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

HEINRICH RIEBLER, MICHAEL LASS, ROBERT MITTENDORF, THOMAS LÖCKE, and CHRISTIAN PLESSL, Paderborn University

Branch and bound (B&B) algorithms structure the search space as a tree and eliminate infeasible solutions early by pruning subtrees that cannot lead to a valid or optimal solution. Custom hardware designs significantly accelerate the execution of these algorithms. In this article, we demonstrate a high-performance B&B implementation on FPGAs. First, we identify general elements of B&B algorithms and describe their implementation as a finite state machine. Then, we introduce workers that autonomously cooperate using work stealing to allow parallel execution and full utilization of the target FPGA. Finally, we explore advantages of instance-specific designs that target a specific problem instance to improve performance.

We evaluate our concepts by applying them to a branch and bound problem, the reconstruction of corrupted AES keys obtained from cold-boot attacks. The evaluation shows that our work stealing approach is scalable with the available resources and provides speedups proportional to the number of workers. Instance-specific designs allow us to achieve an overall speedup of 47× compared to the fastest implementation of AES key reconstruction so far. Finally, we demonstrate how instance-specific designs can be generated just-in-time such that the provided speedups outweigh the additional time required for design synthesis.

CCS Concepts: • **Theory of computation → Branch-and-bound**; **Backtracking**; • **Computer systems organization → Reconfigurable computing**; • **Hardware → Hardware accelerators**; **Reconfigurable logic applications**; *Resource binding and sharing*; Finite state machines; Hardware-software codesign; • **Security and privacy →** *Cryptanalysis and other attacks*; Block and stream ciphers;

Additional Key Words and Phrases: Work stealing in hardware, instance-specific computing, FPGA, just-in-time synthesis, on-demand synthesis, cold-boot attacks, AES, key reconstruction, key schedule

**ACM Reference Format:**
Heinrich Riebler, Michael Lass, Robert Mittendorf, Thomas Löcke, and Christian Plessl. 2017. Efficient branch and bound on FPGAs using work stealing and instance-specific designs. ACM Trans. Reconfigurable Technol. Syst. 10, 3, Article 24 (June 2017), 23 pages.
DOI: http://dx.doi.org/10.1145/3053687

## 1. INTRODUCTION

In recent years, the growing importance of data analysis and processing has become omnipresent. To take advantage of the promises of Big Data, an efficient search for valid solutions within a large search space is a major benefit. One method of processing those search spaces is using Branch and Bound (B&B) algorithms. B&B algorithms organize

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and the European Union Seventh Framework Programme under Grant Agreement No. 610996 (SAVE). Calculations leading to the results presented here were performed on resources provided by the Paderborn Center for Parallel Computing and by the Maxeler program MAX-UP.
Authors' addresses: H. Riebler, M. Lass, R. Mittendorf, T. Löcke, and C. Plessl, Paderborn University, PC2, Warburger Str. 100, 33098 Paderborn, Germany; emails: {firstname.lastname}@uni-paderborn.de.

DOI: http://dx.doi.org/10.1145/3053687

the search space in a tree structure and try to eliminate infeasible solutions as early as possible by pruning unpromising subtrees.

In software, there are several techniques available to accelerate B&B algorithms. One technique 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*). 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* 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. Another opportunity for acceleration is the fact that the organization of the search (i.e., starting point or order of branches) 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. A custom-tailored, *instance-specific* tree structure for a given problem can therefore lead to a significant speedup compared to a general solution.

Branch and bound algorithms are not the typical class of problems that have been tackled in hardware, because they are usually control-driven and not data-driven. Hence, the computations within each tree level must be significantly complex to amortize the overheads of offloading them onto hardware. In our earlier work [24], we proposed a hardware design for a B&B problem: the reconstruction of corrupted AES keys required to perform cold-boot attacks. This kind of side-channel attack has the goal to obtain cryptographic key material from the main memory of a running system. Cold-boot attacks exploit the fact that DRAM contents decay slowly over time and therefore often can be recovered even after transplanting the memory module into another system. But as this procedure cuts the power to the memory module, the stored contents often contain bit errors. Due to the high number of key candidates, the reconstruction produces very large search trees and for high error rates finding the correct solution is practically infeasible in software. Hence, we implemented a hardware design for a Field Programmable Gate Array (FPGA). FPGAs have proven to be highly efficient in terms of area, power consumption and performance. As a key feature, they can be optimized not just for an application but even for a particular problem instance.

In this work, we show how both acceleration techniques, work stealing and instance-specific computing, can be applied to reconfigurable hardware for B&B problems. We start with our hardware design for AES key reconstruction and use it as a case study throughout this article. Although this design is non-instance-specific and processes the search tree sequentially with only one worker, it is the state-of-the-art in performance for AES key reconstruction. The main contributions of this article are:

(1) We identify general elements in our case study design that are required to implement B&B problems in hardware and abstract them in a finite state machine.
(2) Then, we extend this design to allow for multiple hardware workers that dynamically share and balance their load using work stealing.
(3) Using this design we explore the advantages of instance-specific computing on FPGAs by generating designs custom tailored to a specific problem instance. In contrast to related work, we also evaluate a just-in-time approach to instance-specific computing by generating and synthesizing custom hardware designs on-demand at runtime.
(4) We evaluate all of our approaches and compare each result to those obtained using similar techniques in software and show that even just-in-time hardware synthesis can lead to a significant speedup compared to a non-instance-specific approach.

The remainder of this work is structured as follows: Section 2 describes the background and related work. Section 3 presents our hardware design for processing search

trees using B&B. Afterwards we use work stealing in Section 4 and instance-specific computing in Section 5 as extensions of this design. Finally, we evaluate all contributions including the just-in-time approach in Section 6 and conclude in Section 8.

## 2. BACKGROUND AND RELATED WORK

In this section we describe the foundations and related work. We start with the main ideas of Branch and Bound (B&B), Work Stealing (WS), and instance-specific computing. We present similar approaches to ours using these techniques and highlight differences to our work. Then, we discuss the principles of cold-boot attacks and cryptographic key reconstruction.

### 2.1. Branch and Bound and Work Stealing

A tree is a well-known and widely used data structure to organize hierarchical data. A search tree is a special subtype used for locating values. The leaves represent possible solutions, while nodes represent intermediate steps toward these solutions. For many problems, the size of the search space is tremendous, resulting in very wide and deep tree structures. In this work, we focus on two common software techniques to process large trees and distribute the work to several workers: The first one, *branch and bound* [14], is an algorithmic design pattern that uses a tree to represent the search space of all possible solutions. The pattern consists of two aspects: The *branch* operation divides the search problem into multiple subproblems, represented as branches in the search tree. The *bound* operation prunes the search tree, eliminating subtrees that cannot lead to a valid solution. Depending on the problem domain and instance, one can implement custom variants of the *branch* and/or *bound* operation to construct the most promising search tree. Even after applying B&B, the trees are typically very large. Hence, a method to efficiently distribute and process the computation is also required. Therefore, we use *work stealing* [3], which is a well-known parallelization strategy for parallel computers. As a basic abstraction, every worker maintains a queue of work packages. If a worker has processed all its work packages and becomes idle, then it gathers or *steals* 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* 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). Hence, work stealing is a well-suited strategy for B&B applications.

With heterogeneous computing becoming widely adopted, WS and B&B have been studied for different architectures [1, 2, 16] and applications. While most of these works target multicore CPUs, newer efforts show results for heterogeneous architectures (CPU-GPU) or manycore CPUs (Intel MIC). Kestur et al. [12] use WS on FPGAs to solve the problem of matrix-vector multiplication, but do not describe their architecture in detail. Elangovan et al. [6] use WS in a parallel execution model based on OpenCL to distribute tasks across up to 4 GPUs to achieve efficient utilization. Ramanathan et al. [23] start also with an OpenCL WS implementation for GPUs but synthesize the code with the help of Altera'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. 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, we use a hardware implementation based on finite-state machines (FSMs).

