. $g_0 = \texttt{0x63} = 01100011_2$. The second candidate is the inverted version of the first (0x9C XORed 0xFF). In contrast to the first candidate, three bits would have flipped from 0 to 1, which is highly unlikely. Accordingly the probability for this candidate to be a promising branch is very small:

$$\begin{aligned}\texttt{probability\_of\_decay}(\texttt{0x63}, \texttt{0x8A}) &= 0.998^2 \cdot 0.002^3 \cdot 0.598^2 \cdot 0.402^1 \\ &= 1.145 \cdot 10^{-9}\end{aligned}$$

[1] Note the probabilities are highly asymmetric, because the decay towards the ground state (here assumed to be 0) of the memory cell is dominant, see Section 3.2.

These calculations have to be performed for all the 16 byte positions of the tree structure represented in the search tree and for all 256 possible byte values that can be guessed. With a total of 4096 calculations, this can easily be done in software before starting the key reconstruction on the FPGA. We therefore sort all possible candidate values for each of the 16 byte positions on the basis of the calculated decay probabilities of that particular problem instance beforehand in software. When starting the execution of the FSM, we transfer the optimized order of candidates along with the decayed key schedule to the FPGA where it is stored in BRAM. The `COMPUTE_BRANCH` state of the FSM has to be changed accordingly to not just increment the value of the bytes but instead read the new value from the BRAM. Please note that with this instance-specific optimization implemented in BRAMs the design does not require to be synthesized again when the branching order is changed. We performed extensive tests and this implementation variant delivers equal performance to statically hard-coding the optimized branching order, but saves the effort of an additional synthesis.

### 7.2.2 Generating Valid and Optimal Search Tree Structures

During the branch-and-bound search process, each selected node to expand should infer as much knowledge as possible to tighten the search space by adding further constraints. To minimize the key reconstruction effort, each byte whose value is guessed should imply values for as many other bytes as possible to exploit the redundancy in the key schedule. One of those optimal tree structures that maximize the knowledge is shown in Table 5.1. Optimality is guaranteed if Equation 5.2 is fulfilled. This ensures that the tree structure is of minimal length and only compatible values for 16 byte positions denoted by $g_0$, $g \in \{0, 1, \ldots, 15\}$ have to be guessed to reach a feasible solution. Also, it is preferable to choose a tree structure that uses many non-linear substitutions (SBox operations), because these cause single-bit errors to multiply and therefore allow efficient pruning of the search tree. Tsow [269] proposed a method to construct tree structures that fulfill these criteria.

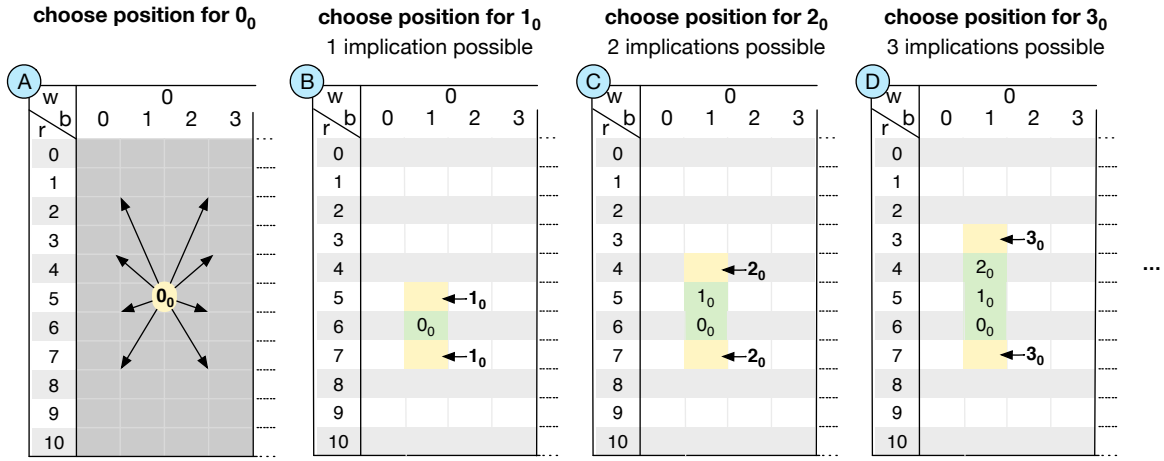



Figure 7.2: Generation of a reconstruction tree structure (first four byte positions). If the positions are chosen according to the construction rule, the number of implications is maximized.

The proposed procedure to gradually construct tree structures is sketched in Figure 7.2 for an example tree structure $P$ where possible choices are highlighted. The tree structure construction starts in (A) with an arbitrarily chosen position for $0_0$ from $P_{*,0}$ (the 44 bytes in any round of word 0, see addressing scheme in Section 3.3.1) for which a value is to be guessed at level 0 of the search tree. In the figure the second byte of round 6 $P_{6,0,1}$ has been chosen as the position for $0_0$. Starting from this byte position, the bytes to be guessed at positions 1 to 10 are chosen from the adjacent positions in the same column because this allows for inferring one additional byte using the complex expansion rule of the AES key expansion scheme (see Figure 3.4, page 21). There are at most two possible choices for each of those byte positions: either the one on top or the one below all previously chosen byte positions. In (B) the position for $1_0$ can either be $P_{5,0,1}$ or $P_{7,0,1}$. Assuming $P_{5,0,1}$ has been chosen, in (C) the adjacent positions for $2_0$ are $P_{4,0,1}$ or $P_{7,0,1}$. After the first 11 byte positions have been selected, the byte positions for the remaining levels $11 \ldots 15$ are fixed. All of the tree structures built upon this strategy result in the reconstruction of a complete round key for round 8 like the example shown in Table 5.1 of Section 5.3. The missing values of the key schedule can be derived from this complete round key for a final compatibility check.

**Number of Possible Tree Structures** Considering a byte from round key $k$ as the starting position, in each of the following 10 levels the decision is made if an upper or a lower byte position is chosen next. The decision for an upper byte position has to be made $k$-times, leading to $\binom{10}{k}$ possible sequences of decisions. Taking all round keys and all of the four first bytes from each key as possible starting positions into account leads to a total of $4 \cdot \sum_{k=0}^{10} \binom{10}{k} = 4096$ choices. So overall, 4096 different search tree structures can be constructed using Tsow's approach.

### 7.2.3 Selecting Instance-Specific Search Tree Structures

The selection of the correct tree structure for a particular instance may affect the runtime dramatically. On the basis of the tree structure selection heuristic of Tsow [269] we describe the basic principle and our extensions. His approach, which only considers the simplified perfect asymmetric decay (PAD) error model, is described in the following paragraph. Afterwards, we present our adaptation of this heuristic to our more realistic expected value as threshold (EVT) error model (see Section 3.2).

The PAD error model only accounts for flips into a single direction, e.g. $1 \rightsquigarrow 0$ *or* $0 \rightsquigarrow 1$. With this property, all bits with value 1 can immediately be assessed as not flipped and are therefore called *known bits* (see Lemma 3.1). During key reconstruction the search is bounded whenever a byte value is guessed that conflicts with a known bit. As early bounds imply a shorter runtime of the algorithm, the heuristic tries to maximize the number of known bits in the tree nodes near the root. The heuristic follows a greedy approach, placing byte positions with a high number of known bits close to the root of the search tree. For the first level, the heuristic selects the byte position $0_0$ with the most known bits from the first four bytes (see (A) in Figure 7.2) of all round keys in the decayed key schedule. While selecting either the upper or lower next byte position for levels $1 \ldots 10$, the heuristic chooses the byte position with the most known bits among these two, see Figure 7.2 (B) - (D).

Wang [281] extended this heuristic to the EVT error model by replacing the known bit criterion with a *dominant bit* criterion to encounter possible bit flips in both directions, $1 \rightsquigarrow 0$ *and* $0 \rightsquigarrow 1$. The dominant bit is chosen as

$$\texttt{DOMINANT\_BIT} = \begin{cases} 1 & \text{if } r_{1\rightsquigarrow 0} > r_{0\rightsquigarrow 1} \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

where $r_{1\rightsquigarrow 0}$ and $r_{0\rightsquigarrow 1}$ determine the total fraction of bits which have flipped towards one direction. As the probabilities of flips in one direction is orders of magnitude higher than flips in the other direction, the dominant bit is used as a replacement for the known bit. Using this adaption to the EVT error model, the heuristic of Tsow can be applied like for the PAD error model to generate and select the most promising tree structure for a particular problem instance.

## 7.3 Generation of Instance-Specific Hardware Designs

Each of the tree structures chosen by the heuristic corresponds to a distinct search tree instance on which the B&B reconstruction algorithm operates. Not only does the order of bytes whose values need to be guessed differ between different reconstruction tree structures but also the partitioning of all bytes of the key schedule into guessed and implied bytes. Depending on the chosen tree structure, the implied bytes may be derived from values of varying sets of other bytes, requiring the implementation of different implication chains (see Section 5.3.1) for all positions in the key schedule. Moreover, for other B&B problems with different algorithmic patterns (see Section 5.2.1), the different search trees may also differ in their structure: e.g. having a different number of children per node, local or global bounding functions, etc. This high variability makes the implementation of a universal hardware design challenging. Instead, we developed a generator for hardware designs to support different $k$-ary search trees. The generator translates a description of the general tree into code for a custom FSM suitable for the Maxeler toolflow [222]. The overall toolflow is sketched in Figure 7.3.

The first step (1) in automatically creating an instance-specific circuit is to analyze the problem instance by a custom software that generates a search tree. In our application we use Tsow's heuristic described in the last section to build the most promising reconstruction tree structure (2). In the AES key reconstruction many XOr operations are performed. If a concrete decayed key schedule is provided as an input, many of these operations can be removed or simplified to inverters. On the basis of the structure of the problem instance, our design generator creates domain-specific code (MaxJ) for each new decayed key schedule (3) and the Maxeler tools transform this code into a hardware design, which is further processed by FPGA vendor tools to generate an FPGA configuration (4). Additionally, the corresponding software for the host computer to transfer the required data and parameters is generated and compiled. The design generator builds fully custom instances of the superstate `GUESS_NEXT_BYTE` (including the states `COMPUTE_BRANCH`, `COMPUTE_DERIVED_BYTES` and `CHECK_COMPATIBILITY`, highlighted in Figure 7.1) for each node in the tree. The order in which children are visited is read from a ROM, allowing to modify the branching order without synthesizing a new design (bo). The number of hardware $N_w$ workers can be set using a parameter (ws).

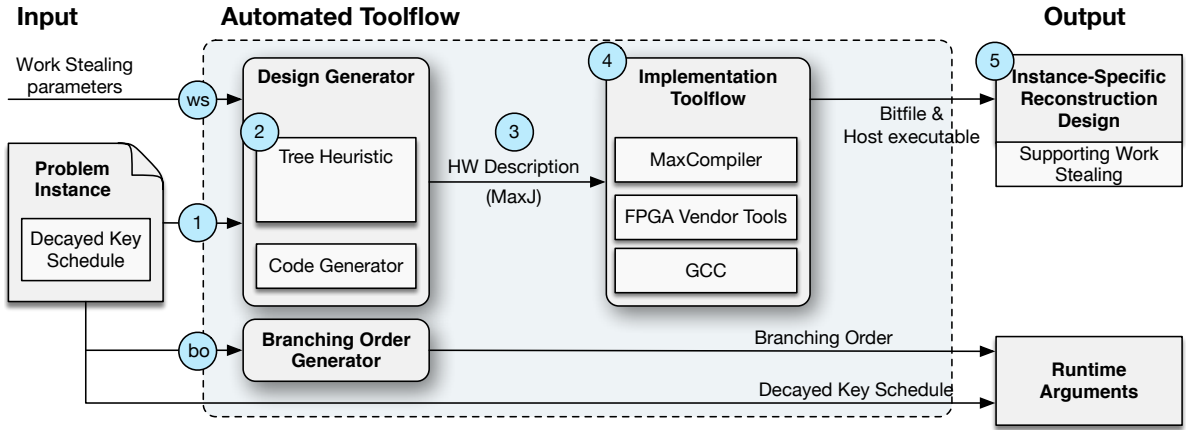



Figure 7.3: Toolflow of our instance-specific design approach.

In our application, the reconstruction of AES keys, the generator also contains application-specific logic. Although the tree structure specifies which bytes are implied in each level, it does not specify rules to imply these bytes. In order to determine possible implications, the design generator keeps track of the set of bytes that already have values assigned. Based on this information the generator automatically selects implication rules to infer as many bytes as possible from already known byte values. Depending on the complexity of the implication rules, multiple implications are combined and performed in a single clock cycle, or split into multiple clock cycles. If the *complex* expansion rule of the AES key expansion scheme (see Figure 3.4 in Section 3.3.2) is required to infer another byte value, a ROM access is used for the SBox operation and additional clock cycles are introduced to access data stored in the ROM following the principle described for a single FSM in Chapter 5.

## 7.4 Evaluation

In this section, we evaluate our instance-specific techniques developed in this chapter, namely the optimized branching order and the dynamic search tree structures. To measure the overall impact, we incrementally add those strategies to the work stealing design from the previous Chapter 6. The evaluation is performed using the same 512 key schedules with the highest error rates. We assess the impact on the total time required to do the same task in software and hardware.

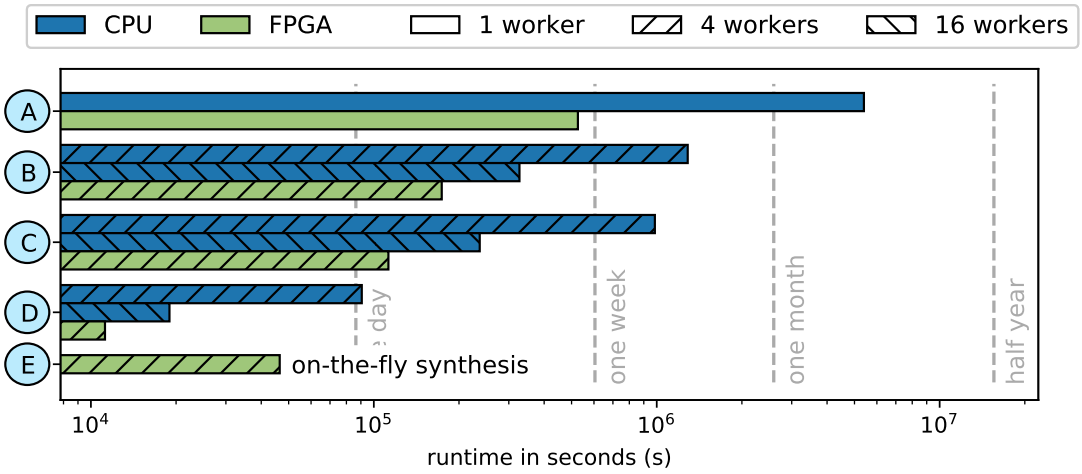
### 7.4.1 Results

The measured runtimes to reconstruct all 512 key schedules using work stealing and the different instance-specific techniques are shown in Table 7.2. The corresponding visualization of the data is depicted in Figure 7.4 and the incremental speedups by each technique are shown in Figure 7.5.

