Compiler-Assisted Selection of Hardware Acceleration Candidates from Application Source Code

Georgios Zacharopoulos[1], Lorenzo Ferretti[1], Giovanni Ansaloni[1],
Giuseppe Di Guglielmo[2], Luca Carloni[2], Laura Pozzi[1]
[1]Università della Svizzera italiana, Lugano, Switzerland
[2]Columbia University, New York, United States

***Abstract*—Hardware design is a difficult task. Beside ensuring functional correctness of an implementation, hardware developers are confronted with multiple and often conflicting constraints, such as performance and area cost targets, that require lengthy explorations. This issue is compounded when considering the acceleration of complex applications, of which some parts are implemented in software, and others are accelerated in hardware. Hardware/Software partitioning must be settled early in the development cycle, and is far from trivial, since at this stage detailed performance measurements are not available, while wrong choices can lead to vastly sub-optimal solutions or to wasted implementation efforts. To address this challenge, we present a framework to automatically identify, from un-modified software code, software segments that are promising candidates for hardware acceleration, to evaluate their potential speedup and resource requirements, and to select a subset of them under resource constraint. Our strategy is based on Intermediate Representation (IR) analysis passes, which we embed in the LLVM compiler toolchain, and does not require any time-consuming synthesis. We explore its effectiveness on the reference software implementation of a complex application, the H.264 Decoder from University of Illinois, and demonstrate that our methodology selects higher-performance sets of accelerators, when compared to strategies only based on profiling information.**

***Index Terms*—Hardware/Software Co-Design, Application Specific Processors, Accelerators Identification, Software Analysis**

I. Introduction

System-level design is witnessing a revolution. Emerging best practices based on High Level Synthesis (HLS) allow unprecedented productivity levels. HLS, in fact, dramatically shortens development cycles by employing C/C++ descriptions as entry points for the development of both software and hardware, greatly easing the task of migrating functionalities between the two.

However, the design of heterogeneous systems comprising software processors and hardware accelerators is still a complex endeavour, during which key decisions are left solely to manual effort and designer expertise [1] [2]. Furthermore, the long time required for hardware synthesis, coupled with the huge space of alternative implementations exposed by real-world applications, limits in practice the number of accelerator choices that can be considered manually by a designer before hardware/software partitioning is settled.

Addressing these issues, performance estimators have been proposed that, while not providing working hardware implementations, can gauge the characteristics of different accelerator implementation alternatives [3] [4]. Nonetheless, these tools can only evaluate one choice of accelerated function at once. Hence, when using them, the evaluation of each potentially viable hardware/software partitioning alternative requires distinct experimental runs, a time-consuming task for large-sized target applications.

To limit the entailed design effort, it is therefore crucial to identify the set of viable acceleration options quickly, and also early in the design process, before performing later and more detailed estimations. This key step is currently poorly supported by design automation tools. Indeed, state of the art early partitioning strategies are solely based on profiling information [5] [6] which, as was also shown by the authors of [7], may often be misleading.

Against this backdrop, herein we present `AccelSeeker`, a methodology to identify and select the suitable acceleration candidates in an application, from its software source code. `AccelSeeker`, which is implemented within the LLVM [6] compiler infrastructure, first provides a measure of the cost (required resources) and merit (potential speedup) of all candidate accelerators in a single, quick pass, and then selects the set that maximises the estimated speedup, while not exceeding a resource target. The use of `AccelSeeker` can therefore guide IC architects in the early design phases, highlighting which segments of a computing flow should be targeted with high level synthesis, and which parts, instead, are not likely to yield tangible benefits if realised in hardware — either because they present a low computational footprint, or because their characteristics hamper their potential for hardware acceleration, e.g. they require an excessive amount of data transfers while performing limited computations.

The approach of `AccelSeeker` is markedly different from that of performance estimators, as we aim at pinpointing the most promising candidates in a single, high-level exploration,

This work was partially supported by the MagicISEs (grant no. 200021-156397) project funded by the Swiss NSF and by the MyPreHealth (grant no. 16073) project funded by the Hasler Stiftung, and by the National Science Foundation (A#: 1764000).