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 WS. Lima et al. [15] present runtime task scheduling for GPUs based on WS 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. [20] 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. Melab et al. [18] target the Intel MIC platform, a many-core coprocessor architecture, and Kumar et al. [13] a combined platform featuring general-purpose ARM cores and special-purpose DSP cores, implementing work stealing between different cores.

### 2.2. Application- and Instance-Specific Computing

Initial work on instance-specific computing for reconfigurable hardware tackles the B&B algorithm for the Boolean Satisfiability Problem (SAT) [9, 29]. FPGAs are suitable for solving SAT problems, because the computations (evaluation of clauses) are highly parallelizable. Nonetheless, it would be impractical to fabricate an Application-Specific Integrated Circuit (ASIC) for each formula due to high development costs. Zhong et al. [30] 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), then the solution is found; if it is 0 (false), then the complement assignment is tried; and if it returns $x$ (undetermined), then the next FSM is activated. If the formula is unsatisfied for all assignments, then 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 [26] and Davis et al. [5] avoid instance-specific placement and routing and move toward 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.

Our application uses a similar implication circuit as described for SAT solvers or covering problems [22], 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 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 3.2.2), which requires global state information and considerable computation effort.

Kašík [11] 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 [17] 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.

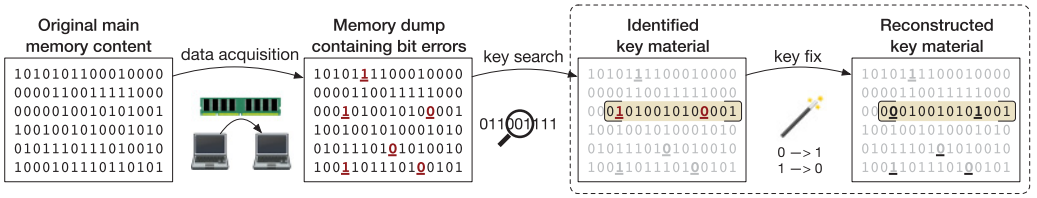


Fig. 1. Main steps of a cold-boot attack.

### 2.3. AES Key Reconstruction

In this article, we focus on the reconstruction of corrupted AES keys as a case study for B&B algorithms. AES [4] is a common symmetric encryption scheme that is used by a variety of systems and tools, for example, disk encryption, WLAN, HTTPS, SSH and VoIP.

Reconstructing corrupted keys is a necessary step in cold-boot attacks. These attacks exploit the fact that main memory contents are not immediately lost when power is cut off, but decay slowly instead. This allows gaining access to encrypted data if an attacker has physical access to a machine that has the corresponding secret keys in main memory (e.g., a computer with a locked screen). The procedure of performing a cold-boot attack is sketched in Figure 1. At first, the attacker obtains an image of the 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, then only a small fraction of the bits will have changed their value. The second step of the attack is to identify the key material in the obtained image (*key search*). Finally, the bit errors have to be corrected (*key fix*), which poses the algorithmic and computational challenge and is tackled in this work. If successful, then the secret keys can be recovered and, for example, a full disk encryption can be circumvented. More details on cold-boot attacks were presented by Halderman et al. [8].

As an AES secret key might be any (pseudo-) random bit string that matches the key size (128, 192, or 256 bit), a naive brute-force approach to the reconstruction of corrupted keys is impractical. However, AES performs several rounds during encryption and decryption, each of which uses a separate round key. The master key is used as the first round key from which all other round keys are derived. The *key schedule* is composed of the master key and the (key-size-dependent) number of other round keys. As the whole key schedule is required for every en- and decryption, it is typically kept in memory as a contiguous data block for performance reasons. Because the round keys are derived from the master key using a bijective function, they provide a great amount of redundancy. Figures 2 and 3 show how the 10 round keys in the AES-128 key schedule are generated from the master key. Each round consists of 16 bytes, divided into four words of four bytes each. The first word of each round key is computed by applying a complex operation 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 remaining words 1...3 of each round key are computed by applying a simple XOR operation between the same word of the previous round key and the previous word of the same round key. Using the key schedule and exploiting the redundancy, the search tree can be reduced considerably.

Kaplan and Geiger [10] were the first to propose exploiting the key schedule's structure to identify relevant memory segments. In our previous work, we already addressed the streaming-oriented key search [25] and an FSM-based key reconstruction [24] on FPGAs and achieved the fastest implementations so far. In this work, we improve

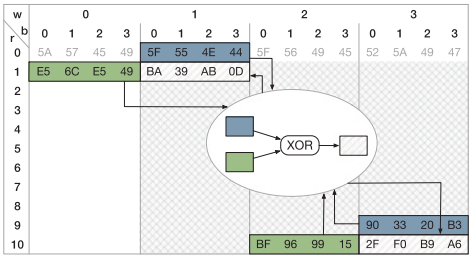


Fig. 2. Simple XOR operation for all but the first word in each round.

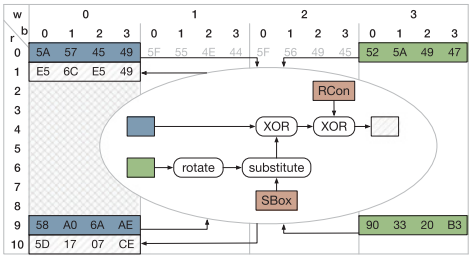


Fig. 3. Complex operation for first word in each round (w, word; b, byte; r, round).

the key reconstruction even more by using algorithmic optimization, work stealing, and instance-specific computing. Our reconstruction strategy is based on the idea of Tsow [27], which is an improved variant of the approach by Halderman et al. [8]. Although the following concepts can be applied for all AES key sizes with minor adaptations, we will describe them for AES-128 for simplicity.

## 3. BRANCH AND BOUND IN HARDWARE

In this section, we present a hardware design for processing search trees using branch and bound. We start with Tsow's reconstruction approach for AES keys in software as an example to highlight the general elements of B&B algorithms. Then we describe how B&B algorithms can be translated into a state machine suitable for implementation on FPGAs. We abstract away most of the details and generalize the concepts to common B&B problems. More details on the AES-specific implementation can be found in the original articles and our previous publication.

### 3.1. Case Study: AES Key Reconstruction

AES key reconstruction is based on traversing a large, highly unbalanced and dynamically growing search tree to find a valid solution. As discussed in the last section, the key material observed in the memory image acquired during a cold-boot attack probably contains bit errors due to memory decay. Hence, the secret key cannot be obtained immediately and key reconstruction techniques to correct bit errors are required.

The main idea of the reconstruction algorithm by Tsow is to sequentially guess the values of 16 specific byte positions in the AES key schedule, such that these values are compatible to the observed key schedule in the memory image. As a basic abstraction, the compatibility reflects the probability, that a guessed value (a *candidate* for the correct value) has decayed into the *observed* value. Table I shows one possible allocation of the 16 byte positions. Bytes $0_0 ... 15_0$ (emphasized in bold) are those for which values have to be guessed. Values for all other bytes can be derived from already fixed bytes following *implication chains*. The 16 byte positions are chosen such that they, together with the implications, determine the complete key schedule.