Our first optimization, reordering the branches based on the specific key schedule that is being reconstructed (C), leads to a 1.55× speedup in hardware on top of the improvements achieved by work stealing. The same optimization leads to a slightly lower speedup of 1.31× (4 workers) and 1.38× (16 workers) in software.

For the PAD error model it was already shown by Tsow [269] that using a heuristically chosen reconstruction tree structure can have a significant impact on the reconstruction time. In our measurements, the adaptation of this heuristic to the EVT error model shows an additional 10.04× speedup on our hardware design in (D). Combined with the aforementioned techniques (B) and (C) a total speedup of 46.95× compared to our hardware baseline implementation (A) is reached. A reconstruction of 10,000 key schedules using instance-specific hardware designs for all key schedules would take about 219,000 seconds, which corresponds to about 61 hours, instead of the four months needed when using the static hardware baseline implementation or over three years when using the static software baseline implementation. To validate these results we also implemented the adapted heuristic (D) in software and observed a slightly higher speedup of 10.86×. For 16 workers, the instance-specific solution even achieves a speedup of 12.49×, showing that the potential benefit of using the instance-specific approach increases for a higher number of workers.

Figure 7.4: Visualization of the reconstruction of 512 key schedules using work stealing and different instance-specific techniques (see Table 7.2).




| | Feature Set | Number of Workers | Sum | Average | Standard Deviation | Total Speedup |
|---|---|---|---|---|---|---|
| CPU | (A) Baseline Software | 1 | 5,399,100 | 10,545 | 65,056 | 1.00 |
| | (B) Parallelization using work stealing | 4 | 1,286,770 | 2,513 | 15,630 | 4.20 |
| | —— " —— | 16 | 327,164 | 639 | 3,988 | 16.50 |
| | (C) ISD: optimized branching order | 4 | 984,943 | 1,924 | 11,674 | 5.48 |
| | —— " —— | 16 | 236,768 | 462 | 2,787 | 22.80 |
| | (D) ISD: dynamic tree structure | 4 | 90,663 | 177 | 1,076 | 59.55 |
| | —— " —— | 16 | 18,952 | 37 | 217 | 284.88 |
| FPGA | (A) Baseline Hardware | 1 | 526,453 | 1,028 | 7,271 | 1.00 |
| | (B) Parallelization using work stealing | 4 | 173,806 | 340 | 2,414 | 3.02 |
| | (C) ISD: optimized branching order | 4 | 112,605 | 220 | 1,522 | 4.68 |
| | (D) ISD: dynamic tree structure | 4 | 11,214 | 22 | 142 | 46.95 |
| | (E) ISD: on-the-fly synthesis | 4 | 46.488 | 91 | 270 | 11.32 |

Runtimes in seconds (s)

Table 7.2: Reconstruction of 512 key schedules using work stealing and different instance-specific techniques.

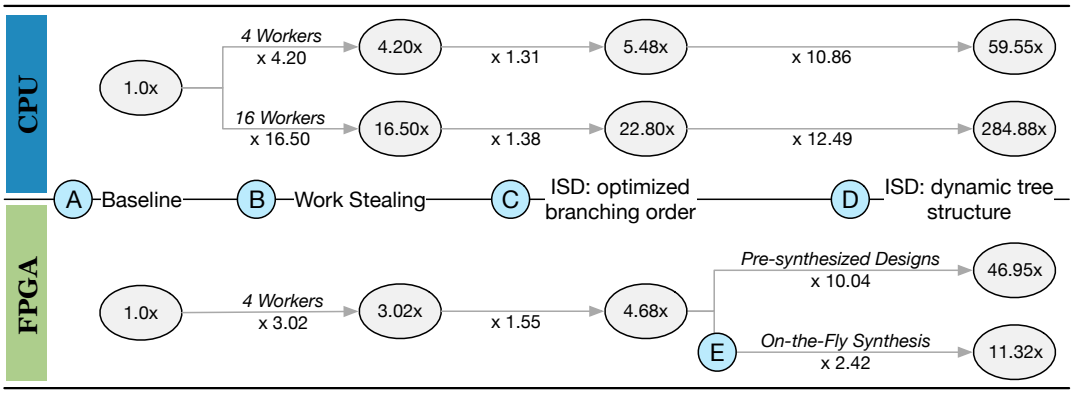



Figure 7.5: Visualization of speedups achieved by the different techniques in Table 7.2.

The presented hardware numbers (A)–(D) only include the runtime of the key reconstruction and do not take into account the synthesis times of instance-specific designs. Since the number of possible tree structures and therefore hardware designs is limited to 4096 (see Section 7.2.2) and synthesis of these designs can be parallelized on many machines, an attacker with sufficient resources may have precomputed and stored these designs in advance, making this a plausible scenario. In the next section, we evaluate whether dynamically generating these designs on-the-fly (OTF) for a particular problem instance can still lead to a significant speedup of our application.

### 7.4.2 On-the-Fly Hardware Synthesis

The first thing to note here is that a previously generated hardware design can be reused for different key schedules if the heuristic chooses the same tree structure for reconstruction. In fact, when evaluating the heuristically chosen tree structures for reconstruction of 10,000 key schedules only about 1,500 different tree structures were generated, as shown in Figure 7.6.

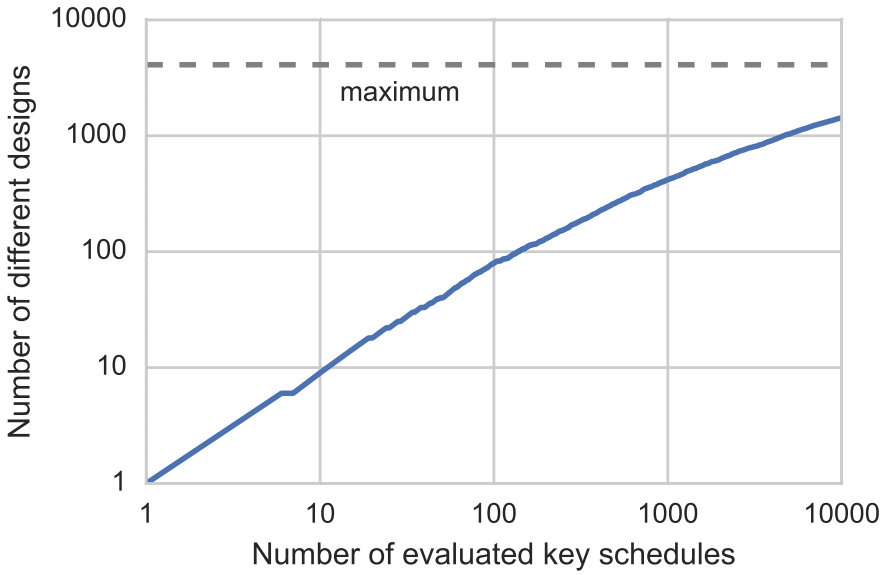



Figure 7.6: Re-use of designs for other instances.

**Generation Strategy** Synthesis of one custom hardware design takes about 2.5 hours on our target platform. As the syntheses of different designs are independent of each other, they can be parallelized depending on the number of CPU cores and amount of memory available. Our target system featuring a total of 12 CPU cores implies an amortized synthesis time of $^{2.5}/_{12}$ hours when running twelve syntheses in parallel. Since most of the key schedules can be reconstructed in significantly less time it is not reasonable to build an instance-specific design for every key schedule. We therefore propose an initial reconstruction of all key schedules on a general hardware design (e.g. the one following the static allocation in Table 5.1) and aborting those reconstructions that run longer than a certain threshold. For these hard key schedules an instance-specific design is generated on-the-fly and used for reconstruction. In the following section we evaluate how this threshold should be chosen for our application and what speedup is achievable using our proposed method.

**Amortization** We denote $p$ as the percentage of key schedules for which we build instance-specific designs on-the-fly. The threshold $t_{\text{thres}}$ after which we abort a key reconstruction is then chosen as the reconstruction time of the easiest from the $p$ hardest key schedules. Further we denote the reconstruction time for a key schedule on the general hardware design as $t_{\text{gen}}$ and on the instance-specific design as $t_{\text{IS}}$. $t_{\text{syn}}$ is the time required to synthesize an instance-specific design. For solving a single key schedule the required amortized time can then be assessed as:

$$\texttt{amortized\_reconstruction\_time} = \begin{cases} \frac{t_{\text{gen}}}{\#\text{FPGA}} & \text{if } t_{\text{gen}} < t_{\text{thres}} \\ \frac{t_{\text{thres}}+t_{\text{IS}}}{\#\text{FPGA}} + \frac{t_{\text{syn}}}{\#\text{CPU Cores}} & \text{otherwise.} \end{cases} \tag{7.4}$$

Figure 7.7 shows the calculated total reconstruction time for all 512 key schedules depending on the value chosen for $p$, based on our target system, performing 12 syntheses in parallel and using one FPGA. It shows that on-the-fly generation of hardware designs leads to a lower total reconstruction time if $p$ is below 27%. The threshold $t_{\text{thres}}$ in this case is 20s. The lowest reconstruction time is achieved by choosing $p$ as 7%, leading to $t_{\text{thres}}$ being 271s. In this case a speedup of 2.42× over using a general hardware design is achieved. Compared to our baseline hardware implementation, this corresponds to a total speedup of 11.32× in (E). This value does not account for possible re-use of previously generated designs. In the scenario of reconstructing 10,000 possible key schedules, this would allow even higher speedups.

Although for this evaluation we used posteriori knowledge about the reconstruction times of our key schedules, it shows that a significant speedup is possible by generating instance-specific hardware designs on-the-fly if the problems show a strong variation in difficulty between problem instances. Also it shows that a wide range of values for $p$ (0–26%) leads to a positive speedup.

### 7.4.3 Discussion and Practical Considerations

Overall the evaluation showed that each of our proposed techniques leads to a significant speedup over the baseline implementation, which was to the best of our knowledge the fastest known hardware implementation at the time of publication. Assuming pre-synthesized instance-specific designs we achieved a total speedup of 46.95× compared to a hardware baseline combining those techniques. Considering

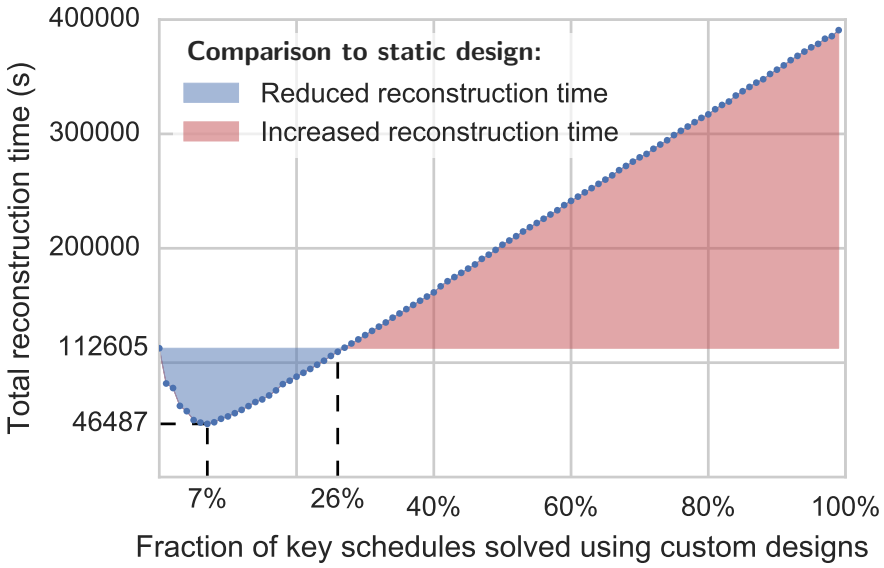



Figure 7.7: Reconstruction time using on-the-fly synthesis.

a dynamic on-the-fly generation of these designs still leads to a total speedup of 11.32×.

Even the comparison of a complete CPU node at full utilization with 16 workers versus one fully utilized FPGA card with only 4 workers gives the FPGA a performance advantage of 1.69×. This assumes that all the designs are pre-synthesized, which is a valid assumption for agencies or attackers that reconstruct secret keys on a daily basis.

In terms of power consumption, we collected the power usage of the FPGA with the Maxeler tool `maxtop -v`. The FPGA itself consumed on average 16.5 W while executing the key reconstruction. The host system is mainly used to trigger the FPGA and is idle afterwards. Exact power measurements of the host cannot be provided, because the Westmere-based processors do not support Intel's running average power limit (RAPL) interface [75]. In this prototypical setup, the host consists of powerful server CPUs, which are not required in a real scenario.

On the other hand, the Sandy Bridge-based microarchitecture of the cluster nodes where the software reference is computed supports the RAPL interface. The RAPL interface provides sensors that can be accessed by model-specific control registers to read the power consumption of different CPU components. The sensors are updated in intervals of about 1 ms. Besides sensors for the different power planes, there are sensors to measure the memory controller or the whole package. We use the tool Likwid Powermeter [266] in version 4.3.2 to read the RAPL sensors in order to get the power consumption. The CPU has a thermal design power (TDP) of 115 W per socket and we measured an average consumption for the whole package including the memory controllers of 95.4 W per socket during the key reconstruction using 16 workers. This gives the FPGA an advantage in energy efficiency of:

$$\frac{2 \cdot 95.4W \cdot 1.69\times}{16.5W} = 19.54\times \tag{7.5}$$

## 7.5 Chapter Conclusion

In this chapter, we studied how the branch-and-bound hardware design developed as a static variant in Chapter 5 and parallelized in Chapter 6 can be combined with instance-specific designs (ISDs). The general idea of ISD is to generate a distinct program heavily specialized and tailored to the concrete characteristics of a particular problem instance. Two important factors for the acceleration of a B&B algorithm are the organization of the tree structure spawning the search tree and the exploration of the search tree by branching to the most promising subtrees first. In this chapter we tackled both problems.

We designed and implemented strategies to create optimal tree structures and to follow most-promising subtrees with the help of heuristics. The result is an automated toolflow that analyzes a particular problem instance and generates the instance-specific hardware design. We evaluated both optimizations and also assessed an on-the-fly approach for the synthesis of instance-specific designs.

# Chapter 8

# Related Work

In this chapter, we outline and categorize related work in the areas covered in this thesis. We begin with side-channel and cold-boot attacks in Section 8.1 and discuss a set of related problems within the cryptography domain. Afterwards, we focus on the main aspect of this work, the branch-and-bound (B&B) algorithmic pattern. In Section 8.2, we cover related work using the general branch-and-bound principle in soft- and hardware. Then we describe related work in the parallelization of B&B in Section 8.2.1 and instance-specific computing in Section 8.2.2.

## 8.1 Side-Channel and Cold-Boot Attacks

Side-channel attacks (SCA) are a branch of the information security and cryptography domain that were covered in Chapter 3. The main goal of an attacker is to obtain sensitive key-dependent data exploiting the physical implementation of a cryptosystem as illustrated in Figure 8.1. SCAs have been exploited over decades [154, 153] to break hardware and software security mechanisms and to gain access to sensitive data. In this section, we outline related work regarding side-channel attacks and put them in perspective to the attack vector presented in Section 3.1.3.