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00024

reducing the scope of further, and more detailed, estimations. On the other hand, it is also distant from approaches based solely on profiling information, because profilers do not offer a measure of costs and run-times of hardware implementations. They also do not account for invocation overheads – potentially leading to the selection of frequently called, but small, candidates – and for data transfers – hence potentially suggesting candidates requiring an excessive amount of communication. In both cases, poor choices may even result in slower systems with respect to a software-only alternative.

Our contribution is two-fold:

- we introduce `AccelSeeker`, a compiler-based framework able to assist system designers in hardware/software partitioning choices.
- we explore `AccelSeeker` effectiveness on a complex application (H.264 decoding [8]), showcasing that its selection of acceleration candidates performs partitioning choices of higher quality compared to those solely based on profiling.

II. Related Work

High Level Synthesis tools have considerably matured in recent years [9]. Nowadays, available commercial tools (e.g.: Xilinx Vivado HLS [10], Cadence Stratus HLS [11]), as well as academic alternatives (e.g.: Legup [12], Bambu [13]) support the design of very large accelerators from C/C++ code. They reach performance levels comparable to hand-crafted implementations specified in low-level Hardware Description Languages (HDL) such as VHDL or Verilog [8].

Nonetheless, the automated selection of the application parts most amenable to hardware acceleration is still an open research topic. Selection approaches based on synthesis results [14] scale poorly to complex applications, as these are only available late in the design process. Estimation frameworks offer a detailed analysis on the performance and resource requirements of a hardware-accelerated system while avoiding full synthesis runs, either by interfacing software and hardware simulators (e.g., gem5 [15] and Aladdin [16] in [3]), or by adopting a hybrid stance, in which hardware execution times are estimated, while software ones are measured on a target platform (as in Xilinx SdSoC [4]). However, in both cases, estimations are performed *after* the partitioning of hardware and software, which is left to a trial-and-error basis. A methodical solution for partitioning is instead proposed here.

The downside of poor partitioning choices, and consequently the importance of automated tools such `AccelSeeker` that guide the selection of high-quality accelerator sets, is even more prominent when considering the high effort required to optimise the implementation of HLS-defined accelerators. Design optimisation entails the specification of multiple directives to steer designs towards the desired area-performance trade-off. The link between directives and the performance of implementations is not straightforward, hence requiring the evaluation of multiple alternatives to reach the intended results, as exemplified in [17] [18] [19] [20] [21] [22]. It is therefore key to focus

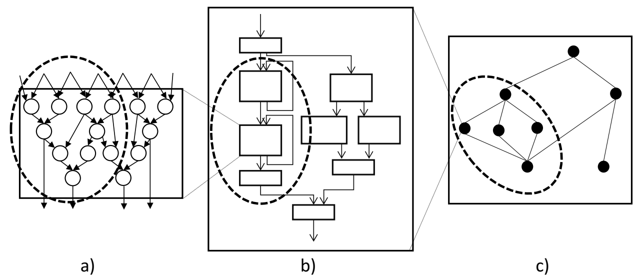


Fig. 1. Evolution of the SoA in automatic selection of custom instructions/accelerators: (a) from data-flow level [23] [24], (b) to control-flow level [25] [26], (c) to function-call graph level (this work).

up-front this optimisation effort only on those candidate accelerators which can lead, from an application perspective, to tangible speedups.

To this end, our approach is inspired by previous works on automatic identification of instruction set extensions. Most techniques in this field target customizable processors augmented with application-specific functional units, within the processor pipelines. Hence, these techniques usually constrain their search to the scope of single basic blocks [23] [24], as depicted in Figure 1a. Recently, the authors of [25] and [26] have instead targeted the identification of larger code segments, including control-flow structures belonging to single functions (depicted in Figure 1b). However, such scope still falls short of the one employed in HLS tools, which are devoted to the implementation of dedicated accelerators interfaced on a system bus [27]. In this setting, the cost of data movement becomes so prominent that even control-flow structures inside functions fail to deliver performance. Suitable accelerator candidates must then encompass entire functions, including in turn all functions called within their call tree. `AccelSeeker` considers this same granularity (Figure 1c), advancing the state of the art in automatic accelerators identification.

III. Methodology

In the following, we detail the methodology embedded in `AccelSeeker`, whose high-level scheme is depicted in Figure 2. First, we define what a candidate for acceleration is, and then we detail how we select, among a pool of such candidates extracted from an application source code, the subset to be implemented in hardware (boxes A and C in the figure). The approach we employed to estimate the candidates performance and resource requirements (box B) is instead detailed in Section IV.