*3.1.1. Implication Chains.* All guessed values for bytes $0_0 ... 15_0$ are combined using the inherent structure of a key schedule (the key expansion function of AES depicted in Figures 2 and 3) to deduce values of other byte positions. Those *implied* bytes, denoted as $0_i ... 15_i$ for $i > 0$, form implication chains. For example, values for the byte $1_1$ in Table I can be implied after guessing values for bytes $0_0$ and $1_0$ using the complex operation shown in Figure 3. After additionally guessing the value for byte $2_0$, implication of the bytes $2_1$ and subsequently $2_2$ is possible. The number of implications depends on the byte position $X_0$ ($X = 0 \ldots 15$) and increases with each byte:

Table I. One Possible Path for Guessing Byte Values for AES-128

| 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

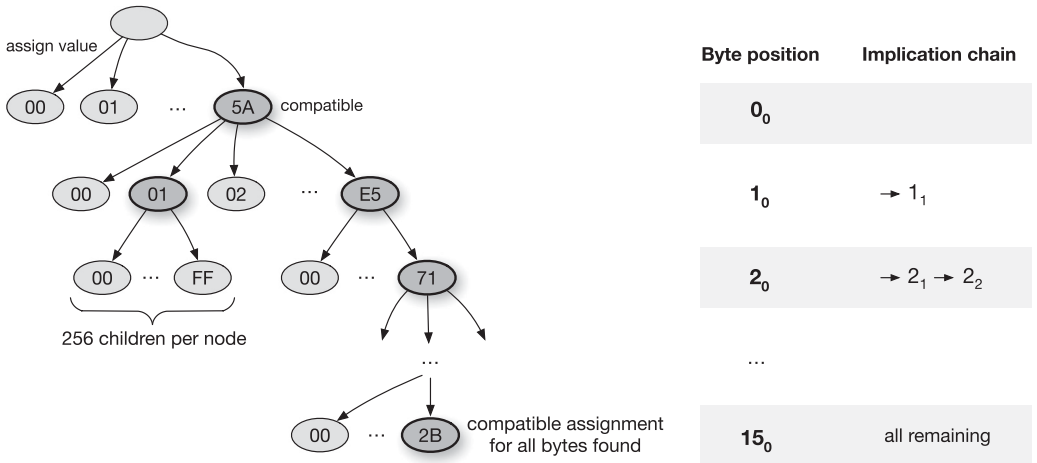


Fig. 4. Search tree for key reconstruction.

$$\texttt{number\_of\_implications\_for\_position}(X_0) = \begin{cases} X & \text{if } X < 11 \\ 10 & \text{if } 11 \leq X < 15 \\ 65 & \text{if } X = 15. \end{cases} \quad (1)$$

After all implications of the last byte position $15_0$ are computed, the entire data for round 8 (highlighted in Table I) of the key schedule is known. From a single complete round key, the entire key schedule and thus the actual key in round 0 can be recovered. This idea of guessing and implying byte values to sequentially complete the key schedule forms a search tree, similar to the trees described for B&B problems in Section 2.1.

*3.1.2. Search Tree.* The reconstruction algorithm traverses a search tree as shown in Figure 4 where in each level of the tree the value of one of 16 bytes $0_0$...$15_0$ of the key schedule is assigned. When guessing a single value for one of these bytes, all possible assignments `0x00...0xFF` are tested sequentially, such that the tree gets wider by a factor of 256 on each level. The large number of nodes makes it necessary to prune parts of the search tree to allow finding a solution in an acceptable amount of time. We use an *Error Model* to prune the search tree, eliminating assignments of bytes (or subtrees of the search tree) that cannot lead to the correct key schedule.

*3.1.3. Error Model.* As mentioned in Section 2.3, DRAM contents will gradually decay when powered off. Halderman [8] demonstrated the properties of memory decay and outlined a pattern: bit errors that flip to a certain ground state of the memory cell are dominant (99.9%) over the bit errors that flip to the opposite direction (0.1%). Hence, they ignore the unlikely flip direction and assume only bit errors in one direction, resulting in the *Perfect Asymmetric Decay* (PAD) error model. This model allows for efficient pruning but fails for certain problem instances, because in reality bit errors in both directions are possible. To generalize Halderman's approach, we proposed a threshold-based method, which takes both error directions into account [24, 28]. Our *Expected Value as Threshold* (EVT) model requires two error rates, $r_{1\to0}$ (bitflips from 1 to 0) and $r_{0\to1}$ (bitflips from 0 to 1). These error rates represent the number of bit errors introduced by memory decay in the dominant and opposite direction. With given $r_{1\to0}$ and $r_{0\to1}$, we can compute the expected number of bitflips in each direction by multiplying them with the total number of bits $N$ in the full key schedule.

We use the EVT model to check if a candidate assignment can lead to a valid solution or can be discarded. Therefore, the error model checks the *compatibility* of the guessed and implied values against the observed key schedule from the memory image. We compute the number of bits that would have flipped from 1 to 0 and 0 to 1 and compare them against the number of expected bit errors. If one of the numbers, denoted by $n_{1\to0}$ and $n_{0\to1}$, exceeds its expected value, then the candidate is not compatible:

$$\texttt{candidate\_is\_compatible}(\mathtt{n}_{1\to0}, \mathtt{n}_{0\to1}) = (n_{1\to0} < r_{1\to0} \cdot N) \wedge (n_{0\to1} < r_{0\to1} \cdot N). \quad (2)$$

The compatible candidates for our example are colored darker in Figure 4. If a candidate assignment for a byte is compatible, then the search proceeds in the next tree level. If an assignment for all 16 guessed bytes is found (such that the error bound is not exceeded), then the whole key schedule can be derived and a final compatibility check determines if a valid solution was found.

Note that from a statistical point of view, the search should not be bounded when the number of bitflips exceeds the expected number of bitflips, 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 the next subsection we describe the implementation of the AES key reconstruction as a B&B algorithm in hardware.

### 3.2. State Machine for Branch and Bound

The first challenge is to translate a B&B algorithm into a hardware design. As B&B algorithms always consist of the same general elements, we are able to design a finite state machine (FSM, see Figure 5) that represents the general elements in FSM states. The states condensed under `BRANCH` are responsible for iterating through all possible branches (state `CHOOSE_NEXT_BRANCH`) and inferring further knowledge (state `INFER_KNOWLEDGE`). In our case study, the branches correspond to all possible values of a byte ($0 \ldots 255$) and we infer other byte values from previously guessed values using implication chains. Next, the state `CHECK_BOUND` is responsible for pruning the search space, going back to the `CHOOSE_NEXT_BRANCH` state and therefore the next branch if the currently considered branch cannot lead to a valid solution. The bound operation is highly application specific, could be an upper or lower threshold, or an error model as in our case. If all possible branches for the current level have been examined and none could hold the bound constraints, then `STATE_POP` is entered to go back one level in the tree (*backtracking*). If `CHECK_BOUND` assesses the currently considered branch as valid, then `STATE_PUSH` is entered to store the context, step into this branch and therefore enter a deeper level of the tree. These states are condensed under `CHECKPOINTING`. They correspond to an ascent or descent in the tree, keeping track of the current state, that

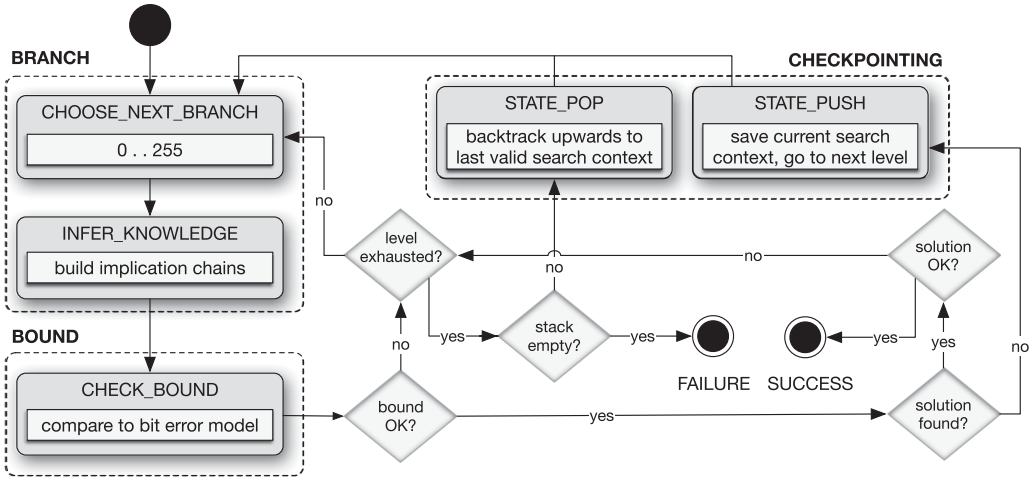


Fig. 5. General elements of a FSM that implements the B&B design paradigm.

is, the current level of the tree (0 . . . 15) and the next branch to consider (0 . . . 255). Checkpointing can be realized with a Last-In-First-Out (LIFO) data structure, that is, a stack.