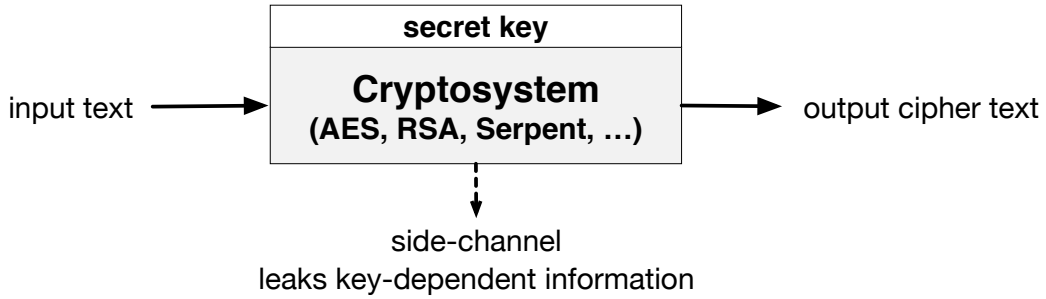



Figure 8.1: Illustration of a cryptosystem that leaks key-dependent information over a side-channel.

### 8.1.1 Acquisition of Sensitive Data

Cryptographic algorithms need to be implemented on real electronic devices and hold sensitive data such as secret key material during execution. However, all electronic devices leak information — sometimes in unusual and unexpected ways. The most prominent attacks on such side-channels are:

**Timing Analysis Attacks** Physical implementations perform computations in non-constant time. These timing variations can be measured and combined with statistical analysis to gain knowledge about secret parameters of a cryptographic algorithm. Timing attacks were first proposed by Kocher [154] on RSA keys. Similar attacks exist for other ciphers [133, 31, 35, 219]. Most recent variants even exploit the microarchitecture of processors with timing attacks to extract information about private data using the speculative execution of instructions [152] or a race condition that can arise between instruction execution and privilege checking [184].

**Power Analysis Attacks** Similar to timing attacks, the power consumption of a device may provide information about the involved computations. Power analysis attacks have been demonstrated for most prominent symmetric and asymmetric key ciphers and are especially critical for smart cards [190, 198, 199].

**Fault attacks** An attacker tries to induce computational faults on the actual device performing the cryptographic computation. This could, for instance, be done physically by precise voltage manipulation [19, 20], optically illuminating specific transistors [256] or exposing the device to high temperatures [136].

**Main Memory Attacks** Some decades ago researchers pointed out the critical decay time of RAM, the remanence effect [183, 123, 255] as introduced in Section 3.1.2. The fundamental work of Halderman et al. [126] is regarded as a milestone and initial impulse for further efforts and investigations. Halderman et al. impressively reproduced and categorized the decay patterns of main memory at different temperatures and provided concrete attack models and vectors characterized as cold-boot attacks. Apart from the remanence effect of main memory, recently Kim et al. [149] were able to bypass the isolation between memory cells and change cell values without permission using the row hammering technique. Kwong et al. [164] build on this technique to also read memory contents.

Using these side-channel attacks, an attacker can measure such unintended sources of leakage to either gain a full dump of the main memory or to guess portions of sensitive data that represent the secret key material. The intercepted data is typically not an exact copy and contains bit errors caused by the attacking method.

If the attacker gets a full memory dump, like in the case of cold-boot attacks, the secret key material needs to be identified. Related work regarding this task is explained in Section 8.1.2. If the attacker already gets portions of the secret key material, the identification can be skipped and reconstruction techniques are required, which are explained afterwards in Section 8.1.3.

### 8.1.2 Search and Extraction of Secret Key Material

The first search for cryptographic keys in streams of data took place in file systems. Shamir et al. [245] describe two concepts for searching for RSA keys. On the one hand, they used the mathematical relationship between the public and private keys for the direct verification of key candidates. On the other hand, they describe the general structure of cryptographic keys in the file system and the high entropy compared to normal data or program code. Klein [150] introduced the search of RSA keys in main memory and provided a free implementation. In contrast to Shamirs

et al. he exploits the standard storage format for private keys and SSL certificates in main memory to identify the secret key material. Kaplan et al. [143] describe the segmentation of the memory into relevant areas. They show that segmentation can significantly reduce the search space and thus speed up the search for key candidates.

### 8.1.3 Reconstruction of Secret Keys

Halderman et al. [126] proposed one of the first algorithms for the correction of bit errors in AES keys. Their method does not use the full structure of the key schedule and has many algorithmic parts using brute force to reconstruct a key. Tsow [269] took up the basic work of Halderman et al. and uses the described guessing and pruning phases to reduce the search space tremendously. Kamal et al. [141] improved the recovery of Tsow by reformulating the method into a boolean SAT problem and solving it with an off-the-shelf SAT solver (MaxSAT). In contrast to our work, both methods only support the simple PAD error model. Albrecht et al. [16] model the reconstruction problem with set of non-linear algebraic equations with noise. Similar to our work, their model supports also asymmetric decay. However, they apply the method to the symmetric ciphers Twofish and Serpent and not to AES.

Similar reconstruction methods exist also for public-key cryptosystems. Heninger et al. [131] presented a general method to recover secret RSA keys given a random fraction of the bits of the private key. Lee et al. [171] applied the same ideas to private RSA keys obtained from cold-boot attacks. Henecka et al. [130] used also Hamming distances with a user-defined threshold to prune key candidates that are unlikely to be the searched key. Paterson et al. [221] further improved this method using an coding-theoretic approach.

FPGAs have been used for many cryptographic applications. Several loosely related articles discuss the implementation of the AES algorithm on FPGAs to mainly explore performance versus area trade-offs [290, 174, 60, 113]. Eisenbach et al. [95] present a good survey on this topic. Other researchers [46, 48, 148] target FPGAs for fault attacks to weaken secure AES implementations on the device. Another related usage of FPGAs in this area is brute-force password testing and breaking. John the Ripper [178] is one of the most popular tools with an open-source FPGA support to break password hashes with a dictionary or brute-force attack. COPACOBANA [159, 117] is an FPGA-based code breaker optimized for cracking DES keys using an exhaustive search. Mencer et al. [197] presented a key search engine for the stream cipher RC4 consisting of 512 FPGAs. Other projects exist tackling different ciphers or specific aspects such as performance to cost ratios [230, 23, 86, 85].

These projects solve a similar problem that is presented in the case study of this thesis. The specialization of FPGAs is used to achieve very high processing rates comparable to ASICs with the full flexibility offered by reconfiguration to tackle different ciphers with the same device. Similar to our work, the related articles make use of the property that all ciphers can be efficiently implemented in hardware by design. However, all the outlined architectures work algorithmically with the principle of exhaustive search and achieve the extremely high processing rates by the parallel testing of millions of key candidates on hundreds of devices and instances.

Each individual evaluation per key candidate is comparably easy to the branch-and-bound problem that we solve. To the best of our knowledge, we published the first articles that use FPGAs for accelerating the search and reconstruction of secret AES keys containing bit errors as obtained through cold-boot attacks.

## 8.2 Branch-and-Bound in Soft- and Hardware

The basic principles and general idea of the branch-and-bound algorithmic pattern have been described in Chapter 5. In this section we outline and categorize related work including efforts to parallelize the computation and use instance-specific optimizations. We present similar approaches to ours using these techniques and highlight differences to our work.

Lawler et al. [92] give an early survey on the essential methods and features of the branch-and-bound approach. They outline the general relationship [213] to several other applications [169, 14], including integer linear programming [100], non-linear programming [121] and various $\mathcal{NP}$-hard assignment problems [107, 185]. These properties make B&B a universal tool for tackling these important problems. Boyd et al. [40] show for many of these problems how they can be formulated with the B&B paradigm and provide useful examples. Further examples can be found by the excellent article from Clausen [65]. A more recent survey by Morrison et al. [210] presents further advantages in the algorithmic design for the main operations (search strategies, branching strategies and bounding rules).

### 8.2.1 Parallelization and Work Stealing

Branch-and-bound problems are inherently difficult and only small problem instances can be solved with one single worker in a reasonable amount of time. Consequently, the parallelization of the branch-and-bound operations was a relevant topic early on [279, 161, 81, 82].

**Early Experiments** First parallelization strategies were designs for multiprocessor systems [279, 161] consisting of a couple of nodes. The first implementations were dealing with anomalies [166, 177] due to insufficiently understood architectural requirements, scalability properties or data structures. Gendron et al. [105] presented a detailed survey of the literature on early parallel branch-and-bound algorithms and architectures. They analyzed existing theoretical and practical work and proposed a classification system for branch-and-bound algorithms similar to the one used in Section 6.1. Parallelism type 1 corresponds to a form of data-level parallelism, where operations on subproblems (e.g. inferring knowledge and bounding the selected node) are executed in parallel, while the exploration of the search tree is sequential. Parallelism type 2 corresponds to task-level parallelism, where the search tree is explored in parallel. Finally, parallelism type 3 describes the processing of completely different search trees in parallel. The design and implementation developed in this thesis uses the parallelism of types 1 and 2 to get the reached efficiency to process one single problem instance.

Another highlight of the survey by Gendron et al. [105] is the synthesis and review of the previous parallel systems using one centralized pool of live nodes implemented as a master-slave paradigm or distributed using multiple pools. The synchronization can be either synchronous or asynchronous in both cases.

**Multiprocessor Systems** Multiprocessor systems were opening up more and more areas for new application thanks to their increasing performance. Over decades researchers presented parallel branch-and-bound algorithms for different domains [93, 139, 161]. Nowadays, the high performance is closely linked to the possibility to parallelize the problem. This mainly includes the division of the overall problem into subtasks, which can be processed on different processors simultaneously. Each subproblem has to be large enough that it requires a relatively long execution time, ideally without further communication. Otherwise, multiprocessor systems cannot be used efficiently because many process changes entail high administrative costs. An over-distribution can also be problematic because the administrative overhead increases even then.

A key classification of multiprocessor systems for branch-and-bound algorithms is the type of memory. It is either shared, where all workers can access the memory of others, or it can be distributed, where each worker has its own dedicated memory that is not accessible for others. Branch-and-bound algorithms using distributed systems typically have a master-slave coordination scheme [134, 29, 94, 74, 39]. In shared memory systems the global variables, error bounds, intermediate solutions, sub-solutions and subtrees are exposed to all workers and branch-and-bound algorithms [98, 70, 202] use conditions and locks to cooperate.

The actual implementation of the proposed algorithms can be either low level, such as POSIX threads, or high level using APIs or frameworks such as OpenMP [73], MPI [116], TBB [234], Cilk [102], OpenCL [259] or OpenACC [284]. Both strategies have been studied for branch-and-bound problems [176]. Talbi [262] provides a good overview. Casado et al. [51], for instance, use low-level POSIX threads. The parallelization mechanisms require a lot of coding and are deeply entangled with the application layer. The authors propose two parallelization schemes: 1) one global pool of live nodes guarded by mutex synchronization accesses, which causes significant overheads and 2) local pools in each thread, which perform better. The number of workers is not static. If a condition is met, a thread can create a new one and migrate parts of his pool. However, the authors observe that this dynamic load balancing can cause dramatic performance losses in parallel B&B. Evtushenko et al. [98] therefore use another strategy in which each thread performs a number of steps on its local pool and then migrates parts of his pool to a shared pool. This approach seems to perform better. Mezmaz et al. [201] also use low-level threads combined with grid-computing to solve the Flow-Shop scheduling problem [156] with branch-and-bound. The load balancing is achieved with scale idle time stealing, which is a similar form of work stealing used in our approach. In contrast to our work only one single pool to store the live nodes is used. The evaluation in the articles shows a big mismatch between the achieved and ideal speedup in terms of scaling. A recent survey on load balancing using work stealing for multi- and many-core systems systems is outlined by Yang et al. [287].

On the high-level strategies, Barreto et al. [29] provide a comparison between a serial C++ and parallel OpenMP and MPI implementations. While the sequential implementation seems to work well, the OpenMP variant only reaches a speedup of 2.1× with 100 processors and MPI a speedup of 9.1×. Both unpromising results can be attributed to the design choices of the authors or limitations of the used tools (for instance, the used ILP was not thread-safe). Dimopoulos et al. [83] describe a general strategy to translate a sequential B&B algorithm into a parallel one using a hybrid

model called MPI-OpenMP [229]. While changing the code into the requested structure seems promising, the experimental results are limited in terms of performance and further investigations are required. Dorta et al. [91] propose a similar idea providing skeleton implementations that need to be adapted to the required problem. With one high-level specification using C++ templates, the tools generate two parallel versions: one with message passing over MPI and one with shared memory with OpenMP. In a second article [90] the same authors also present experimental results for the approach that show a good scaling for up to 24 processors. For larger numbers the algorithms are not scalable.

Recent research proposes generic frameworks [132, 24] that are in the spirit of skeletons for certain types of optimization problems, including tree searches such as branch-and-bound, instead of writing everything from scratch. For instance, ZRAM [42] is one relevant framework, but many more such parallel codes exist [70, 191, 87]. An extensive survey on this topic is given by Avis et al. [25].

**Systems with Instruction-Programmable Accelerators** GPUs have emerged as an efficient way for massively parallel computations. The GPU architecture is optimized for a SIMD-based execution model. Hence, the B&B algorithmic pattern is also not the typical class of directly suitable problems. Plenty of related articles address specific branch-and-bound applications (e.g. Knapsack [38, 167], FlowShop [57, 194, 193], Traveling Salesman [49], etc. [37, 111, 22]) or various optimization aspects such as thread divergence [57], caching [204] or memory [249, 30]. The main challenge to utilize the GPU architecture is to transform the irregular workload associated to the unpredictable tree traversal into a regular, data-parallel one that is easier to schedule and more likely to lead to a balanced execution. A good overview on recent efforts on GPUs is provided by Boyer et al. [41].

Regarding a multi-GPU setup a few investigations also exist [179, 53]. Chakroun et al. [55, 56, 54] use a master-slave approach similar to the ones discussed for CPUs. While the focus is on the feasibility of the approach, the reported speedups and scalability seems limited. Similar to our work, Vu et al. [278, 277] use a work stealing approach instead to solve the balancing and scaling issues. To avoid clustering of work packages the stealing is performed using randomization to select a victim. Gmys et al. [110, 112] also use a work stealing approach on GPUs. Instead of a simple queue/dequeue pool of work packages they use a custom integer-vector-matrix data structure [200] that better reflects the proposed hierarchical work stealing strategy. Lima et al. [180] present runtime task scheduling for multi-GPUs based on work stealing on top of Intel's Cilk Plus framework. By overlapping communication and computation, they try to hide overheads and utilize up to 20 GPUs. Navarro et al. [214] propose strategies to dynamically resize work packages for multi-CPU/GPU architectures to prevent underutilization due to too small or too large chunks for different types of workers. Elangovan et al. [96] use work stealing in a parallel execution model based on OpenCL to distribute tasks across up to 4 GPUs to achieve efficient utilization. Kumar et al. [160] implement work stealing between ARM and DSP cores, preferring stealing victims of the same processor type. Guo et al. [120] implement locality-aware work stealing, preferring victims that share the same L2 cache. Min et al. [203] extend this concept to a hierarchical model reflecting the whole memory topology.