A. Candidate Identification

In order to discover which parts of an application can be most profitably accelerated in hardware, we investigate its function-call graph, i.e., a Directed Acyclic Graph $G(N, E)$, where every node $n \in N$ corresponds to a function and every edge $e = (u, v) \in E$ corresponds to a call from function u to function v. A *root* is a node that reaches all other nodes of the graph, i.e., for every other node $n \in N$, there exists a path from the root to it. The function-call graph G has a root, which represents the top-level function of the application. Figure 3a

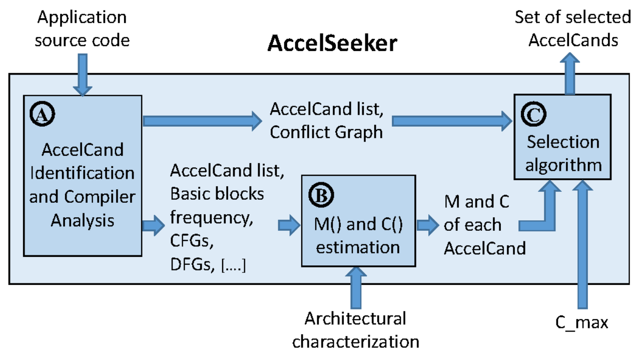


Fig. 2. The phases of the `AccelSeeker` approach. A) Candidates for acceleration are identified, from source code analysis. B) Given an estimation of merit and cost for each candidate, and C) given a maximum available cost, `AccelSeeker` performs selection of a subset of such candidates.

shows an example of a call graph (note that edge directions are not shown here, for picture clarity; they are, however, intended from top to bottom).

We define as candidate accelerator, and call it **AccelCand**, a subgraph $S(N_s, E_s)$ of graph G, exhibiting the following two characteristics: the subgraph has a root; the subgraph has zero outgoing edges. The former means that the subgraph has a node that reaches all other nodes in the subgraph; the latter means that, for every node $n_s \in N_s$, no edge (n_s, m_s) exists in G such that $m_s \notin N_s$.

Figure 3a and 3b show three example subgraphs, labelled A, B, C. While subgraph A is an AccelCand, subgraph B is not, because it does not have zero outgoing edges, and subgraph C also is not an AccelCand, because it does not have a root. The call graph resulting from selection of AccelCand A as accelerator is shown in Figure 3c: the whole subgraph is subsumed to a single (accelerator) call.

Note that we limit our methodology to considering call graphs that are acyclic, and hence we cannot deal with constructs such as recursion. This is in line with the limitations of HLS tools.

B. Problem Statement and Candidate Selection

Given a call graph $G(N, E)$, there exist $|N|$ AccelCands in it; every node of G is, in fact, the root of one and only AccelCand. The problem of *Selection* is that of choosing, among all of the $|N|$ AccelCands, the subset to be realised as accelerators.

We associate to each AccelCand a merit $M()$ – an estimation of the number of cycles saved by it when implemented in HW as opposed to SW – and a cost $C()$ – an estimation of the area needed by it when implemented in HW. Note that the methodology here proposed is agnostic to the way cost and merit are defined. Of course, their definition should correctly reflect the SW and HW architectures that the methodology is targeting, and the details of how functions $M()$ and $C()$ are defined for the experiments in this paper are given in Section IV.

Given a set of AccelCands, defined and identified as described in the previous subsection, and given a cost and merit

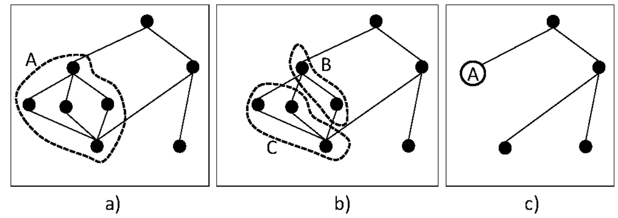


Fig. 3. a) An example call graph. Black nodes are the functions present in a SW application, and edges represent function calls. Subgraph A is an AccelCand. b) Subgraphs B and C are not AccelCands (B has outgoing edges, C has no root). c) Call graph resulting from selection of A as accelerator.