In our application the steps performed to infer knowledge and to check the bound are different for every tree level. The implication chains depend on the set of bytes that already have values assigned and the bound is applied to more and more bytes in each level. For that reason, in our implementation there are 16 different `INFER_KNOWLEDGE` and `CHECK_BOUND` states, one for each level.

*3.2.1. Checkpointing Tree Traversal.* The checkpoints created on the stack after a successful compatibility check are used for backtracking as soon as all candidates have been tried and the level is exhausted. A checkpoint needs to contain all information that is required by the `CHOOSE_NEXT_BRANCH` and `CHECK_BOUND` states. These are 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 derived 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.

*3.2.2. Maintaining Bound.* The bound is used to prune the search tree. In our application it is defined as checking the compatibility of the guessed and implied byte values against the decayed key schedule, that is, if the byte values could have decayed to the observed key schedule in memory according to the error model. The state `CHECK_BOUND` of the FSM in Figure 5 is responsible to check this property after each guess and its corresponding implications. The total number of errors is added up using adder trees and stored in registers.

*3.2.3. Selecting Branches.* B&B search algorithms can traverse subtrees in various orders, for example, using a static order or some heuristic. Optimally, a heuristic should pick subtrees first that promise the best solution or the highest chance for a valid solution. The state `CHOOSE_NEXT_BRANCH` of the FSM in Figure 5 selects and sets the next value and therefore determines the taken branch.

Table II. Calculation of the Decay Probability from a Candidate Byte 0x9C to the Observed Byte 0x8A for All Possible Flip Types

| | from candidate byte 0x9C | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **flip type** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | **count** | **probability** |
| $0 \to 0$ | | ✓ | ✓ | | | | | ✓ | 3 | $1 - p_{0\to1}$ |
| $0 \to 1$ | | | | | | | ✓ | | 1 | $p_{0\to1}$ |
| $1 \to 0$ | | | | ✓ | | ✓ | | | 2 | $p_{1\to0}$ |
| $1 \to 1$ | ✓ | | | | ✓ | | | | 2 | $1 - p_{1\to0}$ |
| | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | |
| | to observed byte 0x8A | | | | | | | | | |

Compared to our previous work on AES key reconstruction we introduce an optimized heuristic for the branching order. Instead of statically incrementing the candidate value for a byte from 0 to 255, we reorder the values individually on each level to visit the most promising branches of our search tree first. The candidate 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. In our error model, the error rates $r_{1\to0}$ and $r_{0\to1}$ determine the total fraction of bits that have flipped in one direction. Because in a key schedule ones and zeros are expected to be equally frequent and represent half of the bits each, we can assess the probability of 1 flipping towards 0 ($p_{1\to0}$) and the probability of 0 flipping towards 1 ($p_{0\to1}$) as

$$r_{1\to0} \cdot N = \frac{1}{2} N \cdot p_{1\to0}, \qquad p_{1\to0} = 2 \cdot r_{1\to0}, \tag{3}$$

$$r_{0\to1} \cdot N = \frac{1}{2} N \cdot p_{0\to1}, \qquad p_{0\to1} = 2 \cdot r_{0\to1}. \tag{4}$$

As an example for our branching heuristic, we examine the observed byte value $\texttt{0x8A} = 10001010_2$ and the candidate byte value $\texttt{0x9C} = 10011100_2$. As shown in Table II, for such a decay 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. Assuming the probabilities $p_{0\to1} = 0.002$ and $p_{1\to0} = 0.598$ are given, we can assess the probability of 0x9C decaying to 0x8A as

$$\begin{aligned} \texttt{probability\_of\_decay}(\texttt{0x9C}, \texttt{0x8A}) &= \prod_{\text{flip type } \gamma} \texttt{probability}(\gamma)^{\texttt{count}(\gamma)} \\ &= (1 - p_{0\to1})^3 \cdot {p_{0\to1}}^1 \cdot {p_{1\to0}}^2 \cdot (1 - p_{1\to0})^2 \\ &= 0.998^3 \cdot 0.002^1 \cdot 0.598^2 \cdot 0.402^2 \\ &= 1.149 \cdot 10^{-4}. \end{aligned} \tag{5}$$

This calculation has to be performed for all the observed values of the 16 bytes represented in the search tree and for all 256 possible candidate byte values. With a total of 4,096 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 bytes based on the calculated decay probabilities beforehand in software. When starting 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 block RAM. The CHOOSE_NEXT_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 block RAM.

```
1  if COUNT(self.q) > 0 :                # Work package available on own deque q, so
2    PROCESS(self.q.TOS)                 # load it and enter CHOOSE_NEXT_BRANCH
3  else :
4    if ( COUNT(victim.q) == 1 and victim.state == STATE_POP ) or
5         COUNT(victim.q) == 0 :         # No work packages available to steal or stealing
6      idleCannotSteal <- true           # would interfere with victim's stack access
7    else :
8      idleCannotSteal <- false
9      self.q.BOS <- victim.q.BOS
10     self.q.TOS <- victim.q.BOS + 1
11     COPY_WORKPACKAGE(FROM=victim.q.BOS, TO=self.q.BOS)
12     victim.q.BOS <- victim.q.BOS + 1
13     PROCESS(self.q.TOS)
```

Listing. 1. Backtracking with work stealing in STATE_POP.

## 4. WORK STEALING IN HARDWARE

In this section we describe the extension of our B&B design to allow parallelization of work. We were able to transform our existing hardware design to use a work stealing approach where several FSMs work in parallel on different parts of the search tree and autonomously obtain work packages from other workers when becoming idle. In the following section, we describe the necessary changes and considerations in detail.

### 4.1. Transformation of the Branch and Bound State Machine

In our work stealing design, the original FSM as well as its state stack are duplicated $N_w$ times, where $N_w$ is the number of workers, so each FSM acts as one worker.

The state stack in its previous version contained information on which parts of the search space were already visited or could be discarded. As we have custom states for each reconstruction level, the state of the FSM determines in which level of the tree the worker is currently located. It can therefore be interpreted as a Top-of-Stack (`TOS`) pointer for the state stack. To coordinate tree traversal between different workers an additional Bottom-of-Stack (`BOS`) pointer is introduced, which marks the level of the tree up to which workers will backtrack when no solution is found. This prevents workers from entering parts of the search tree that are examined by other workers. By introducing this `BOS` pointer, the stack becomes a double-ended queue (deque).

We define each entry on the stack 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 stack is equivalent to creating a new child work package and beginning to process it.

Listing 1 shows backtracking of a worker entering `STATE_POP`. If `BOS` and `TOS` of its deque `q` are identical, then the worker has explored his whole search subspace and needs to steal work from another worker. 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 11) and incrementing the victim's `BOS` by one (line 12). The `BOS` and `TOS` of the thief are set to the recently stolen work package (lines 9 and 10). Moving the `BOS` of the victim ensures that it will not enter the search subspace that is now explored by the thief. The whole procedure of stealing work is also shown in Figure 6. Work packages located low on the stack represent large subtrees and therefore tend to contain more work than packages located higher on the stack.