The B&B algorithmic pattern has also been studied on the Intel Xeon Phi many-core processor [175, 192]. Melab et al. provide an extensive study [195] of big branch-and-bound problems on many- versus multi-core processors and most recently a study on many-core versus GPU coprocessors [196].

**Systems with Reconfigurable Accelerators** Branch-and-bound search algorithms are often used in the FPGA design synthesis process [260, 258, 104, 146]. For the actual application logic, the branch-and-bound pattern has also been studied. Shimai et al. [248] present a solver for mixed integer quadratic programming used in a real-time, low-power environment for robot control. In contrast to our work, they use a master-slave architecture, where the master (called the sequence control module) statically spawns the slaves (the quadratic solver core) without dynamic work balancing. Each solver core is equivalent on all search tree levels. Bakos et al. [27] map the $\mathcal{NP}$-hard median-breakpoint problem onto an FPGA. The algorithm is extracted from GRAPPA [209], an open-source branch-and-bound algorithm for phylogenetic inference using gene order data. Similar to our work, the authors present a design based on finite state machine (FSM) and use fine- and coarse-grained parallelism of the FPGA. However, each solver core works independently on his own search tree (single median computation). The cores can only communicate with each other to query or exchange the current bound value.

Kestur et al. [147] use work stealing on FPGAs to solve the problem of matrix-vector multiplication, but do not describe their architecture in detail. Ramanathan et al. [232] start with an OpenCL work stealing implementation for GPUs and synthesize the code with the help of Intel's OpenCL SDK for an FPGA. The authors focus on the synchronization of work items with the help of OpenCL's atomic operation instead of locks or mutexes. Despite performance and resource utilization disadvantages, the atomic operations seem to simplify a kernel design. The work is evaluated for $k$-means clustering, which can also be formulated as a B&B problem. In contrast to this approach, which uses high-level synthesis (HLS), we use a hardware implementation based on lower-level finite state machine (FSM). FSMs give us more control on optimizations, serve as a natural level of abstraction for B&B problems, and are highly suitable for parallelization with work stealing. Yan et al. [286] use also OpenCL for FPGAs to solve the $k$-means clustering problem. They try to overcome the performance limitations of Ramanathan et al. by pipelining the computation. Due to syntax limitations of the Intel FPGA OpenCL tool flow the authors were not able to implement a pipelined work stealing strategy and used a static work distribution approach instead. Chen et al. [59] propose a native work stealing support for FPGAs similar to ours without HLS, which is more efficient than the OpenCL variant in terms of resource usage and shows an equivalent scalability as our approach. In contrast to our work, idle workers can steal work packages from random victims and send the result back to the victim after computation. Sbîrlea et al. [242] present an extension for the Habanero-Rice runtime system [52] that can map an image-processing pipeline onto heterogeneous components including an FPGA. The main focus of this work is the cross-device scheduling and stealing of tasks based on the affinity. Shen et al. [247, 135] present a work stealing approach on an FPGA for matrix multiplication. In contrast to our work, in both cases the tackled applications are highly regular and follow a streaming pattern.

Several other articles [292, 238, 272, 88] tackle load balancing with/without work stealing in a heterogeneous environment, but none of them instantiates several hardware workers on one device. The logic for cooperation is outside the FPGA.

### 8.2.2 Instance-Specific Computing

In software, utilizing instance- or problem-specific information to improve the performance of branch-and-bound problems has been preliminary studied. For instance, Morén [208] analyzes the structure of a manpower planning problem to improve the bound and the branch operation. The work is evaluated for CPUs with off-the-shelf ILP solvers.

Initial work on instance-specific computing for reconfigurable hardware tackles the B&B algorithm for the propositional satisfiability problem (SAT) [289, 127, 142]. FPGAs are suitable for solving SAT problems because the computations (evaluation of clauses) are highly parallelizable, similar to the solvers outlined in the previous section. Nonetheless, it would be completely impractical to fabricate an application-specific integrated circuit (ASIC) for each formula, due to high development costs and inflexibility. Zhong et al. [291] present an implication circuit with a serial chain of FSMs. Each FSM corresponds to a variable in the SAT formula and the assignments of variables are tried in a fixed order. First, an FSM tries to assign 0 and a deduction logic evaluates the result. If it is 1 (true), the solution is found; if it is 0 (false), the complement assignment is tried; if it returns $x$ (undetermined), the next FSM is activated. If the formula is unsatisfied for all assignments, the values are reset and the previous FSM is activated (backtracking). The backtracking is usually guided by a cost function, which reduces the search effort by activating the most promising FSM when the formula is unsatisfiable with the current assignment. However, long synthesis times of several hours restrict the scope of SAT problems for which an FPGA solution is overall faster than a software-based solution. Hence, following works by Skliarova and Ferrari [254] and Davis et al. [76] avoid instance-specific placement & routing and move towards HW/SW-codesign approaches. The HW circuit is pre-synthesized and optimized only once and can then be reused for different problem instances using dynamic reconfiguration. Rashid et al. [233] give a good overview of the development process to generate instance-specific circuits for SAT problems. Skliarova et al. [253] present a survey of systems using reconfigurable hardware to solve similar problems.

Our application uses a similar implication circuit as described for SAT solvers or covering problems [224, 223]; but in contrast, our FSM corresponds to the entire problem instance and not just a variable in the SAT formula. Additionally, the order of variables for which different values are tried out is completely instance-specific. Each variable in our problem has 256 possible values (instead of just 0 and 1), which results in different $k$-ary search trees. For hardware-based SAT solvers it is necessary to utilize large amounts of off-chip memory to scale to real-world problem instances, whereas our problem is computation bound. The deduction of a SAT formula does not require any error model and a variable assignment can be tested locally and quickly. In contrast, we use a complex error model for pruning (see Section 5.4.5), which requires global state information and considerable computation effort.

Grigoras [115] presents an instance-specific tuning approach for the sparse matrix-vector multiplication problem. The developed framework can utilize the problem dimension and sparsity pattern to accelerate the computation. All possible configurations are pre-synthesized into a library. In contrast to our work, the tackled problem can be mapped to a data flow architecture, similar to other literature [244, 280] that tackles graph problems. Both approaches utilize the knowledge on the graph structure that is associate with a concrete problem instance. Koester et al. [155] analyze the assembler code of an application in order to instantiate a very long instruction word (VLIW) processor core, which is specialized exactly for these instructions.

Kašík [144] presents an instance-specific approach for solving the *Eternity II* puzzle with backtracking on an FPGA. The specific puzzle problem is encoded into the FPGA and the search for matching candidates is performed only by a single worker. Malakonakis and Dollas [187] examine the same problem but use an exhaustive depth-first search with up to 22 workers on one FPGA card. All workers are initialized with a static configuration and search completely independent from each other (work sharing). In contrast, we start the computation with one worker and the distribution and balancing is performed autonomously using work stealing without the interception of a host.

## 8.3 Chapter Conclusion

In this chapter, we presented an overview of the related articles in the areas covered in this thesis. While various branch-and-bound algorithms, work stealing and instance-specific computing have been studied on FPGA in isolation, to the best of our knowledge related work that covers the combination of these techniques does not exist.

# Chapter 9

# Conclusion

In this last chapter, we briefly summarize the topics and results covered in this thesis and outline possible directions for future work.

## 9.1 Summary

In this thesis, we have studied how the branch-and-bound (B&B) algorithmic pattern can be efficiently implemented in reconfigurable hardware. B&B is highly relevant because it is the most commonly used algorithmic pattern to solve combinatorial optimization or planning problems. The search space is represented as a tree and infeasible solutions are eliminated early by pruning subtrees which cannot lead to a feasible or optimal solution. On the other hand, FPGAs have been proven to be highly efficient in terms of chip area, power consumption and performance for a wide range of applications and application domains. However, branch-and-bound algorithms are not the typical class of problems that have been tackled with FPGAs, because the computation is control- and not data-driven. In this thesis, we fill the gap of the insufficiently understood branch-and-bound algorithms for reconfigurable hardware.

We have systematically designed and implemented a high-performance B&B implementation on FPGAs. First, we identified general elements of B&B algorithms and described their translation into a finite state machine (FSM). We developed an FSM architecture that uses highly optimized combinational datapaths for the performance-critical higher levels of the search tree and more resource-efficient pipelined ones for the less frequent and more complex lower levels. Then we extended the FSM to allow for multiple hardware workers that autonomously share and balance the computation using work stealing. Our evaluation showed that speedups proportional to the number of workers can be expected if the clock rate can be kept constant. The number of workers was bounded by synthesis times and achievable clock rates and should scale with more modern technology. Using this design we further explored the advantages of instance-specific computing on reconfigurable hardware by generating designs that are custom tailored to a specific problem instance. Using instance-specific designs we achieved improvements in performance and energy efficiency, although in this case all instance-specific hardware designs had to be pre-synthesized. Finally, we also evaluated an on-the-fly approach for instance-specific computing by generating and synthesizing custom hardware designs on demand at runtime. We showed that even on-the-fly generated hardware designs can amortize the synthesis times for particularly hard problem instances and lead to speedups over a non instance-specific approach.

## 9.2 Outlook

We see a number of possible future directions to extend the topics covered in this thesis, which we briefly outline:

- We instantiated all hardware workers for parallelization on one FPGA card and achieved near linear speedups. On the other hand, the target Maxeler platform has up to four FPGA cards, connected through a dedicated interconnect (see Section 2.3 and Section 5.5.1). This multi-FPGA architecture is also getting more and more attention in academia [106, 268] and industry [228, 263]. An extension of our work stealing concepts using multiple FPGAs and custom circuit- or packet-switched interconnects for different levels of the search tree and other network topologies are very interesting directions.
- Our work has demonstrated the feasibility and overall conditions of efficient branch-and-bound search on FPGAs using work stealing and instance-specific designs. We used a concrete case study to show the required steps on an interesting and yet relevant example. We described and explained different variants of branch-and-bound operations and it would be an interesting direction to analyze to which degree the presented techniques complement each other for different workloads.
- Our FSM-based architecture enables a clear, modular design of the different branch-and-bound operations and the parallelization mechanism. Separating them additionally into a library could simplify and accelerate the coding effort for new workloads, similar to the related work described in Chapter 8.

## Supplemental Material

All artifacts (source codes, supporting scripts and documentation) to reproduce the presented results are available online on github.

# List of Tables

# Listings

# List of Figures

# Acronyms

$\mathcal{NP}$ class of problems that can only be solved in non-deterministic polynomial time. 40, 43, 94, 97

**3DES** DES applied three times. 19

**AES** advanced encryption standard. 11, 12, 19, 20, 21, 23, 24, 25, 26, 27, 28, 30, 37, 45, 47, 48, 49, 50, 56, 62, 63, 77, 79, 83, 84, 85, 93, 94, 107, 109

**AES-256** AES with key size of 256 bits. 19, 30, 33, 34, 103

**AES-192** AES with key size of 192 bits. 19

**AES-128** AES with key size of 128 bits. 18, 19, 20, 21, 22, 27, 28, 30, 32, 33, 34, 45, 46, 48, 49, 57, 103, 105, 107, 108, 114, 115

**API** application programming interface. 6, 9, 95

**ASIC** application-specific integrated circuit. 79, 93, 98

**B&B** branch-and-bound. vii, ix, x, 1, 2, 4, 11, 35, 37, 38, 40, 41, 42, 43, 44, 45, 48, 49, 50, 53, 55, 62, 63, 65, 66, 67, 74, 75, 77, 79, 82, 84, 90, 91, 94, 95, 96, 97, 98, 99, 101, 102, 105, 107

**BeFS** best-first search. 39, 80

**BFS** breadth-first search. 39, 80

**BOS** bottom-of-stack. 68, 69, 70

**BRAM** block RAM. 4, 31, 33, 51, 52, 53, 55, 82, 107

**CBA** cold-boot attack. 11, 12, 15, 62, 92

**CPU** central processing unit. x, 2, 5, 6, 7, 27, 33, 34, 35, 44, 55, 56, 59, 60, 61, 62, 66, 74, 79, 88, 89, 96, 98, 111

**DDR** double data rate (SDRAM). 4, 56

**DES** data encryption standard. 19, 93, 109

**DFE** data flow engine. 6, 7

**DFS** depth-first search. 39, 43, 46, 53, 80

**DMA** direct memory access. 7

**DRAM** dynamic random access memory. 12, 13, 15, 16, 107

**DSP** digital signal processor. 4, 33, 55, 107

**ECC** elliptic-curve cryptography. 12

**EVT** expected value as threshold. 17, 18, 19, 25, 29, 53, 54, 56, 58, 60, 61, 62, 71, 83, 84, 86, 103, 105, 108

**FCCM** International Symposium on Field-Programmable Custom Computing Machines. 37

**FF** flip-flop. 3, 33, 55, 74

**FIFO** first-in-first-out. 39

**FPGA** field programmable gate array. vii, viii, ix, x, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 27, 30, 33, 34, 35, 37, 53, 54, 55, 56, 59, 60, 61, 62, 63, 65, 70, 73, 74, 79, 82, 84, 88, 89, 93, 94, 97, 98, 99, 101, 102, 103, 107

**FSM** finite state machine. 9, 44, 45, 50, 51, 53, 54, 62, 63, 67, 68, 69, 70, 74, 80, 82, 84, 85, 97, 98, 101, 102, 105, 107, 108