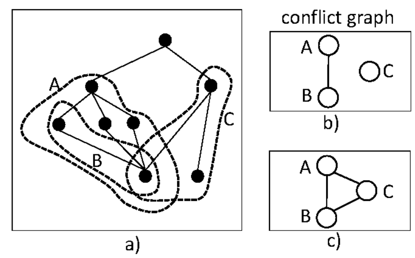


Fig. 4. a) Three overlapping AccelCands: A, B, C. b) Conflict graph considered in our problem formulation: complete overlap is a conflict, while partial overlap is allowed c) Conflict graph adopted in [25] instead, where any kind of overlap is considered a conflict.

associated to each of them, the problem we solve in this paper is now presented.

Problem: ***Accel Selection***

Let $A = \{A_1, A_2, \ldots, A_n\}$ be a set of AccelCands, with associated cost and merit functions C and M. For any subset $S \subseteq A$, we denote by $M(S) = \sum_{i \in S} M(A_i)$ the *sum* of the candidate merits, and we denote by $C(S) = \sum_{i \in S} C(A_i)$ the *sum* of their costs.

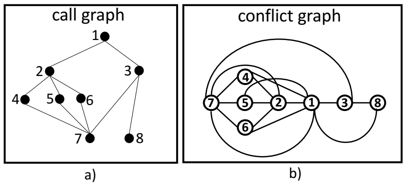
We want to select a subset S of AccelCands such that

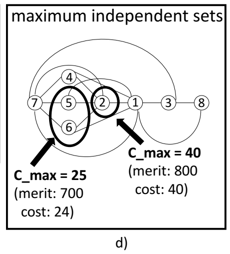
1) The merit $M(S)$ is maximised
2) The cost $C(S)$ is within a user-given cost budget $C_{\max}$
3) No two candidates in set S are in *conflict*

We define the concept of *conflict* in the following way: two candidates A_i and A_j in S are in conflict if and only if $A_i \subset A_j \vee A_j \subset A_i$, i.e., if one completely includes the other.

The concept of conflict that we consider in our problem formulation is exemplified with the help of Figure 4. A call graph is shown, where 3 AccelCands (out of the possible 8, as there are 8 nodes in the call graph) are highlighted, and they are labelled A, B, and C. We also show a *conflict graph*, in Figure 4b, which is an undirected graph where nodes represent AccelCands, and an edge is added between two nodes if the two candidates are in conflict. A and B are in conflict because B is completely contained in A.

The reason behind the concept of conflict is that, in the problem formulation (and in our implementation to solve it) the merit of a set of candidates is defined as *additive*: it is the sum of the individual merits of candidates selected. Since the merit of B is already completely counted within the merit of A, the two candidates should not be both selected (and their merit should not be counted twice). Note that, if the overlap is instead only partial – as is the case for candidates A and C, which in this example share a call to a single function – we





AccelCand	1	2	3	4	5	6	7	8
Merit	900	800	150	50	300	400	100	50
Cost	100	40	60	20	12	12	10	30

c)

Fig. 5. a) An example call graph with eight nodes, and hence eight AccelCands, and b) the corresponding conflict graph. c) Given example merits and cost values associated to the eight AccelCands, d) and given a maximum tolerated cost C_{max}, maximum independent sets that solve problem *Accel Selection* are shown. A maximum independent set maximises merit, while not exceeding the given cost, and not including conflicts.

are able to correctly model the two merits separately, because we can identify how many calls to the 'shared' functions come from within candidate A, and how many come from within candidate B. Hence, partial overlap does not constitute a conflict.

The problem formulation of *Accel Selection* borrows from that found in [25]. It has however an important difference. Since [25] targets Regions within a the control-flow graph of a single functions as candidates for acceleration, no overlaps are allowed within the same selection. Conversely, we consider subgraphs of the function-call graph, hence allowing solutions including partially overlapping AccelCands.

C. Selection Algorithm

Solving the *Accel Selection* problem on the function-call graph of the application corresponds to solving the *independent set problem* on the resulting conflict graph. The conflict graph, in fact, expresses which pairs of AccelCands are in conflict; thus, an *independent set* of the conflict graph satisfies condition 3 of the *Accel Selection* Problem.