### 4.2. Coordination and Synchronization

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

A stealing victim can be chosen uniformly at random among all available FSMs, which is done in most software implementations and is known to be efficient [3]. In

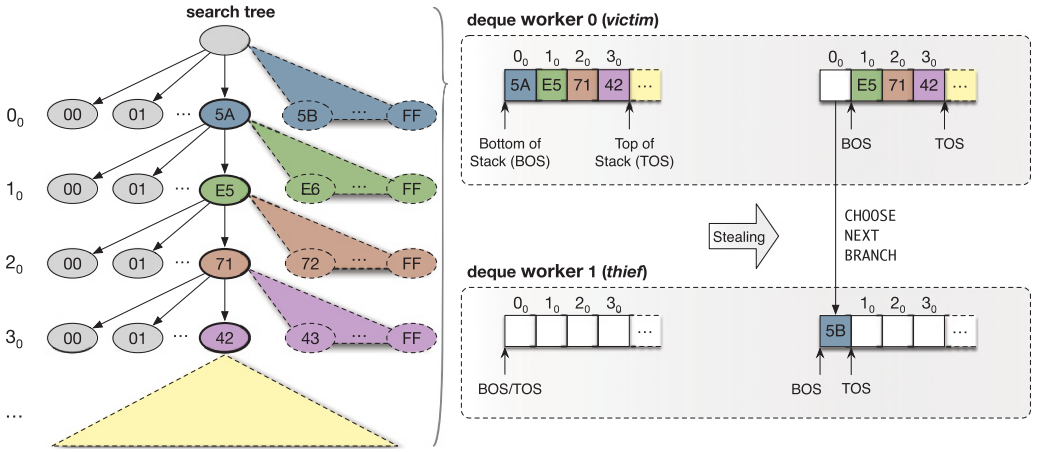


Fig. 6. Stealing work by copying the bottommost stack entry.

recent work, also non-randomized approaches of work stealing have been used to better fit the underlying topology. Kumar et al. [13] implement work stealing between ARM and DSP cores, preferring stealing victims of the same processor type. Guo et al. [7] implement locality-aware work stealing, preferring victims that share the same L2 cache. Min et al. [19] extend this concept to a hierarchical model of the memory topology.

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 stack. In our application, an entry on the stack contains 176 bytes holding the key schedule and additional bytes for previously calculated error rates. Allowing to steal from all workers would require a large number of multiplexers, which would further increase 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: The worker $w_x$ only steals work packages from its direct neighbor $w_{(x-1 \mod N_w)}$, that is, workers are arranged in a ring topology.

This restriction simplifies synchronization of stack accesses between different workers, as only the worker itself and one of its direct neighbors can access its stack. During normal operation a worker only changes the `TOS` of its own stack (line 2 of Listing 1) and a thief only the `BOS` of its victim's stack (line 12); therefore, concurrent access is possible in general. However, if the victim is in `STATE_POP` to fetch a work package and it is the only one left on the stack, stealing would lead to a conflict, and therefore stealing is postponed until the next clock cycle (line 4).

Apart from the conserved synchronization effort, the ring structure allows us 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 us to add additional workers with only constant additional resource utilization.

### 4.3. Initialization and Termination

The initialization and termination of our FSM require further considerations. On initialization the whole search space is assigned to the worker $w_0$. During processing, the worker spawns new work packages by pushing its state to the state stack, so its

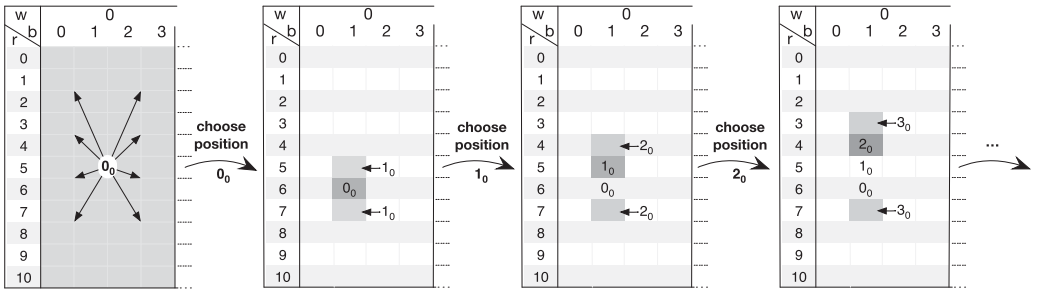


Fig. 7. Generation of a reconstruction path (first four levels).

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.

For the termination there are two possible conditions: either one worker has found a valid solution or all workers are idle and none of them has found a valid solution. In the first case, the corresponding FSM enters the `SUCCESS` state (see Figure 5) and its result gets transferred 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, 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 6 of Listing 1). As soon as this signal is set by all workers, worker $w_0$ enters the `FAILURE` state and the host computer is informed that no solution was found.

## 5. INSTANCE-SPECIFIC COMPUTING IN HARDWARE

The structure of the search tree can heavily impact the time required to find a valid solution. Depending on the B&B algorithm, different trees may contain different elements, elements may be arranged differently, or the number of children per node may vary. In this section, we describe how different search trees can be dynamically constructed to improve the search process. We start with concepts implied from our problem, the AES key reconstruction, how the most promising tree structure can be chosen for a particular problem instance, and how we adapt our hardware design to any of those search trees automatically, resulting in instance-specific hardware designs.

### 5.1. Instance-Specific Search Trees for AES Key Reconstruction

The search tree in our application depends on the path that is followed during reconstruction and is determined by the selection and order of byte positions to guess values for. The path shown earlier in Table I of Section 3 is one of many possible ways to reconstruct a key schedule. In the following, we discuss how different paths can be built and how a path can be chosen to efficiently solve a particular problem instance.

*5.1.1. Generating Reconstruction Paths.* To minimize the 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. This ensures that the path is of minimal length and only values for 16 bytes have to be guessed. Also, it is preferable to choose a path 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 [27] proposed a method to construct paths that fulfill these criteria.

The procedure is sketched in Figure 7, where possible choices are highlighted. The path construction starts with an arbitrarily chosen position from the first four bytes (the first word) of any round key for which a value is to be guessed at level 0. Starting

from this byte position, the bytes to be guessed at levels 1 to 10 are chosen from the adjacent positions in the same column, because this allows for inferring one additional byte using the *complex* rule of the AES key expansion scheme. There are at most two possible choices for each of those levels: either the one on top or the one below all previously chosen byte positions. After the first 11 levels have been selected, the byte positions for the remaining levels 11...15 are fixed. All of the paths built upon this strategy result in the reconstruction of a complete round key for round 8, like the example shown in Table I of Section 3. The missing values of the key schedule can be derived from this complete round key for a final compatibility check.

*5.1.2. Number of Possible Paths.* 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 $4 \cdot \sum_{k=0}^{10} \binom{10}{k} = 4{,}096$ choices. So overall, 4,096 different search paths can be constructed using Tsow's approach.

*5.1.3. Path Selection Heuristic.* As discussed in the beginning of this section, the selection of the path may effect the runtime dramatically. Thus, Tsow also proposed a heuristic to choose the most promising path among a set of paths. His approach, which 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.

The PAD error model only accounts for flips into a single direction, for example, 1 to 0. With this property, all bits with value 1 can immediately be assessed as not flipped and are therefore called *known bits*. During key reconstruction the search is bounded whenever a byte value is guessed which conflicts with a known bit. As early bounds imply a shorter runtime of the algorithm, Tsows heuristic tries to maximize the number of known bits in 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 with the most known bits from the first four bytes of all round keys in the observed key schedule. While selecting either the upper or lower next byte position for rounds 1...10, the heuristic chooses the byte position with the most known bits among these two; see Figure 7.

Wang [28] extended this heuristic to his EVT error model by replacing the known bit criterion with a *dominant bit* criterion to encounter possible bit flips in both directions. The dominant bit is chosen as

$$\texttt{dominant\_bit} = \begin{cases} 1 & \text{if } r_{1\to 0} > r_{0\to 1} \\ 0 & \text{otherwise} \end{cases}, \tag{6}$$

where $r_{1\to 0}$ and $r_{0\to 1}$ determine the total fraction of bits that have flipped toward 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.

### 5.2. Generation of Instance-Specific Hardware Designs

Each of the paths 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 paths but also the partitioning of all bytes of the key schedule into guessed and implied bytes.