**GPU** graphics processing unit. 96, 97

**HDD** hard disk drive. 12

**HLS** high-level synthesis. 97

**HTTPS** hypertext transfer protocol secure. 12, 19

**HW/SW** hardware/software. 98

**ICFPT** International Conference on Field-Programmable Technology. 27

**IP** intellectual property. 4

**ISC** instance-specific computing. 75, 77, 78, 79, 80, 108

**ISD** instance-specific design. 77, 79, 86, 90

**LIFO** last-in-first-out. 39

**LUT** lookup table. 3, 4, 31, 33, 52, 55, 74, 107

**MaxJ** extended version of Java used by MaxCompiler and MaxIDE. 84

**OTF** on-the-fly. 87, 88, 89, 108

**PAD** perfect asymmetric decay. 16, 17, 18, 25, 29, 53, 54, 56, 57, 58, 59, 60, 83, 84, 86, 93, 103, 108

**PCIe** peripheral component interconnect express. 6, 7, 34, 56, 107

**RAM** random access memory. 4, 52, 57, 92, 109, 111

**RAPL** running average power limit. 89

**RCon** Rijndael round constant. 21, 22, 31, 51, 52, 114

**ROM** read-only memory. 84, 85

**RSA** Rivest–Shamir–Adleman. 12, 92, 93

**SAT** propositional satisfiability problem. 93, 98

**SBox** Rijndael substitution box. 21, 22, 24, 52, 82, 85, 107, 113

**SCA** side-channel attack. 91

**SDK** software development kit. 97

**SDRAM** synchronous dynamic RAM. 56, 109

**SIMD** single instruction stream, multiple data streams. 96

**SLiC** simple live CPU. 7, 8

**SM** state machine. 6, 9, 10

**SRAM** static random access memory. 12, 13

**SSH** secure shell. 19

**SSL** secure sockets layer. 93

**TDP** thermal design power. 89

**TOS** top-of-stack. 53, 68, 69, 70

**TRETS** ACM Transactions on Reconfigurable Technology and Systems. 37

**VLIW** very long instruction word. 99

**VoIP** voice over Internet protocol. 19

**VPN** virtual private network. 12

**WLAN** wireless LAN. 19

**WPA2** Wi-Fi Protected Access 2. 12

**WS** work stealing. 40, 65, 66, 70, 74, 75, 77, 79, 96, 97

**XOr** exclusive Or. 21, 22, 23, 25, 28, 29, 31, 32, 51, 52, 78, 81, 84, 107

# Appendix A

# Supplemental Material

### Substitution Box SBox

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 63 | 7C | 77 | 7B | F2 | 6B | 6F | C5 | 30 | 01 | 67 | 2B | FE | D7 | AB | 76 |
| 10 | CA | 82 | C9 | 7D | FA | 59 | 47 | F0 | AD | D4 | A2 | AF | 9C | A4 | 72 | C0 |
| 20 | B7 | FD | 93 | 26 | 36 | 3F | F7 | CC | 34 | A5 | E5 | F1 | 71 | D8 | 31 | 15 |
| 30 | 04 | C7 | 23 | C3 | 18 | 96 | 05 | 9A | 07 | 12 | 80 | E2 | ED | 27 | B2 | 75 |
| 40 | 09 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| 50 | 53 | D1 | 00 | ED | 20 | FC | C1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| 60 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 02 | 7F | 50 | 3C | 9F | A8 |
| 70 | 51 | A3 | 40 | 8F | 92 | 9D | 38 | F5 | BC | B6 | DA | 21 | 10 | FF | F3 | D2 |
| 80 | CD | 0C | 13 | EC | 5F | 97 | 44 | 17 | C4 | A7 | 7E | 3D | 64 | 5D | 19 | 73 |
| 90 | 60 | 81 | 4F | DC | 22 | 2A | 90 | 88 | 46 | EE | B8 | 14 | DE | 5E | 0B | DB |
| A0 | E0 | 32 | 3A | 0A | 49 | 06 | 24 | 5C | C2 | D3 | AC | 62 | 91 | 95 | E4 | 79 |
| B0 | E7 | C8 | 37 | 6D | 8D | D5 | 4E | A9 | 6C | 56 | F4 | EA | 65 | 7A | AE | 08 |
| C0 | BA | 78 | 25 | 2E | 1C | A6 | B4 | C6 | E8 | DD | 74 | 1F | 4D | BD | 8B | 8A |
| D0 | 70 | 3E | B5 | 66 | 48 | 03 | F6 | 0E | 61 | 35 | 57 | B9 | 86 | C1 | 1D | 9E |
| E0 | E1 | F8 | 98 | 11 | 69 | D9 | 8E | 94 | 9B | 1E | 87 | E9 | CE | 55 | 28 | DF |
| F0 | 8C | A1 | 89 | 0D | BF | E6 | 42 | 68 | 41 | 99 | 2D | 0F | B0 | 54 | BB | 16 |

### Inverse Substitution Box $SBox^{-1}$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 52 | 09 | 6A | D5 | 30 | 36 | A5 | 38 | BF | 40 | A3 | 9E | 81 | F3 | D7 | FB |
| 10 | 7C | E3 | 39 | 82 | 9B | 2F | FF | 87 | 34 | 8E | 43 | 44 | C4 | DE | E9 | CB |
| 20 | 54 | 7B | 94 | 32 | A6 | C2 | 23 | 3D | EE | 4C | 95 | 0B | 42 | FA | C3 | 4E |
| 30 | 08 | 2E | A1 | 66 | 28 | D9 | 24 | B2 | 76 | 5B | A2 | 49 | 6D | 8B | D1 | 25 |
| 40 | 72 | F8 | F6 | 64 | 86 | 68 | 98 | 16 | D4 | A4 | 5C | CC | 5D | 65 | B6 | 92 |
| 50 | 6C | 70 | 48 | 50 | FD | ED | B9 | DA | 5E | 15 | 46 | 57 | A7 | 8D | 9D | 84 |
| 60 | 90 | D8 | AB | 00 | 8C | BC | D3 | 0A | F7 | E4 | 58 | 05 | B8 | B3 | 45 | 06 |
| 70 | D0 | 2C | 1E | 8F | CA | 3F | 0F | 02 | C1 | AF | BD | 03 | 01 | 13 | 8A | 6B |
| 80 | 3A | 91 | 11 | 41 | 4F | 67 | DC | EA | 97 | F2 | CF | CE | F0 | B4 | E6 | 73 |
| 90 | 96 | AC | 74 | 22 | E7 | AD | 35 | 85 | E2 | F9 | 37 | E8 | 1C | 75 | DF | 6E |
| A0 | 47 | F1 | 1A | 71 | 1D | 29 | C5 | 89 | 6F | B7 | 62 | 0E | AA | 18 | BE | 1B |
| B0 | FC | 56 | 3E | 4B | C6 | D2 | 79 | 20 | 9A | DB | C0 | FE | 78 | CD | 5A | F4 |
| C0 | 1F | DD | A8 | 33 | 88 | 07 | C7 | 31 | B1 | 12 | 10 | 59 | 27 | 80 | EC | 5F |
| D0 | 60 | 51 | 7F | A9 | 19 | B5 | 4A | 0D | 2D | E5 | 7A | 9F | 93 | C9 | 9C | EF |
| E0 | A0 | E0 | 3B | 4D | AE | 2A | F5 | B0 | C8 | EB | BB | 3C | 83 | 53 | 99 | 61 |
| F0 | 17 | 2B | 04 | 7E | BA | 77 | D6 | 26 | E1 | 69 | 14 | 63 | 55 | 21 | 0C | 7D |

### Round Constants RCon

AES-128 only takes the highlighted 10 round constants.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 8D | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 | 6C | D8 | AB | 4D | 9A |
| 10 | 2F | 5E | BC | 63 | C6 | 97 | 35 | 6A | D4 | B3 | 7D | FA | EF | C5 | 91 | 39 |
| 20 | 72 | E4 | D3 | BD | 61 | C2 | 9F | 25 | 4A | 94 | 33 | 66 | CC | 83 | 1D | 3A |
| 30 | 74 | E8 | CB | 8D | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 | 6C | D8 |
| 40 | AB | 4D | 9A | 2F | 5E | BC | 63 | C6 | 97 | 35 | 6A | D4 | B3 | 7D | FA | EF |
| 50 | C5 | 91 | 39 | 72 | E4 | D3 | BD | 61 | C2 | 9F | 25 | 4A | 94 | 33 | 66 | CC |
| 60 | 83 | 1D | 3A | 74 | E8 | CB | 8D | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B |
| 70 | 36 | 6C | D8 | AB | 4D | 9A | 2F | 5E | BC | 63 | C6 | 97 | 35 | 6A | D4 | B3 |
| 80 | 7D | FA | EF | C5 | 91 | 39 | 72 | E4 | D3 | BD | 61 | C2 | 9F | 25 | 4A | 94 |
| 90 | 33 | 66 | CC | 83 | 1D | 3A | 74 | E8 | CB | 8D | 01 | 02 | 04 | 08 | 10 | 20 |
| A0 | 40 | 80 | 1B | 36 | 6C | D8 | AB | 4D | 9A | 2F | 5E | BC | 63 | C6 | 97 | 35 |
| B0 | 6A | D4 | B3 | 7D | FA | EF | C5 | 91 | 39 | 72 | E4 | D3 | BD | 61 | C2 | 9F |
| C0 | 25 | 4A | 94 | 33 | 66 | CC | 83 | 1D | 3A | 74 | E8 | CB | 8D | 01 | 02 | 04 |
| D0 | 08 | 10 | 20 | 40 | 80 | 1B | 36 | 6C | D8 | AB | 4D | 9A | 2F | 5E | BC | 63 |
| E0 | C6 | 97 | 35 | 6A | D4 | B3 | 7D | FA | EF | C5 | 91 | 39 | 72 | E4 | D3 | BD |
| F0 | 61 | C2 | 9F | 25 | 4A | 94 | 33 | 66 | CC | 83 | 1D | 3A | 74 | E8 | CB | |

### Distribution of Ones and Zeros in AES Key Schedule

Figure A.1 shows the distribution of ones for the 10,000 AES-128 key schedules used in the evaluation of Chapter 5. The total number of bits in the whole key schedule is 1,408. The measured average of ones is 704.12 bits, which well approximates the expected 50%.

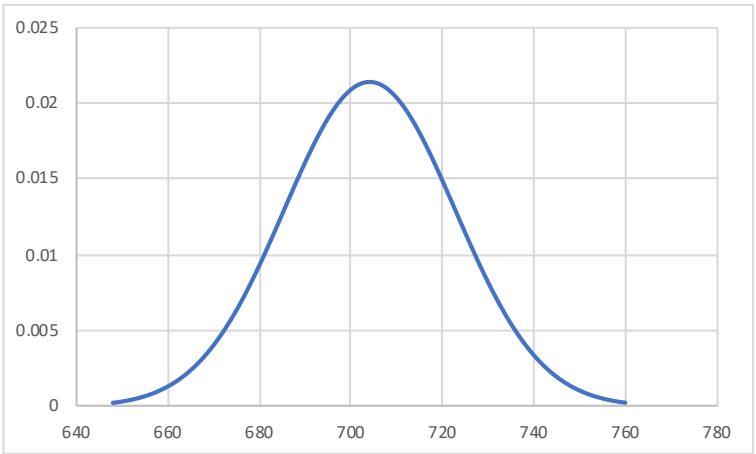



Figure A.1: Distribution of ones in 10,000 randomly generated AES-128 key schedules with $\mu = 704.12$ and $\sigma = 18.67$.

### Distribution of Values for the First Byte

Figure A.2 shows the distribution of the value for $0_0 \in [0, \ldots, 255]$ for the 10,000 AES-128 key schedules used in the evaluation of Chapter 5. This value is especially sensitive for the reconstruction with the static path from Table 5.1, page 46. As depicted in the figure, the values are equally distributed over the whole interval.

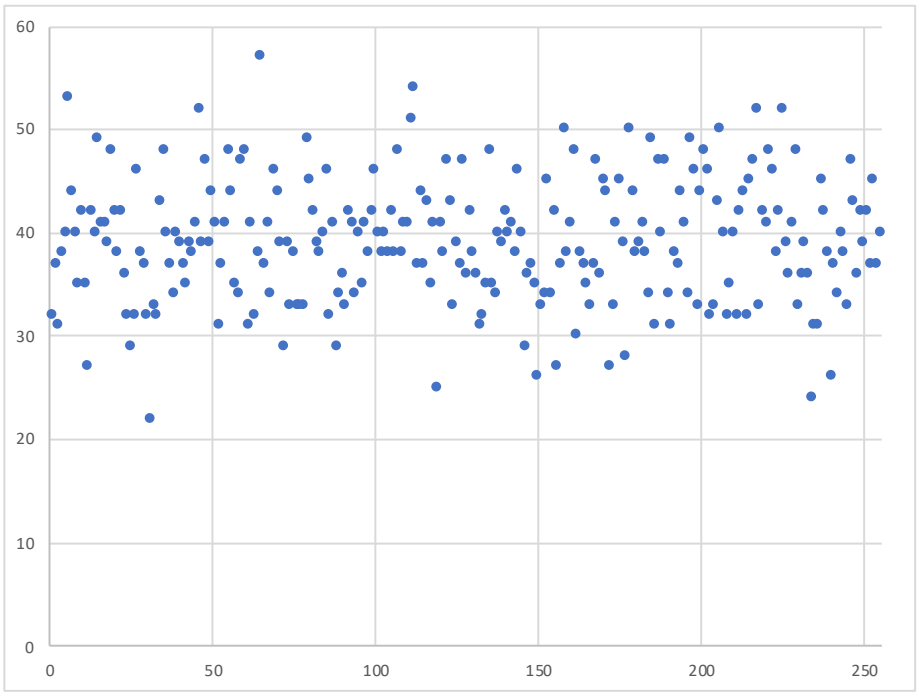



Figure A.2: Distribution of the value for $0_0 \in [0, \ldots, 255]$ in the 10,000 key schedules.

# Author's Publications


[1] Heinrich Riebler, Michael Lass, Robert Mittendorf, Thomas Löcke, and Christian Plessl. Efficient Branch and Bound on FPGAs Using Work Stealing and Instance-Specific Designs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(3):24:1–24:23, 2017.

[2] Heinrich Riebler, Tobias Kenter, Christian Plessl, and Christoph Sorge. Reconstructing AES Key Schedules from Decayed Memory with FPGAs. In *Proceedings International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 222–229. IEEE, 2014.

[3] Heinrich Riebler, Tobias Kenter, Christoph Sorge, and Christian Plessl. FPGA-accelerated Key Search for Cold-Boot Attacks against AES. In *Proceedings International Conference on Field Programmable Technology (ICFPT)*, page 386–389. IEEE, 2013.

[4] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Transparent Acceleration for Heterogeneous Platforms With Compilation to OpenCL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):14, 2019.

[5] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Automated Code Acceleration Targeting Heterogeneous OpenCL Devices. In *Proceedings ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, PPoPP '18, pages 417–418, New York, NY, USA, 2018. ACM.

[6] Heinrich Riebler, Gavin Vaz, Christian Plessl, Ettore MG Trainiti, Gianluca C Durelli, Emanuele Del Sozzo, Marco D Santambrogio, and Cristiana Bolchini. Using just-in-time code generation for transparent resource management in heterogeneous systems. In *Proceedings International Forum on Research and Technologies for Society and Industry (RTSI)*, pages 1–5. IEEE, 2016.

[7] Gianluca C Durelli, Marcello Pogliani, Antonio Miele, Christian Plessl, Heinrich Riebler, Marco D Santambrogio, Gavin Vaz, and Cristiana Bolchini. Runtime resource management in heterogeneous system architectures: The save approach. In *Proceedings International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 142–149. IEEE, 2014.

[8] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. Transparent offloading of computational hotspots from binary code to Xeon Phi. In *Proceedings Design, Automation and Test in Europe Conference (DATE)*, pages 1078–1083. EDA Consortium, 2015.

[9] Tobias Kenter, Gavin Francis Vaz, Heinrich Riebler, and Christian Plessl. Opportunities for deferring application partitioning and accelerator synthesis to runtime (extended abstract). In *Proceedings HiPEAC Workshop on Reonfigurable Computing (WRC)*, 2016.

[10] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Potential and methods for embedding dynamic offloading decisions into application code. *Computers and Electrical Engineering*, 55:91–111, 2016.

[11] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Deferring accelerator offloading decisions to application runtime. In *Proceedings International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2014. **Received best paper award.**

# Bibliography

[12] Specification of the Advanced Encryption Standard (AES). Nist standard 197, Federal Information Processing Standards, 2001.