We therefore implement an algorithm that recursively explores the independent sets of the conflict graph, similarly to the Bron-Kerbosch algorithm [28], and that returns the set S with the highest merit $M(S)$ (hence satisfying condition 1 of the problem formulation) and whose cost $C(S)$ does not exceed a user-given maximum cost C_{max} (hence satisfying condition 2 of the problem formulation). This returned set represents the optimal solution to the *Accel Selection* Problem.

An algorithm solving an independent set problem is of course one of non-polynomial complexity. Our exact implementation is still able to find the optimal solution for the experiments in this paper in a matter of milliseconds, even for the considerable dimension of the function-call graph of the application considered (a function-call graph of 63 nodes, as detailed in the experiments section).

An example of selection can be seen in Figure 5. First the example call graph is depicted, which has 8 nodes and hence 8 AccelCands, each *rooted* in one of the 8 nodes. The eight corresponding AccelCands are not depicted in this figure, but some of them can be seen in Figure 4a: for example, the AccelCand rooted in node 2 is depicted there and labelled A,

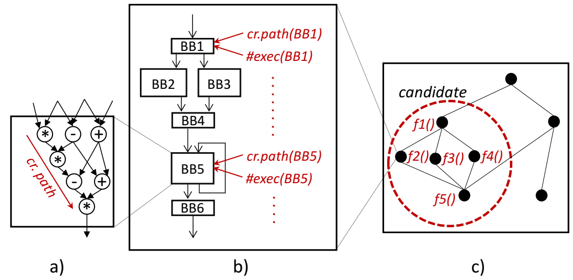


Fig. 6. Estimation of hardware computation times at the basic block (a), function (b) and AccelCand (c) levels.

the AccelCand rooted in node 3 is also depicted in the same Figure and labelled B, etc. Figure 5b depicts the complete conflict graph corresponding to this example. As can be seen, candidate 1 (corresponding to the whole graph) is in conflict with all other candidates, candidates 2 and 3 are not in conflict (they only overlap in function 7) etc. Now, given example values of cost and merit for each candidate (in Figure 4c), the maximum independent set is found in the conflict graph, which maximises the sum of merits of the candidates selected, does not overcome a maximum sum of cost, and does not include conflicts. Two examples (for $C_{max} = 25$ and for $C_{max} = 40$) are shown in Figure 4d.

IV. Cost and Merit estimation

Herein, we detail how the abstract cost and merit employed in the previous section are automatically computed from source code (Figure 2, box B). As the goal of our framework is to select the most performing candidates *in advance of their optimisation*, `AccelSeeker` considers their default implementations, e.g., ones where no function is inlined and no loop is unrolled. High-performance implementations will likely have greater resource requirements, in turn potentially requiring to discard some of the selected AccelCands. Nonetheless, these additional design decisions will be performed within the limited scope of the candidate set retrieved by our tool (as opposed to the whole design) thereby easing the ensuing effort.

A. Architectural characterisation

`AccelSeeker` bases its estimations on few parameters characterising the target platform. Being them only related to the modelled architecture, but independent from the application, the characterisation represents a one-time effort for a given hardware target.

For the experiments in Section VII, this task was performed by employing a series of micro-benchmarks, synthesised on a Zynq Programmable System-on-Chip (PSoC). Our methodology, however, is not limited to this target. On the contrary, it can be adapted to different computing architectures (e.g.: ASIC implementations) by measuring 1) the area and critical path of single operators (adders, multipliers, etc...), 2) the overhead entailed by initiating an acceleration invocation, 3) the time required to transfer inputs and outputs and 4) the resources employed to realise accelerator-memory links (realised by default as `master axi` ports in Zynq systems).

B. Estimating cost

We compute the cost $C()$ of an AccelCand as the sum of its estimated logic and memory real estates. As for logic, we add up the required resources (independently for look-up tables and DSP blocks) of the arithmetic operations present in its top function. If function calls are present, we recursively account for the area of the called functions as well. Furthermore, again mimicking the default implementation of Xilinx PSoC accelerators, we add the cost of the logic required by a `master axi` port for each array present in the AccelCand parameters list.

Then, the memory area is derived from the size of the arrays storing the input/output and intermediate values required by the accelerator. The I/O size is determined by analysing the elements in the parameters list of the candidate top function, while the memory required for intermediate values is derived from the variable declarations in each candidate, ultimately determining the number of required BRAM blocks. In line with the limitation of HLS tools, we do not support dynamic memory allocations.