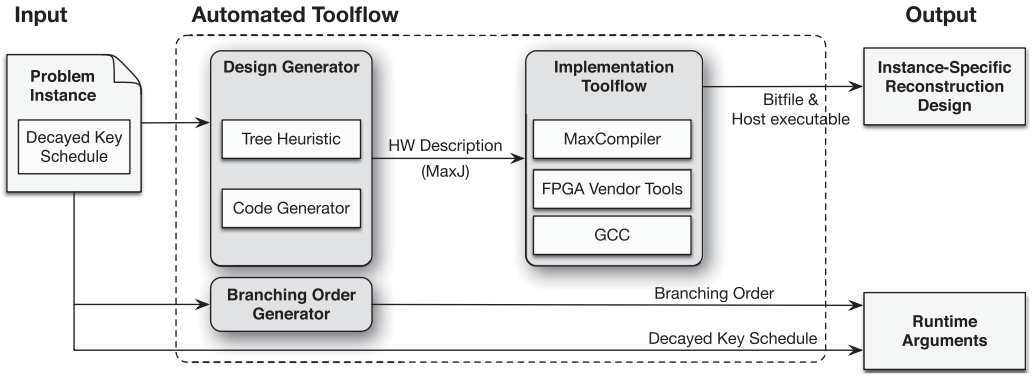


Fig. 8. Toolflow of our instance-specific design approach.

Depending on the chosen path, the implied bytes may be derived from values of varying sets of other bytes, requiring the implementation of different implication chains for all positions in the key schedule. Moreover, for other B&B algorithms, the different search trees may also differ in their structure, for example, having a different number of children per node. This high variability makes the implementation of a universal hardware design challenging. To support different search trees, we developed a generator for hardware designs instead. It translates a description of the tree into code for a custom FSM suitable for the Maxeler toolflow [21]. The overall toolflow is sketched in Figure 8.

The first step in creating an instance-specific circuit is to analyze the problem instance by a custom software that generates a search tree. For our application, this means using Tsow's heuristic to build a reconstruction path. Based on the tree structure, our design generator creates domain-specific code (MaxJ) and the Maxeler tools transform this code into a hardware design, which is further processed by FPGA vendor tools to generate an FPGA configuration. Additionally, software for the host computer is generated and compiled. The design generator builds custom instances of the states `CHOOSE_NEXT_BRANCH` and `INFER_KNOWLEDGE` for each node in the tree. The order in which children are visited is read from a ROM, allowing us to modify the branching order without synthesizing a new design.

For our application, the AES key reconstruction, the generator also contains application-specific logic. Although the path 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 cycles. If the *complex* rule of the AES key expansion scheme (see Figure 3 in Section 2.3) is required to infer another byte value, then a ROM access is used for the `SBox` operation and additional clock cycles are introduced to access data stored in the ROM.

The generated hardware design requires the decayed key schedule and the order of candidate values as runtime inputs. The order of candidate values, that is, the branching order, is generated by a separate tool as described in Section 3.2.3.

## 6. EVALUATION

We have presented three approaches to accelerate the execution of a B&B algorithm on an FPGA, namely an algorithmic change to visit most promising branches first, parallel

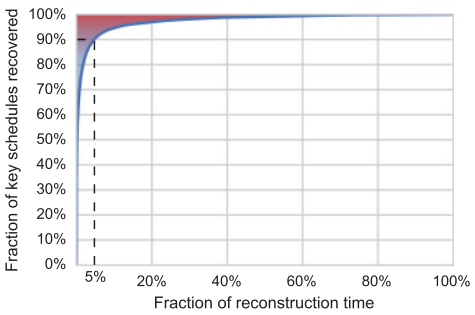


Fig. 9. Few key schedules dominate the runtime.

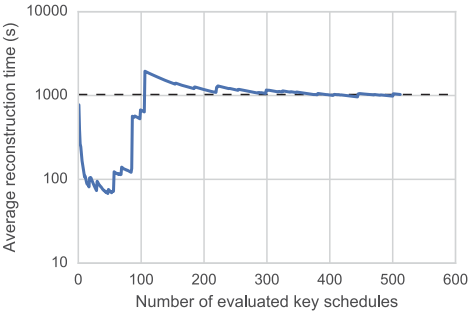


Fig. 10. 512 key schedules provides stable results.

execution using work stealing, and generation of instance-specific hardware designs. We applied these techniques onto a hardware design used for the reconstruction of AES keys. In this section, we evaluate our proposed concepts by assessing their impact on the total time required for the key reconstruction as part of cold-boot attacks.

### 6.1. Evaluation Scenario

As described in our previous work [24], the number of key schedule candidates obtained from a cold-boot attack can be in the order of 10,000. We assume a decay according to the EVT model of $r_{1\to0} = 0.299$ and $r_{0\to1} = 0.001$, leading to a total error rate of 30%. This total error rate is higher than all cases studied in previous work but still allows extensive evaluation. We focus on particularly hard reconstruction problems to show the potential of our different strategies. Choosing even higher error rates would render the comparison to our baseline as impracticable, because reconstruction of some key schedules would take several weeks. All key schedules used in this evaluation were created by first generating a key from 16 random bytes, expanding it to a full 176 bytes key schedule using the AES-128 key expansion and, finally, randomly flipping single bits to simulate a decay with the considered error rate.

The run time required for the reconstruction of a large number of key schedules is very unevenly spread over the single key schedules. Figure 9 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.

Due to the high error rates, an evaluation based on 10,000 key schedules would have been impractical. 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, we observed the average reconstruction time for a single key schedule depending on the number of evaluated key schedules. Figure 10 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, so we used the generated set of 512 key schedules throughout this evaluation.

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

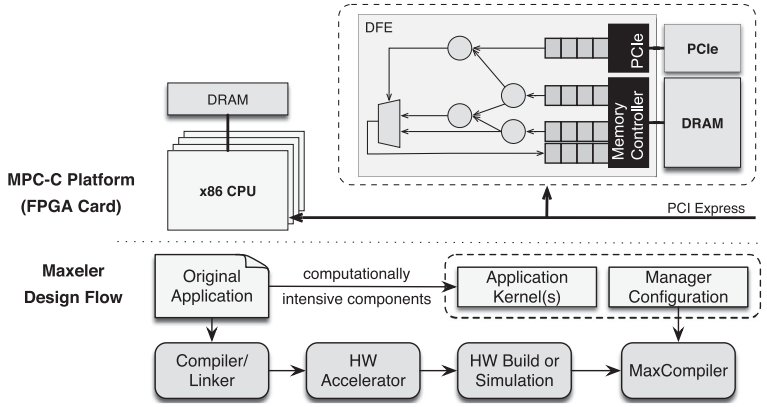


Fig. 11. Maxeler MPC-C platform architecture and design flow.

### 6.2. Target Platform: 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 and the corresponding programming model are sketched in Figure 11. The system is a two-socket Intel Xeon server system with an FPGA accelerator card connected via PCIe. This FPGA card features a Xilinx Virtex-6 FPGA (XC6VSX475T) and 24GB of on-board SDRAM memory. The host uses two 6-core (12 threads) X5650 CPUs running at 2.67GHz and provides 48GB RAM.

The design flow for mapping applications is based on the MaxCompiler [21] programming model. It is mainly driven by Java and offers predefined APIs for specifying *data flow engines* (DFEs). Each DFE comprises one or more *kernels*, which implement the application logic and a *manager* that controls the routing of data streams between kernels, the CPU, and off-chip memory. MaxCompiler is able to take care of type conversions and can automatically perform optimizations like retiming, buffer-size optimization, and pipelining. Furthermore, it also offers an API to describe FSMs that can control memory streams and data paths for applications that cannot be expressed as a simple streaming data path. The specified kernel and manager code is translated into a hardware design that is further processed by FPGA vendor tools to generate a configuration bitfile. Afterwards, the bitfile is merged with additional information into a *maxfile* and linked with the original CPU application.

### 6.3. Results

To measure the impact of our strategies, we incrementally added the proposed features to our existing Maxeler DFE design, which is the baseline for our comparison. The measured runtimes to reconstruct all 512 key schedules are shown in Table III. To verify the observed speedups we repeated the measurements using software implementations. As an evaluation of the software implementation on our target platform would have taken too long, we performed it on a computing cluster, using up to 50 compute nodes in parallel. Each node contains an Intel Xeon E5-2670 CPU with 16 cores.