[13] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The Data Locality of Work Stealing. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12. ACM, 2000.

[14] Norman Agin. Optimum Seeking with Branch and Bound. *Management Science*, 13(4):B–176, 1966.

[15] Andreas Agne, Markus Happe, Achim Lösch, Christian Plessl, and Marco Platzner. Self-Awareness as a Model for Designing and Operating Heterogeneous Multicores. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):13, 2014.

[16] Martin Albrecht and Carlos Cid. Cold Boot Key Recovery by Solving Polynomial Systems with Noise. In *Proceedings of the International Conference on Applied Cryptography and Network Security*, ACNS'11, pages 57–72, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Louis Victor Allis et al. *Searching for Solutions in Games and Artificial Intelligence*. Ponsen & Looijen Wageningen, 1994.

[18] Ross Anderson, E Biham, and L Knudsen. A Candidate Block Cipher for the Advanced Encryption Standard, 1998.

[19] Ross Anderson and Markus Kuhn. Tamper Resistance-a Cautionary Note. In *Proceedings of the USENIX Workshop on Electronic Commerce*, volume 2, pages 1–11, 1996.

[20] Ross Anderson and Markus Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *International Workshop on Security Protocols*, pages 125–136. Springer, 1997.

[21] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance Comparison of FPGA, GPU and CPU in Image Processing. In *International Conference on Field Programmable Logic and Applications*, pages 126–131. IEEE, 2009.

[22] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[23] Jean-Philippe Aumasson, Itai Dinur, Luca Henzen, Willi Meier, and Adi Shamir. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128. *Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS)*, page 147, 2009.

[24] David Avis and Charles Jordan. MTS: a Light Framework for Parallelizing Tree Search Codes. *arXiv preprint arXiv:1709.07605*, 2017.

[25] David Avis and Charles Jordan. MPLRS: A Scalable Parallel Vertex/Facet Enumeration Code. *Mathematical Programming Computation*, 10(2):267–302, 2018.

[26] Zachary K Baker and Viktor K Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the International symposium on Field programmable gate arrays (ACM/SIGDA)*, pages 223–232. ACM, 2004.

[27] Jason D Bakos. FPGA Acceleration of Gene Rearrangement Analysis. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 85–94. IEEE, 2007.

[28] Egon Balas and Paolo Toth. Branch and Bound Methods for the Traveling Salesman Problem. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP, 1983.

[29] Lucio Barreto and Michael Bauer. Parallel Branch and Bound Algorithm-a Comparison between Serial, OpenMP and MPI Implementations. In *Journal of Physics: Conference Series*, volume 256, page 12018. IOP Publishing, 2010.

[30] A Tarun Beri, B Sorav Bansal, and C Subodh Kumar. Locality Aware Work-Stealing based Scheduling in Hybrid CPU-GPU Clusters. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 48, 2015.

[31] Daniel J Bernstein. Cache-timing Attacks on AES, 2005.

[32] Carl Bialik. About Those Hundreds of Thousands of Lost Laptops at Airports. https://blogs.wsj.com/numbers/about-those-hundreds-of-thousands-of-lost-laptops-at-airports-413/. Accessed: 2019-09-18.

[33] Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. Hybrid Metaheuristics in Combinatorial Optimization: A Survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

[34] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *JACM*, 46(5):720–748, September 1999.

[35] Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[36] Abraham Bookstein, Vladimir A Kulyukin, and Timo Raita. Generalized Hamming Distance. *Information Retrieval*, 5(4):353–375, 2002.

[37] Andrey Borisenko, Michael Haidl, and Sergei Gorlatch. A GPU Parallelization of Branch-and-Bound for Multiproduct Batch Plants Optimization. *The Journal of Supercomputing*, 73(2):639–651, 2017.

[38] Abdelamine Boukedjar, Mohamed Esseghir Lalami, and Didier El-Baz. Parallel Branch and Bound on a CPU-GPU System. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 392–398. IEEE, 2012.

[39] Benoıt Bourbeau, Teodor Gabriel Crainic, and Bernard Gendron. Branch-and-Bound Parallelization Strategies Applied to a Depot Location and Container Fleet Management Problem. *Parallel Computing*, 26(1):27–46, 2000.

[40] Stephen Boyd and Jacob Mattingley. Branch and Bound Methods. 2007.

[41] Vincent Boyer and Didier El Baz. Recent Advances on GPU Computing in Operations Research. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1778–1787. IEEE, 2013.

[42] Adrian Brüngger, Ambros Marzetta, Komei Fukuda, and Jürg Nievergelt. The Parallel Search Bench ZRAM and its Applications. *Operations Research*, 90:45–63, 1999.

[43] Michael J Brusco and Stephanie Stahl. *Branch-and-Bound Applications in Combinatorial Data Analysis*. Springer Science & Business Media, 2006.

[44] A Victor Cabot and S Selcuk Erenguc. Some Branch-and-Bound Procedures for Fixed-cost Transportation Problems. *Naval Research Logistics Quarterly*, 31(1):145–154, 1984.

[45] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation. In *ACM SIGPLAN Notices*, volume 21, pages 152–161. ACM, 1986.

[46] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-based FPGA. *Journal of Cryptology*, 24(2):247–268, 2011.

[47] Richard Carbone, C Bean, and Martin Salois. An In-Depth Analysis of the Cold Boot Attack. *DRDC Valcartier, Defence Research and Development, Canada, Tech. Rep*, 2011.

[48] Vincent Carlier, Hervé Chabanne, Emmanuelle Dottax, and Hervé Pelletier. Electromagnetic Side Channels of an FPGA Implementation of AES. In *CRYPTOLOGY EPRINT ARCHIVE, REPORT*. Citeseer, 2004.

[49] Tiago Carneiro, Albert Einstein Muritiba, Marcos Negreiros, and Gustavo Augusto Lima de Campos. A New Parallel Schema for Branch-and-Bound Algorithms using GPGPU. In *International Symposium on Computer Architecture and High Performance Computing*, pages 41–47. IEEE, 2011.

[50] Giorgio Carpaneto and Paolo Toth. Some new Branching and Bounding Criteria for the Asymmetric Travelling Salesman Problem. *Management Science*, 26(7):736–743, 1980.

[51] Leocadio G Casado, JA Martinez, Inmaculada García, and Eligius MT Hendrix. Branch-and-Bound Interval Global Optimization on Shared Memory Multiprocessors. *Optimization Methods & Software*, 23(5):689–701, 2008.

[52] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[53] Imen Chakroun. *Parallel Heterogeneous Branch and Bound Algorithms for Multi-core and Multi-GPU Environments*. PhD thesis, 2013.

[54] Imen Chakroun, Nordine Melab, Mohand Mezmaz, and Daniel Tuyttens. Combining Multi-Core and GPU Computing for Solving Combinatorial Optimization Problems. *Journal of Parallel and Distributed Computing*, 73(12):1563–1577, 2013.

[55] Imen Chakroun and Nouredine Melab. An Adaptative multi-GPU based Branch-and-Bound. a Case Study: the Flow-Shop Scheduling Problem. In *IEEE International Conference on High Performance Computing and Communication, International Conference on Embedded Software and Systems*, pages 389–395. IEEE, 2012.

[56] Imen Chakroun and Nouredine Melab. Towards a Heterogeneous and adaptive parallel Branch-and-Bound Algorithm. *Journal of Computer and System Sciences*, 81(1):72–84, 2015.

[57] Imen Chakroun, Mohand Mezmaz, Nouredine Melab, and Ahcene Bendjoudi. Reducing Thread Divergence in a GPU-Accelerated Branch-and-Bound Algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[58] David Chase and Yossi Lev. Dynamic Circular Work-Stealing Deque. In *Proceedings of the Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28. ACM, 2005.

[59] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 55–67. IEEE, 2018.

[60] Paweł Chodowiec and Kris Gaj. Very Compact FPGA Implementation of the AES Algorithm. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 319–333. Springer, 2003.

[61] Jaeseok Choi, AA El-Keib, and Trungtinh Tran. A Fuzzy Branch and Bound-based Transmission System Expansion Planning for the Highest Satisfaction Level of the Decision Maker. *IEEE Transactions on Power Systems*, 20(1):476–484, 2005.

[62] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. A Survey of Network Virtualization. *Computer Networks*, 54(5):862–876, 2010.

[63] Robert Churchhouse and RF Churchhouse. *Codes and Ciphers: Julius Caesar, the Enigma, and the Internet*. Cambridge University Press, 2002.

[64] Christopher R Clark and David E Schimmel. Scalable Pattern Matching for High Speed Networks. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257. IEEE, 2004.

[65] Jens Clausen. Branch and Bound Algorithms-principles and Examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[66] Don Coppersmith. The Data Encryption Standard (DES) and its Strength Against Attacks. *IBM journal of research and development*, 38(3):243–250, 1994.

[67] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[68] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[69] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the Conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[70] Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel Branch-and-Bound Algorithms. *Parallel combinatorial Optimization*, 1:1–28, 2006.

[71] D Crookes, K Benkrid, A Bouridane, K Alotaibi, and A Benkrid. Design and Implementation of a High Level Programming Environment for FPGA-based Image Processing. *IEE Proceedings-Vision, Image and Signal Processing*, 147(4):377–384, 2000.

[72] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In *Proceedings International Conference on Smart Card Research and Applications (CARDIS)*, pages 277–284. Springer, 2000.

[73] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[74] GH Dastghaibifard, E Ansari, SM Sheykhalishahi, A Bavandpouri, and E Ashoor. A Parallel Branch and Bound Algorithm for Vehicle Routing Problem. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 2, pages 19–21, 2008.

[75] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[76] John D Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. A Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers. In *Proceedings Design Automation Conference (DAC)*, pages 780–785. ACM, 2008.

[77] Lawrence Davis. Genetic Algorithms and Simulated Annealing. 1987.

[78] R Davis. The Data Encryption Standard in Perspective. *IEEE Communications Society Magazine*, 16(6):5–9, 1978.

[79] Guerric Meurice de Dormale and Jean-Jacques Quisquater. High-speed Hardware Implementations of Elliptic Curve Cryptography: A Survey. *Journal of Systems Architecture*, 53(2-3):72–84, 2007.

[80] Erik Demeulemeester and Willy Herroelen. A Branch-and-Bound Procedure for the Multiple Resource-constrained Project Scheduling Problem. *Management Science*, 38(12):1803–1818, 1992.

[81] BC Desai. The BPU, a Staged Parallel Processing System to Solve the Zero-One Problem. In *Proceedings of ICS*, volume 78, pages 802–817, 1978.

[82] Bipin C Desai. A Parallel Microprocessing System. In *Proceedings of the International Conference on Parallel Processing*, volume 136, 1979.

[83] Alexandros C Dimopoulos, Christos Pavlatos, and George Papakonstantinou. A General Purpose Branch and Bound Parallel Algorithm. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 317–321. IEEE, 2016.

[84] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE, 2009.

[85] Itai Dinur, Tim Güneysu, Christof Paar, Adi Shamir, and Ralf Zimmermann. An Experimentally Verified Attack on Full Grain-128 using Dedicated Reconfigurable Hardware. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 327–343. Springer, 2011.

[86] Itai Dinur and Adi Shamir. Breaking Grain-128 with Dynamic Cube Attacks. In *International Workshop on Fast Software Encryption*, pages 167–187. Springer, 2011.

[87] A Djerrah, Bertrand Le Cun, V-D Cung, and Catherine Roucairol. Bob++: Framework for Solving Optimization Problems with Branch-and-Bound Methods. In *IEEE International Conference on High Performance Distributed Computing*, pages 369–370. IEEE, 2006.

[88] Chrilly Donninger, Alex Kure, and Ulf Lorenz. Parallel Brutus: the First Distributed, FPGA Accelerated Chess Program. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 44. IEEE, 2004.

[89] Sumanth Donthi and Roger L Haggard. A Survey of Dynamically Reconfigurable FPGA Devices. In *Proceedings of the Southeastern Symposium on System Theory*, pages 422–426. IEEE, 2003.

[90] Isabel Dorta, Coromoto León, and Casiano Rodríguez. A Comparison Between MPI and OpenMP Branch-and-Bound Skeletons. In *Proceedings International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 66–73. IEEE, 2003.

[91] Isabel Dorta, Coromoto Leon, and Casiano Rodriguez. Parallel Branch-and-Bound Skeletons: Message Passing and Shared Memory Implementations. In *International Conference on Parallel Processing and Applied Mathematics*, pages 286–291. Springer, 2003.

[92] D. E. Wood E. L. Lawler. Branch-And-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, 1966.

[93] Jonathan Eckstein. Parallel Branch-and-Bound Algorithms for General Mixed Integer programming on the CM-5. *SIAM Journal on Optimization*, 4(4):794–814, 1994.

[94] Jonathan Eckstein. Distributed versus Centralized Storage and Control for Parallel Branch and Bound: Mixed Integer Programming on the CM-5. *Computational Optimization and Applications*, 7(2):199–220, 1997.

[95] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A Survey of Lightweight-Cryptography Implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.

[96] Vinoth Krishnan Elangovan, Rosa.M. Badia, and Eduard Ayguadé. Scalability and Parallel Execution of OmpSs-OpenCL Tasks on Heterogeneous CPU-GPU Environment. In *Proceedings International Conference on Supercomputing (ISC)*, pages 141–155, 2014.

[97] Shimon Even. *Graph Algorithms*. Cambridge University Press, 2011.

[98] Yuri Evtushenko, Mikhail Posypkin, and Israel Sigal. A Framework for Parallel Large-Scale Global Optimization. *Computer Science-Research and Development*, 23(3-4):211–215, 2009.

[99] Robert Neil Faiman Jr. Method of Constructing a Constant-Folding Mechanism in a Multilanguage Optimizing Compiler, November 10 1998. US Patent 5,836,014.

[100] Marshall L Fisher. The Lagrangian Relaxation Method for Solving Integer Programming Problems. *Management Science*, 27(1):1–18, 1981.

[101] Michael J Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.

[102] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[103] Clemens Fruhwirth. *New Methods in Hard Disk Encryption*. na, 2005.

[104] Zhaohui Fu and Sharad Malik. Solving the Minimum-Cost Satisfiability Problem using SAT based Branch-and-Bound Search. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 852–859. ACM, 2006.

[105] Bernard Gendron and Teodor Gabriel Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[106] Alan George, Herman Lam, and Greg Stitt. Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing. *Computing in Science & Engineering*, 13(1):82, 2011.

[107] Paul C Gilmore. Optimal and Suboptimal Algorithms for the Quadratic Assignment Problem. *Journal of the Society for Industrial and Applied Mathematics*, 10(2):305–313, 1962.

[108] Kyrre Glette, Jim Torresen, and Moritoshi Yasunaga. Online Evolution for a High-Speed Image Recognition System Implemented on a Virtex-II Pro FPGA. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 463–470. IEEE, 2007.

[109] Fred Glover and Manuel Laguna. Tabu Search. In *Handbook of Combinatorial Optimization*, pages 2093–2229. Springer, 1998.