Using the baseline hardware implementation for the reconstruction of 10,000 keys would take 10,280,000s, which corresponds to nearly four months. The software solution is nearly ten times slower and therefore would take more than three years.

Figure 12 shows the observed speedups for our evaluated techniques both in hardware and in software. Our first optimization, reordering the branches based on the specific key schedule that is being reconstructed, leads to a 1.5× speedup. This speedup is consistent between the hardware and software implementations. The B&B and

Table III. Runtime (s) of the Reconstruction of 512 Key Schedules with Different Techniques

| | feature set | number of workers | sum | average | standard deviation | total speedup |
|---|---|---|---|---|---|---|
| Hardware | HW Baseline: Fastest currently available implementation | 1 | 526,453 | 1,028 | 7,271 | 1.0 |
| | Optimized branching order | 1 | 351,272 | 686 | 4,679 | 1.5 |
| | Parallelization using work stealing | 4 | 112,605 | 220 | 1,522 | 4.7 |
| | Instance-specific design following heuristically chosen reconstruction path | 4 | 11,214 | 22 | 142 | 46.9 |
| | Just-in-time synthesis of instance-specific designs (based on Equation 7) | 4 | 46.488 | 91 | 270 | 11.4 |
| Software | SW Baseline | 1 | 5,399,100 | 10,545 | 65,056 | 1.0 |
| | Optimized branching order | 1 | 3,502,560 | 6,841 | 40,853 | 1.5 |
| | Parallelization using work stealing | 4 | 984,943 | 1,924 | 11,674 | 5.5 |
| | —— " —— | 16 | 236,768 | 462 | 2,787 | 22.8 |
| | Following heuristically chosen reconstruction path | 4 | 90,663 | 177 | 1,076 | 59.6 |
| | —— " —— | 16 | 18,952 | 37 | 217 | 284.9 |

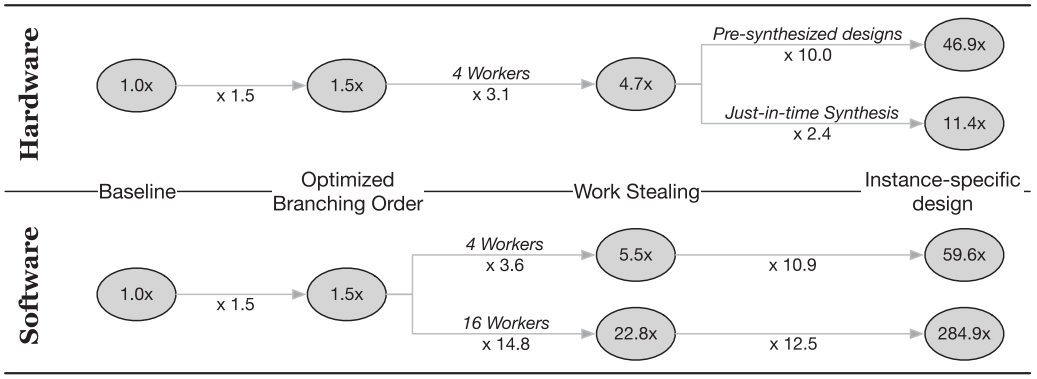


Fig. 12. Visualization of speedups achieved by the different techniques in Table III.

Table IV. Synthesis Results for Different Numbers of Workers

| number of workers | utilization * [%] | clock [MHz] max | clock [MHz] used | single worker equiv. max | single worker equiv. used | used in eval. |
|---|---|---|---|---|---|---|
| 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

instance-specific approaches which are the main focus of this article are discussed in the following sections.

*6.3.1. Work Stealing.* 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 highest achievable clock rate for different numbers of workers, shown in Table IV. 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 are required for each additional worker.

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 80MHz for our single-worker design and 60MHz for our four-worker design. With these clock rates synthesis times are below 2.5h per design.

Corresponding to the reduction in clock rate, we expect a 3× speedup for our design. We observed a slightly higher speedup of 3.1× over a single worker. Combined with the optimized branching order (1.5× speedup), this leads to a total speedup of 4.7×.

The software implementation used as a reference utilizes the Intel Cilk Plus[2] framework for implementation of the work stealing approach. During execution the process is bound to four CPU cores using the `taskset` utility. The 3.6× speedup observed in software is below the 4× speedup that should be reached using four CPU cores. This is likely to be caused by additional overhead for thread synchronization in software, whereas our hardware design introduces little additional overhead. 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.

We used the software implementation to determine the scalability of work stealing with respect to our B&B implementation. Using all the available 16 CPU cores, we achieve a speedup of 14.8× compared to the single-threaded implementation. This shows the potential of using work stealing in B&B applications in general and for AES key reconstruction.

*6.3.2. Instance-Specific Design.* For the PAD error model it was already shown by Tsow [27] that using a heuristically chosen reconstruction path 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 10.0× speedup on our hardware design. Combined with the aforementioned techniques a total speedup of 46.9× compared to our baseline implementation 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 61h, instead of the 4 months needed when using the baseline implementation. To validate these results we also implemented the adapted heuristic in software and observed a slightly higher speedup of 10.9×. For 16 workers, the instance-specific solution even achieves a speedup of 12.5×, showing that the potential benefit of using the instance-specific approach increases for a higher number of workers.

The presented numbers 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 paths and therefore hardware designs is limited to 4,096 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.

### 6.4. Just-in-Time Synthesis

Until this point we assumed that all instance-specific designs are pre-synthesized and directly available. We now evaluate whether dynamically generating these designs just-in-time (JIT) can still lead to a significant speedup of our application. The first thing to note here is that a previously generated hardware design can be reused for

[2]https://software.intel.com/en-us/intel-cilk-plus.

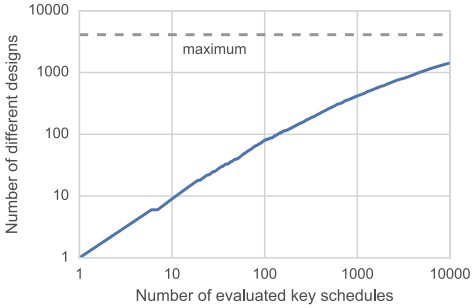


Fig. 13. Reuse of designs for other instances.

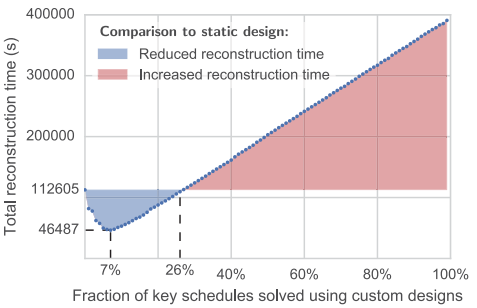


Fig. 14. Reconstruction time using JIT synthesis.

different key schedules if the heuristic chooses the same path for reconstruction. In fact, when evaluating the heuristically chosen paths for reconstruction of 10,000 key schedules only about 1,500 different paths were generated as shown in Figure 13. The key schedules for this evaluation were generated as described in Section 6.1.

*6.4.1. Strategy.* Synthesis of our custom hardware design takes about 2.5h 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 $^{2.5}/_{12}$h when running 12 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 path in Table I) and aborting those reconstructions that run longer than a certain threshold. For these hard key schedules an instance-specific design is generated 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.

*6.4.2. Evaluation.* We denote $p$ as the percentage of key schedules for which we build instance-specific designs. 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

$$\texttt{amortized\_reconstruction\_time} = \begin{cases} \frac{t_{\text{gen}}}{\#\text{DFEs}} & \text{if } t_{\text{gen}} < t_{\text{thres}} \\ \frac{t_{\text{thres}}+t_{\text{IS}}}{\#\text{DFEs}} + \frac{t_{\text{syn}}}{\#\text{CPU Cores}} & \text{otherwise} \end{cases}. \quad (7)$$

Figure 14 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 DFE. It shows that just-in-time 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 implementation, this corresponds to a total speedup of 11.4×. 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 *a posteriori* knowledge about the reconstruction times of our key schedules it shows that a significant speedup is possible by generating instance-specific hardware designs just-in-time, if the problems show a strong variation in difficulty. Also, it shows that a wide range of values for $p$ (0–26%) leads to a positive speedup.