[110] Jan Gmys, Rudi Leroy, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. Work Stealing with Private Integer–Vector–Matrix Data Structure for Multi-Core Branch-and-Bound Algorithms. *Concurrency and Computation: Practice and Experience*, 28(18):4463–4484, 2016.

[111] Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. A GPU-Based Branch-and-Bound Algorithm using Integer–Vector–Matrix Data Structure. *Parallel Computing*, 59:119–139, 2016.

[112] Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. IVM-based Parallel Branch-and-Bound using Hierarchical Work Stealing on Multi-GPU Systems. *Concurrency and Computation: Practice and Experience*, 29(9):e4019, 2017.

[113] Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 427–440. Springer, 2005.

[114] Bur Goode. Voice over Internet Protocol (VoIP). *Proceedings of the IEEE*, 90(9):1495–1517, 2002.

[115] Paul Grigoras. Instance Directed Tuning for Sparse Matrix Kernels on Reconfigurable Accelerators. 2018.

[116] William Gropp, William D Gropp, Argonne Distinguished Fellow Emeritus Ewing Lusk, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, volume 1. MIT Press, 1999.

[117] Tim Güneysu, Timo Kasper, Martin Novotnỳ, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.

[118] Guang-liang Guo, Quan Qian, and Rui Zhang. Different Implementations of AES Cryptographic Algorithm. In *IEEE International Conference on High Performance Computing and Communications, International Symposium on Cyberspace Safety and Security, and IEEE International Conference on Embedded Software and Systems*, pages 1848–1853. IEEE, 2015.

[119] Xu Guo, Zhimin Chen, and Patrick Schaumont. Energy and Performance Evaluation of an FPGA-based SoC Platform with AES and PRESENT Coprocessors. In *International Workshop on Embedded Computer Systems*, pages 106–115. Springer, 2008.

[120] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A Scalable Locality-aware Adaptive Work-Stealing Scheduler. In *Proceedings International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[121] Omprakash K Gupta and Arunachalam Ravindran. Branch and Bound Experiments in Convex Nonlinear Integer Programming. *Management Science*, 31(12):1533–1546, 1985.

[122] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. IMPACT: Imprecise Adders for Low-Power Approximate Computing. In *Proceedings of the IEEE/ACM International Symposium on Low-Power Electronics and Design*, pages 409–414. IEEE Press, 2011.

[123] Peter Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings Conference on USENIX Security Symposium*, SSYM'01, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

[124] S Haffner, A Monticelli, A Garcia, J Mantovani, and R Romero. Branch and Bound Algorithm for Transmission System Expansion Planning using a Transportation Model. *IEE Proceedings-Generation, Transmission and Distribution*, 147(3):149–156, 2000.

[125] Sérgio Haffner, A Monticelli, A Garcia, and R Romero. Specialised Branch-and-Bound Algorithm for Transmission Network Expansion Planning. *IEE Proceedings-Generation, Transmission and Distribution*, 148(5):482–488, 2001.

[126] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, May 2009.

[127] Youssef Hamadi and David Merceron. Reconfigurable Architectures: A new Vision for Optimization Problems. In *Principles and Practice of Constraint Programming (CP)*, pages 209–221. Springer, 1997.

[128] Jie Han and Michael Orshansky. Approximate Computing: An Emerging Paradigm for Energy-Efficient Design. In *IEEE European Test Symposium (ETS)*, pages 1–6. IEEE, 2013.

[129] Scott Hauck and Andre DeHon. *Reconfigurable Computing: the Theory and Practice of FPGA-based Computation*. Morgan Kaufmann, 2010.

[130] Wilko Henecka, Alexander May, and Alexander Meurer. Correcting Errors in RSA Private Keys. In *Annual Cryptology Conference*, pages 351–369. Springer, 2010.

[131] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '09, pages 1–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[132] Juan FR Herrera, José MG Salmerón, Eligius MT Hendrix, Rafael Asenjo, and Leocadio G Casado. On Parallel Branch and Bound Frameworks for Global Optimization. *Journal of Global Optimization*, 69(3):547–560, 2017.

[133] Alejandro Hevia and Marcos Kiwi. Strength of two Data Encryption Standard Implementations under Timing Attacks. *ACM Transactions on Information and System Security (TISSEC)*, 2(4):416–437, 1999.

[134] Udo Hönig and Wolfram Schiffmann. A Parallel Branch–and–Bound Algorithm for Computing Optimal Task Graph Schedules. In *International Conference on Grid and Cooperative Computing*, pages 18–25. Springer, 2003.

[135] You Huang, Junzhong Shen, Yuran Qiao, Mei Wen, and Chunyuan Zhang. MALMM: A Multi-Array Architecture for Large-Scale Matrix Multiplication on FPGA. *IEICE Electronics Express*, pages 15–20180286, 2018.

[136] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.

[137] Edward Ignall and Linus Schrage. Application of the Branch and Bound Technique to some Flow-Shop Scheduling Problems. *Operations Research*, 13(3):400–412, 1965.

[138] Kimmo Järvinen, Matti Tommiska, and Jorma Skyttä. Comparative Survey of High-Performance Cryptographic Algorithm Implementations on FPGAs. *IEE Proceedings-Information Security*, 152(1):3–12, 2005.

[139] Michael Jünger and Peter Störmer. Solving Large-Scale Traveling Salesman Problems with Parallel Branch-and-Cut. 1995.

[140] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC-Instance-Specific Algorithm Configuration. In *ECAI*, volume 215, pages 751–756, 2010.

[141] Abdel Alim Kamal and Amr M. Youssef. Applications of SAT Solvers to AES Key Recovery from Decayed Key Schedule Images. In *Proceedings International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE '10, pages 216–220, Washington, DC, USA, July 2010. IEEE Computer Society.

[142] Kenji Kanazawa and Tsutomu Maruyama. An Approach for Solving Large SAT Problems on FPGA. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(1):10, 2010.

[143] Brian Kaplan and Matthew Geiger. RAM is Key: Extracting Disk Encryption Keys From Volatile Memory. Master's thesis, Carnegie Mellon University, 2007.

[144] Vladimir Kašík. Acceleration of Backtracking Algorithm with FPGA. In *Proceedings International Conference on Applied Electronics (AE)*, pages 1–4. IEEE, 2010.

[145] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on Parallel Programming Model. In *IFIP International Conference on Network and Parallel Computing*, pages 266–275. Springer, 2008.

[146] Meenakshi Kaul and Ranga Vemuri. Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures. In *Proceedings Design, Automation and Test in Europe*, pages 389–396. IEEE, 1998.

[147] Srinidhi Kestur, John D. Davis, and Eric S. Chung. Towards a Universal FPGA Matrix-Vector Multiplication Architecture. In *Proceedings International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, Washington, DC, USA, 2012. IEEE Computer Society.

[148] Farouk Khelil, Mohamed Hamdi, Sylvain Guilley, Jean Luc Danger, and Nidhal Selmane. Fault Analysis Attack on an FPGA AES Implementation. In *New Technologies, Mobility and Security*, pages 1–5. IEEE, 2008.

[149] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory without Accessing them: An Experimental Study of DRAM Disturbance Errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[150] Tobias Klein. All Your Private Keys are Belong to us - Extracting RSA Private Keys and Certificates from Process Memory. http://www.trapkit.de/research/sslkeyfinder/, February 2006.

[151] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[152] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[153] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[154] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[155] M. Koester, W. Luk, and G. Brown. A Hardware Compilation Flow for Instance-Specific VLIW Cores. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, September 2008.

[156] Samia Kouki, Mohamed Jemni, and Talel Ladhari. Scalable Distributed Branch and Bound for the Permutation Flow Shop Problem. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 503–508. IEEE, 2013.

[157] C. Kroer and Y. Malitsky. Feature Filtering for Instance-Specific Algorithm Configuration. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 849–855, November 2011.

[158] Volker Krummel. Sicherheit und Anwendungen des Advaned Enryption Standards (AES) Rijndael. Master's thesis, Paderborn University, November 2001.

[159] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer Berlin Heidelberg, 2006.

[160] V. Kumar, A. Sbîrlea, A. Jayaraj, Z. Budimlić, D. Majeti, and V. Sarkar. Heterogeneous Work-Stealing across CPU and DSP Cores. In *Proceedings High Performance Extreme Computing Conference (HPEC)*, pages 1–6, September 2015.

[161] Vipin Kumar and Laveen N Kanal. Parallel Branch-and-Bound Formulations for And/Or Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):768–778, 1984.

[162] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[163] Ian Kuon, Russell Tessier, Jonathan Rose, et al. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[164] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[165] VARONIS DATA LAB. *Data Under Attack: 2018 Global Data Risk Report*. PhD thesis, University of Portsmouth, 2018.

[166] Ten-Hwang Lai and Sartaj Sahni. Anomalies in Parallel Branch-and-Bound Algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[167] Mohamed Esseghir Lalami and Didier El-Baz. GPU Implementation of the Branch and Bound Method for Knapsack Problems. In *IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1769–1777. IEEE, 2012.

[168] Christoph H Lampert, Matthew B Blaschko, and Thomas Hofmann. Efficient Subwindow Search: A Branch and Bound Framework for Object Localization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12):2129–2142, 2009.

[169] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[170] Arash Habibi Lashkari, Mir Mohammad Seyed Danesh, and Behrang Samadi. A Survey on Wireless Security Protocols (WEP, WPA and WPA2/802.11 i). In *IEEE International Conference on Computer Science and Information Technology*, pages 48–52. IEEE, 2009.

[171] Hyung Tae Lee, HongTae Kim, Yoo-Jin Baek, and Jung Hee Cheon. Correcting Errors in Private Keys Obtained from Cold Boot Attacks. In *Proceedings International Conference on Information Security and Cryptology*, ICISC'11, pages 74–87, Berlin, Heidelberg, 2012. Springer-Verlag.

[172] Uroš Legat. *On-line Testing and Recovery of FPGA-based Systems*. PhD thesis, Jožef Stefan International Postgraduate School, Ljubljana, Slovenia, Mai 2012.

[173] Charles E Leiserson, Flavio M Rose, and James B Saxe. Optimizing Synchronous Circuitry by Rretiming. In *Third Caltech Conference on Very Large Scale Integration*, pages 87–116. Springer, 1983.

[174] Stefan Lemsitzer, Johannes Wolkerstorfer, Norbert Felber, and Matthias Braendli. Multi-Gigabit GCM-AES Architecture Optimized for FPGAs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 227–238. Springer, 2007.

[175] Rudi Leroy. *Parallel Branch-and-Bound Revisited for Solving Permutation Combinatorial Optimization Problems on Multi-Core Processors and Coprocessors*. PhD thesis, 2015.

[176] Rudi Leroy, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. Work Stealing Strategies for Multi-Core Parallel Branch-and-Bound Algorithm using Factorial Number System. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, page 111. ACM, 2014.

[177] Guo-Jie Li and Benjamin W Wah. Coping with Anomalies in Parallel Branch-and-Bound Algorithms. *IEEE Transactions on Computers*, 100(6):568–573, 1986.

[178] Ryan Lim. Parallelization of John the Ripper (JtR) using MPI. *Nebraska: University of Nebraska*, 37, 2004.

[179] João VF Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin, and Nicolas Maillard. Design and Analysis of Scheduling Strategies for Multi-CPU and Multi-GPU Architectures. *Parallel Computing*, 44:37–52, 2015.

[180] J.V.F. Lima, T. Gautier, N. Maillard, and V. Danjean. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 75–82, October 2012.

[181] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of Regular Expression Pattern Matching Circuits on FPGA. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 2, pages 1–6. IEEE, 2006.

[182] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of Pattern Matching Circuits for Regular Expression on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(12):1303–1310, 2007.

[183] Walter Link and Hardo May. Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen. *Archiv fur Elektronik und Ubertragungstechnik 33*, pages 229–235, June 1979.

[184] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (USENIX Security)*, 2018.

[185] John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An Algorithm for the Traveling Salesman Problem. *Operations Research*, 11(6):972–989, 1963.

[186] Thomas Löcke. Instance-specific Computing in Hard- and Software for the Reconstruction of Decayed AES Key Schedules in Cold Boot Attacks. Master's thesis, Paderborn University, March 2015.

[187] Pavlos Malakonakis and Apostolos Dollas. Exploitation of Parallel Search Space Evaluation with FPGAs in Combinatorial Problems: the Eternity II Case. In *Proceedings International Conference on Field Programmable Logic and Applications (FPL)*, pages 264–268. IEEE, 2011.

[188] Maxeler Technologies Inc. MaxCompiler - Manager Compiler Tutorial, Dezember 2012.

[189] Maxeler Technologies Inc. MaxCompiler - State Machine Tutorial, Dezember 2012.

[190] Rita Mayer-Sommer. Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92. Springer, 2000.

[191] Ciaran McCreesh and Patrick Prosser. The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound. *ACM Transactions on Parallel Computing*, 2(1):8, 2015.

[192] N. Melab, R. Leroy, M. Mezmaz, and D. Tuyttens. Parallel Branch-and-Bound using Private IVM-based Work Stealing on Xeon Phi MIC Coprocessor. In *c-hpcs*, pages 394–399, July 2015.

[193] Nouredine Melab, Imen Chakroun, and Ahcène Bendjoudi. Graphics Processing Unit-Accelerated Bounding for Branch-and-Bound Applied to a Permutation Problem using Data Access Optimization. *Concurrency and Computation: Practice and Experience*, 26(16):2667–2683, 2014.

[194] Nouredine Melab, Imen Chakroun, Mohand Mezmaz, and Daniel Tuyttens. A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem. In *IEEE International Conference on Cluster Computing*, pages 10–17. IEEE, 2012.

[195] Nouredine Melab, Jan Gmys, Mohand Mezmaz, and Daniel Tuyttens. Multi-Core Versus Many-Core Computing for Many-Task Branch-and-Bound Applied to Big Optimization Problems. *Future Generation Computer Systems*, 82:472–481, 2018.

[196] Nouredine Melab, Jan Gmys, Mohand Mezmaz, and Daniel Tuyttens. Many-Core Branch-and-Bound for GPU Accelerators and MIC Coprocessors. In *High-Performance Simulation-Based Optimization*, pages 275–291. Springer, 2020.

[197] Oskar Mencer, Kuen Hung Tsoi, Stephen Craimer, Timothy Todman, Wayne Luk, Ming Yee Wong, and Philip Heng Wai Leong. Cube: A 512-fpga Cluster. In *Southern Conference on Programmable Logic (SPL)*, pages 51–57. IEEE, 2009.

[198] Thomas S Messerges, Ezzat A Dabbish, and Robert H Sloan. Examining Smartcard Security under the Threat of Power Analysis Attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

[199] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 144–157. Springer, 1999.

[200] Mohand Mezmaz, Rudi Leroy, Nouredine Melab, and Daniel Tuyttens. A Multi-Core Parallel Branch-and-Bound Algorithm using Factorial Number System. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1203–1212. IEEE, 2014.

[201] Mohand Mezmaz, Nouredine Melab, and El-Ghazali Talbi. A Grid-enabled Branch and Bound Algorithm for Solving Challenging Combinatorial Optimization Problems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–9. IEEE, 2007.