Overall, the evaluation showed that each of our proposed techniques leads to a significant speedup over the baseline implementation, which was the fastest known hardware implementation. Assuming pre-synthesized instance specific designs, we achieved a total speedup of 46.9× combining those techniques. Considering a dynamic just-in-time generation of these designs still leads to a total speedup of 11.4×.

## 7. SUPPLEMENTAL MATERIAL

The source code of our software and hardware implementations as well as all input data used in our evaluation are available online at https://github.com/pc2/coldboot.

## 8. CONCLUSION

In this article, we have presented the realization of branch and bound (B&B) algorithms as well as two acceleration strategies (work stealing and instance-specific design) on FPGAs. B&B algorithms are used to efficiently process huge data spaces organized as search trees by eliminating infeasible solutions as early as possible. Although these algorithms are not the typical class of problems that have been tackled in hardware, we have shown in earlier work [24] that (if the computation within each tree level is significantly complex) the overheads of offloading can be amortized and significant speedups can be achieved. The hardware design from our previous work examines the reconstruction of corrupted AES keys as part of cold-boot attacks [8]. We started with this hardware design and used it as a case study throughout this article. Although this design processes the search tree sequentially with only one worker and is non-instance-specific, it is the state-of-the-art in performance for AES key reconstruction.

Based on this work, we identified general elements that are required to implement B&B problems in hardware and abstracted them into a finite state machine (FSM). Then, we extended the FSM to allow for multiple hardware workers that autonomously share and balance the computation using work stealing. Compared to our previously published hardware design, which serves as a baseline, we achieved a speedup of about 4.7× using a general hardware design, which utilizes four hardware workers on one FPGA. Our evaluation shows that speedups proportional to the number of workers can be expected. 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 custom tailored to a specific problem instance. Using instance-specific designs we achieved a total speedup of 46.9× although in this case all instance-specific hardware designs had to be pre-synthesized. Finally, we also evaluated a just-in-time approach for instance-specific computing by generating and synthesizing custom hardware designs on-demand at runtime. We showed that even just-in-time generated hardware designs can amortize the synthesis times and lead to speedups over a non-instance-specific approach. In our application, we achieve a total speedup of 11.4×.

The approaches shown in this article have been described based on AES key reconstruction but with general B&B algorithms in mind. Hence, they are easily adaptable to other B&B problems. Our evaluation demonstrates that using the presented techniques can lead to higher resource utilization and significant speedups compared to general designs using a single hardware worker.

## REFERENCES


[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput.: Pract. Exper.* 23, 2 (2011), 187–198.

[2] A. Tarun Beri, B. Sorav Bansal, and C. Subodh Kumar. 2015. Locality aware work-stealing based scheduling in hybrid CPU-GPU clusters. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'15)*. 48.

[3] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5, 720–748.

[4] Joan Daemen and Vincent Rijmen. 2000. The block cipher rijndael. In *Proceedings of the International Conference on Smart Card Research and Applications (CARDIS'00)*. Springer, 277–284.

[5] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. 2008. A practical reconfigurable hardware accelerator for boolean satisfiability solvers. In *Proceedings of the Design Automation Conference (DAC'08)*. ACM, 780–785.

[6] Vinoth Krishnan Elangovan, Rosa.M. Badia, and Eduard Ayguad. 2014. Scalability and parallel execution of OmpSs-OpenCL tasks on heterogeneous CPU-GPU environment. In *Proceedings of the International Conference on Supercomputing (ISC'14)*. 141–155.

[7] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS'10)*. 1–12.

[8] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM* 52, 5, 91–98.

[9] Youssef Hamadi and David Merceron. 1997. Reconfigurable architectures: A new vision for optimization problems. In *Principles and Practice of Constraint Programming—CP97*. Springer, 209–221.

[10] Brian Kaplan and Matthew Geiger. 2007. *RAM Is Key: Extracting Disk Encryption Keys from Volatile Memory*. Master's thesis. Carnegie Mellon University.

[11] Vladimir Kašík. 2010. Acceleration of backtracking algorithm with FPGA. In *Proceedings of the International Conference on Applied Electronics (AE'10)*. IEEE, 1–4.

[12] Srinidhi Kestur, John D. Davis, and Eric S. Chung. 2012. Towards a universal FPGA matrix-vector multiplication architecture. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*. IEEE Computer Society, 9–16.

[13] V. Kumar, A. Sbrlea, A. Jayaraj, Z. Budimli, D. Majeti, and V. Sarkar. 2015. Heterogeneous work-stealing across CPU and DSP cores. In *Proceedings of the High Performance Extreme Computing Conference (HPEC'15)*. 1–6.

[14] A. H. Land and A. G. Doig. 1960. An automatic method of solving discrete programming problems. *Econometrica* 28, 3 (1960), 497–520.

[15] J. V. F. Lima, T. Gautier, N. Maillard, and V. Danjean. 2012. Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In *Proceedings of the IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12)*. 75–82.

[16] João V. F. Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin, and Nicolas Maillard. 2015. Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures. *Parallel Comput.* 44 (2015), 37–52.

[17] Pavlos Malakonakis and Apostolos Dollas. 2011. Exploitation of parallel search space evaluation with fpgas in combinatorial problems: The eternity II case. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'11)*. IEEE, 264–268.

[18] N. Melab, R. Leroy, M. Mezmaz, and D. Tuyttens. 2015. Parallel branch-and-bound using private IVM-based work stealing on xeon phi MIC coprocessor. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS'15)*. 394–399.

[19] Seung-Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical work stealing on manycore clusters. In *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS'11)*.

[20] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. 2014. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *J. Supercomput.* 70, 2 (2014), 756–771.

[21] Oliver Pell and Vitali Averbukh. 2012. Maximum performance computing with dataflow engines. *IEEE Comput. Sci. Eng.* 14 (2012), 98–103.

[22] Christian Plessl and Marco Platzner. 2003. Instance-specific accelerators for minimum covering. *J. Supercomput.* 26, 2 (2003), 109–129.

[23] Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A. Constantinides. 2016. A case for work-stealing on FPGAs with OpenCL atomics. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. ACM, New York, 48–53.

[24] Heinrich Riebler, Tobias Kenter, Christian Plessl, and Christoph Sorge. 2014. Reconstructing AES key schedules from decayed memory with FPGAs. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. IEEE.

[25] Heinrich Riebler, Tobias Kenter, Christoph Sorge, and Christian Plessl. 2013. FPGA-accelerated key search for cold-boot attacks against AES. In *Proceedings of the International Conference on Field Programmable Technology (ICFPT'13)*. IEEE.

[26] Iouliia Skliarova and António B. Ferrari. 2004. A software/reconfigurable hardware SAT solver. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 12, 4 (2004), 408–419.

[27] Alex Tsow. 2009. An improved recovery algorithm for decayed AES key schedule images. In *Proceedings of the International Workshop on Selected Areas in Cryptography (SAC'09)*. 215–230.

[28] Qichao Wang. 2012. *Localization and Extraction of Cryptographic Keys from Memory Images and Data Streams*. Master's thesis. University of Paderborn.

[29] Makoto Yokoo, Takayuki Suyama, and Hiroshi Sawada. 1996. Solving satisfiability problems using field programmable gate arrays: First results. In *Principles and Practice of Constraint Programming—CP96*. Springer, 497–509.

[30] Peixin Zhong, Margaret Martonosi, Sharad Malik, and Pranav Ashar. 1997. Implementing boolean satisfiability in configurable hardware. In *Proceedings of the Logic Synthesis Workshop*. Citeseer.

Received July 2016; revised December 2016; accepted February 2017