[202] DL Miller and JF Pekny. Results from a Parallel Branch and Bound Algorithm for the Asymmetric Traveling Salesman Problem. *Operations Research Letters*, 8(3):129–135, 1989.

[203] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical Work Stealing on Manycore Clusters. In *c-pgas*, 2011.

[204] Sparsh Mittal. A Survey of Techniques for Managing and Leveraging Caches in GPUs. *Journal of Circuits, Systems, and Computers*, 23(8):1430002, 2014.

[205] Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys (CSUR)*, 48(4):62, 2016.

[206] LG Mitten. Branch-and-Bound Methods: General Formulation and Properties. *Operations Research*, 18(1):24–34, 1970.

[207] Robert Mittendorf. Advanced AES-Key Recovery from Decayed RAM-Dumps using Multi-threading and FPGAs. Master's thesis, Paderborn University, August 2014.

[208] Björn Morén. Utilizing Problem Specific Structures in Branch and Bound Methods for Manpower Planning, 2012.

[209] Bernard ME Moret, Jijun Tang, Li-San Wang, and Tandy Warnow. Steps Toward Accurate Reconstructions of Phylogenies from Gene-Order Data. *Journal of Computer and System Sciences*, 65(3):508–525, 2002.

[210] David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound Algorithms: A Survey of Recent Advances in Searching, Branching, and Pruning. *Discrete Optimization*, 19:79–102, 2016.

[211] Tilo Müller, Andreas Dewald, and Felix C. Freiling. AESSE: a Cold-Boot Resistant Implementation of AES. In *Proceedings of the European Workshop on System Security*, EUROSEC '10, pages 42–47, New York, NY, USA, 2010. ACM.

[212] Patrenahalli M. Narendra and K. Fukunaga. A Branch and Bound Algorithm for Feature Subset Selection. *IEEE Transactions on Computers*, C-26(9):917–922, September 1977.

[213] Dana S Nau, Vipin Kumar, and Laveen Kanal. General Branch and Bound, and its Relation to A* and AO*. *Artificial Intelligence*, 23(1):29–58, 1984.

[214] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. Strategies for Maximizing Utilization on Multi-CPU and Multi-GPU Heterogeneous Architectures. *Journal of Supercomputing*, 70(2):756–771, 2014.

[215] Nadia Nedjah and Chao Wang. *Reconfigurable and Adaptive Computing: Theory and Applications*. CRC press, 2018.

[216] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-Based Selection of Policies for SAT Solvers. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2009.

[217] Sean O'Neill. Who Really Believes that Fliers Lose 12,000 Laptops a Week? https://www.budgettravel.com/article/who-really-believes-that-fliers-lose-12000-laptops-a-week_10048. Accessed: 2019-09-18.

[218] Joo Guan Ooi and Kok Horng Kam. A Proof of Concept on Defending Cold Boot Attack. In *Asia Symposium on Quality Electronic Design*, pages 330–335. IEEE, 2009.

[219] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' track at the RSA Conference*, pages 1–20. Springer, 2006.

[220] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):36, 2011.

[221] Kenneth G Paterson, Antigoni Polychroniadou, and Dale L Sibborn. A Coding-Theoretic Approach to Recovering Noisy RSA Keys. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 386–403. Springer, 2012.

[222] Oliver Pell and Vitali Averbukh. Maximum Performance Computing with Dataflow Engines. *j-cse*, 14:98–103, 2012.

[223] M. Platzner. Reconfigurable Accelerators for Combinatorial Problems. *Computer*, 33(4):58–60, April 2000.

[224] Christian Plessl and Marco Platzner. Instance-Specific Accelerators for Minimum Covering. *Journal of Supercomputing*, 26(2):109–129, 2003.

[225] Larry Ponemon. Airport Insecurity: The Case of Missing and Lost Laptops. http://www.dell.com/downloads/global/services/dell_lost_laptop_study.pdf, June 2008.

[226] Larry Ponemon. The Billion Euro Lost Laptop Problem, 2011.

[227] Viktor K. Prasanna and Andreas Dandalis. FPGA-based Cryptography for Internet Security. In *Online Symposium for Electronic Engineers*. University of Southern California, Los Angeles, USA, 2000.

[228] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[229] Lin Quan, Weichang Shen, Jiao Cui, and Duan Geng. Application of 0–1 Knapsack MPI+ OpenMP Hybrid Programming Algorithm at MH Method. In *International Conference on Fuzzy Systems and Knowledge Discovery*, pages 2452–2456. IEEE, 2012.

[230] Jean-Jacques Quisquater and François-Xavier Standaert. Exhaustive Key Search of the DES: Updates and Refinements. *SHARCS 2005*, 2005.

[231] Sundararaman Rajagopalan, Rengarajan Amirtharajan, Har Narayan Upadhyay, and John Bosco Balaguru Rayappan. Survey and Analysis of Hardware Cryptographic and Steganographic Systems on FPGA. *Journal of Applied Sciences*, 12(3):201, 2012.

[232] Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A. Constantinides. A Case for Work-stealing on FPGAs with OpenCL Atomics. In *Proceedings International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 48–53, New York, NY, USA, 2016. ACM.

[233] A. Rashid, J. Leonard, and W.H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 196–204, April 1998.

[234] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. "O'Reilly Media, Inc.", 2007.

[235] Eric Rescorla. HTTP over TLS. 2000.

[236] Man Young Rhee. *Internet Security: Cryptographic Principles, Algorithms and Protocols*. John Wiley & Sons, 2003.

[237] 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.

[238] Andrés Rodríguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Rubén Gran, Darío Suárez, and Jose Nunez-Yanez. Parallel Multiprocessing and Scheduling on the Heterogeneous Xeon+ FPGA Platform. *The Journal of Supercomputing*, pages 1–21, 2019.

[239] Donald J Rose. On Simple Characterizations of k-Trees. *Discrete mathematics*, 7(3-4):317–322, 1974.

[240] Gaël Rouvroy, F-X Standaert, J-J Quisquater, and J-D Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications. In *Proceedings International Conference on Information Technology: Coding and Computing (ITCC)*, volume 2, pages 583–587. IEEE, 2004.

[241] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[242] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. Mapping a Data-Flow Programming Model onto Heterogeneous Platforms. *ACM SIGPLAN Notices*, 47(5):61–70, 2012.

[243] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: a 128-bit Block Cipher*. John Wiley & Sons, Inc., 1999.

[244] Micaela Serra and K Kent. Using FPGAs to solve the Hamiltonian cycle problem. In *Proceedings International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages III–228. IEEE, 2003.

[245] Adi Shamir and Nicko van Someren. Playing Hide and Seek with Stored Keys. In *Proceedings International Conference on Financial Cryptography (FC)*, FC '99, pages 118–124, London, UK, UK, 1999. Springer-Verlag.

[246] Claude E Shannon. Programming a Computer for Playing Chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.

[247] Junzhong Shen, Yuran Qiao, You Huang, Mei Wen, and Chunyuan Zhang. Towards a Multi-Array Architecture for Accelerating Large-Scale Matrix Multiplication on FPGAs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[248] Yusuke Shimai, Junichi Tani, Hiroki Noguchi, Hiroshi Kawaguchi, and Masahiko Yoshimoto. FPGA Implementation of Mixed Integer Quadratic Programming Solver for Mobile Robot Control. In *International Conference on Field-Programmable Technology*, pages 447–450. IEEE, 2009.

[249] Juliana MN Silva, Cristina Boeres, Lúcia MA Drummond, and Artur A Pessoa. Memory Aware Load Balance Strategy on a Parallel Branch-and-Bound Application. *Concurrency and Computation: Practice and Experience*, 27(5):1122–1144, 2015.

[250] Patrick Simmons. Security Through Amnesia: a Software-based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC '11, pages 73–82, New York, NY, USA, 2011. ACM.

[251] Gurpreet Singh. A Study of Encryption Algorithms (RSA, DES, 3DES and AES) for Information Security. *International Journal of Computer Applications*, 67(19), 2013.

[252] Filippo Sironi, Marco Triverio, Henry Hoffmann, Martina Maggio, and Marco D Santambrogio. Self-Aware Adaptation in FPGA-based Systems. In *International Conference on Field Programmable Logic and Applications*, pages 187–192. IEEE, 2010.

[253] Iouliia Skliarova and Antonio de Brito Ferrari. Reconfigurable Hardware SAT Solvers: A Survey of Systems. *IEEE Transactions on Computers*, 53(11):1449–1461, 2004.

[254] Iouliia Skliarova and António B Ferrari. A Software/Reconfigurable Hardware SAT Solver. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(4):408–419, 2004.

[255] Sergei Skorobogatov. Low Temperature Data Remanence in Static RAM. Technical report, University of Cambridge, June 2002.

[256] Sergei P Skorobogatov and Ross J Anderson. Optical Fault Induction Attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 2–12. Springer, 2002.

[257] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer Security: Principles and Practice*. Pearson Education Upper Saddle River (NJ, 2012.

[258] Carl Sechen Ted Stanion. A Method for Finding Good Ashenhurst Decompositions and its Application to FPGA Synthesis. In *Design Automation Conference*, pages 60–64. IEEE, 1995.

[259] John E Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[260] Welson Sun, Michael J Wirthlin, and Stephen Neuendorffer. FPGA Pipeline Synthesis Design Exploration using Module Selection and Resource Sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):254–265, 2007.

[261] Eric J Swankoski, Richard R Brooks, Vijaykrishnan Narayanan, Mahmut Kandemir, and Mary Jane Irwin. A Parallel Architecture for Secure FPGA Symmetric Encryption. In *Proceedings International Symposium on Parallel and Distributed Processing*, page 132. IEEE, 2004.

[262] El-Ghazali Talbi. *Parallel Combinatorial Optimization*, volume 58. John Wiley & Sons, 2006.

[263] Naif Tarafdar, Nariman Eskandari, Thomas Lin, and Paul Chow. Designing for FPGAs in the Cloud. *IEEE Design & Test*, 35(1):23–29, 2017.

[264] Jim Torresen, Christian Plessl, and Xin Yao. Self-Aware and Self-Expressive Systems. *Computer*, 48(7):18–20, 2015.

[265] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM, 2002.

[266] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A Lightweight Performance-oriented Tool Suite for x86 Multicore Environments. In *International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.

[267] Stephen M Trimberger and Jason J Moore. FPGA Cecurity: Motivations, Features, and Applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[268] Kuen Hung Tsoi and Wayne Luk. Axel: a Heterogeneous Cluster with FPGAs and GPUs. In *Proceedings International symposium on Field programmable gate arrays (ACM/SIGDA)*, pages 115–124. ACM, 2010.

[269] Alex Tsow. An Improved Recovery Algorithm for Decayed AES Key Schedule Images. In *Proceedings International Workshop on Selected Areas in Cryptography (SAC)*, pages 215–230, 2009.

[270] Alex Tsow. An Improved Recovery Algorithm for Decayed AES Key Schedule Images. In *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin Heidelberg, 2009.

[271] Isa Servan Uzun, Abbes Amira, and Ahmed Bouridane. FPGA Implementations of Fast Fourier Transforms for Real-Time Signal and Image Processing. *IEE Proceedings-Vision, Image and Signal Processing*, 152(3):283–296, 2005.

[272] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. Heterogeneous Resource-Elastic Scheduling for CPU+ FPGA Architectures. In *Proceedings International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, page 1. ACM, 2019.

[273] Peter JM Van Laarhoven and Emile HL Aarts. Simulated Annealing. In *Simulated Annealing: Theory and Applications*, pages 7–15. Springer, 1987.

[274] John Villasenor and William H Mangione-Smith. Configurable Computing. *Scientific American*, 276(6):54–9, 1997.

[275] Stefan Vömel and Felix C. Freiling. A Survey of Main Memory Acquisition and Analysis Techniques for the Windows Operating System. *Digit. Investig.*, 8(1):3–22, July 2011.

[276] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[277] Trong-Tuan Vu and Bilel Derbel. Parallel Branch-and-Bound in Multi-Core Multi-CPU Multi-GPU Heterogeneous Environments. *Future Generation Computer Systems*, 56:95–109, 2016.

[278] Trong-Tuan Vu, Bilel Derbel, and Nouredine Melab. Adaptive Dynamic Load Balancing in Heterogeneous Multiple GPUs-CPUs Distributed Setting: Case Study of b&b Tree Search. In *International Conference on Learning and Intelligent Optimization*, pages 87–103. Springer, 2013.

[279] Benjamin W Wah and YW Ma. MANIP-a Parallel Computer System for Implementing Branch and Bound Algorithms. In *Proceedings of the Annual Symposium on Computer Architecture*, pages 239–262. IEEE Computer Society Press, 1981.

[280] Shin'ichi Wakabayashi and Kenji Kikuchi. An Instance-specific Hardware Algorithm for Finding a Maximum Clique. In *Proceedings International Conference on Field Programmable Logic and Applications (FPL)*, pages 516–525, 2004.

[281] Qichao Wang. Localization and Extraction of Cryptographic Keys from Memory Images and Data Streams. Master's thesis, Paderborn University, 2012.

[282] Shuenn-Shyang Wang and Wan-Sheng Ni. An Efficient FPGA Implementation of Advanced Encryption Standard Algorithm. In *IEEE International Symposium on Circuits and Systems (IEEE Cat. No. 04CH37512)*, volume 2, pages II–597. IEEE, 2004.

[283] Mark N Wegman and F Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[284] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC—First Experiences with Real-World Applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[285] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Security on FPGAs: State-of-the-art Implementations and Attacks. *ACM Transactions Embedded Computing Systems*, 3(3):534–574, August 2004.

[286] Hui Yan, Zhaoshi Li, Leibo Liu, Shouyi Yin, and Shaojun Wei. Constructing Concurrent Data Structures on FPGA with Channels. In *Proceedings International Symposium on Field-Programmable Gate Arrays (ACM/SIGDA)*, pages 172–177. ACM, 2019.

[287] Jixiang Yang and Qingbi He. Scheduling Parallel Computations by Work Stealing: A Survey. *International Journal of Parallel Programming*, 46(2):173–197, 2018.

[288] Tatu Ylonen and Chris Lonvick. The Secure Shell (SSH) Protocol Architecture. 2006.

[289] Makoto Yokoo, Takayuki Suyama, and Hiroshi Sawada. Solving Satisfiability Problems using Field Programmable Gate Arrays: First Results. In *Principles and Practice of Constraint Programming (CP)*, pages 497–509. Springer, 1996.

[290] Joseph Zambreno, David Nguyen, and Alok Choudhary. Exploring Area/Delay Tradeoffs in an AES FPGA Implementation. In *International Conference on Field Programmable Logic and Applications*, pages 575–585. Springer, 2004.

[291] Peixin Zhong, Margaret Martonosi, Sharad Malik, and Pranav Ashar. Implementing Boolean Satisfiability in Configurable Hardware. In *Logic Synthesis Workshop*. Citeseer, 1997.

[292] Shijie Zhou and Viktor K Prasanna. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE, 2017.