C. Estimating merit

The merit $M()$ of an AccelCand is expressed in terms of the number of clock cycles that are saved by implementing it as a hardware accelerator instead of executing it in software. In turn, the estimation of hardware run times must account both for computation bounds and host-accelerator communication overheads. The latter are retrieved by considering the number of required memory accesses, scaled by an architecture-specific factor, as discussed in the previous section.

To assess the computation time of candidates in hardware, we instead proceed in a bottom-up fashion, as also exemplified in Figure 6. First, we compute the maximum propagation delay of each of the Basic Blocks (BBs) present in an AccelCand (both in the top function and in its callees) by traversing their DFGs and accounting for the operations delays, thus retrieving the longest input-to-output paths (Figure 6a). Critical paths of BBs are then expressed in clock cycles, dividing the propagation delays with the period of the system clock. By multiplying the critical paths with the number of executions of each BB, we calculate the associated workload. Finally, an estimate of the computation time of an AccelCand is the sum of the workloads of its constituent BBs (Figure 6b-c).

Software run-times are estimated in a similar fashion, but instead of computing critical paths at the BB level, we sum the latency (in clock cycles) of all its constituent operations, modelling that these are processed sequentially in software.

From the gathered data, the merit of an AccelCand i is computed as follows:

$$M(i) = [T_{Sw}(i) - (T_{oh} + max(T_{Hw}^{comp}(i), T_{Hw}^{comm}(i)))] \times n_{exec}$$

where $T_{Sw}(i)$ is the AccelCand run-time in software, T_{oh} is the fixed overhead required to configure and start the hardware acceleration, $T_{Hw}^{comp}(i)$ and $T_{Hw}^{comm}(i)$ are the run-times when i is hardware-accelerated, assuming its performance is either computation or communication bound. Finally, n_{exec} is the number of times the AccelCand is executed in the application.

V. AccelSeeker Compiler Analysis

AccelSeeker is implemented as a compiler pass within the LLVM 3.8 [6] infrastructure. The resulting implementation comprises methods for the identification and analysis of the AccelCands (Figure 2, box A), for the estimation of their merit and cost (Figure 2, box B), and for their selection (Figure 2, box C). In this section we give further details on how the data needed in these phases is retrieved using our LLVM IR-level analysis.

AccelCands identification and analysis. For the generation of the call graph, we annotate every function with caller/callee relationships; we then traverse the call graph to identifies all valid AccelCands as defined in section III-A. At this level we also detect information regarding the overlapping of such AccelCands, needed for the creation of the conflict graph, and for subsequent selection. The control flow graph of each candidate, and the data flow graph of each basic block are extracted so that they can be used as input for the cost and merit calculation, according to the method already detailed in Section IV-B and IV-C

Execution Frequency. The number of invocations of each candidate as well as the execution frequency of each basic block in each candidate is retrieved via LLVM and Clang, using a profiling-via-instrumentation routine, which requires the generation of an instrumented version of the code, and then enables the obtained frequencies to be annotated back to the IR level.

HW Latency Communication Estimation. In order to take into account the memory latency overhead due to data exchange between the implemented HW accelerators and main memory (term $T_{Hw}^{comm}(i)$ in Section IV-C), I/O requirements for each AccelCand are estimated within the LLVM framework by retrieving the parameter list of each candidate and obtaining the data requirements of each candidate type (e.g. size of array of integers, size of a struct etc).

Area of Master AXI ports Estimation. To account for the HW resources required for a Master AXI port, the parameter list of each candidate is analysed. Every array identified accounts for extra logical units (LUTs), contributing to the total area.

Selection. The selection algorithm described in Section III-C is implemented as a C++ program which receives, from the LLVM passes, the list of identified AccelCands, their associated cost and merit, the conflict graph, and the value of the maximum available resources C_{max}. In output, it returns the maximum independent set that solves the *Accel Selection* Problem, i.e., the set of AccelCands to be implemented in HW.

VI. Experimental setup

Validation. We evaluated the outcome of our selection of candidates by implementing the corresponding hardware-accelerated systems on a Xilinx Zynq Ultrascale+ PSoC running the Linux operating systems. The system is clocked at 100MHz, with one of its Cortex A53 processors being

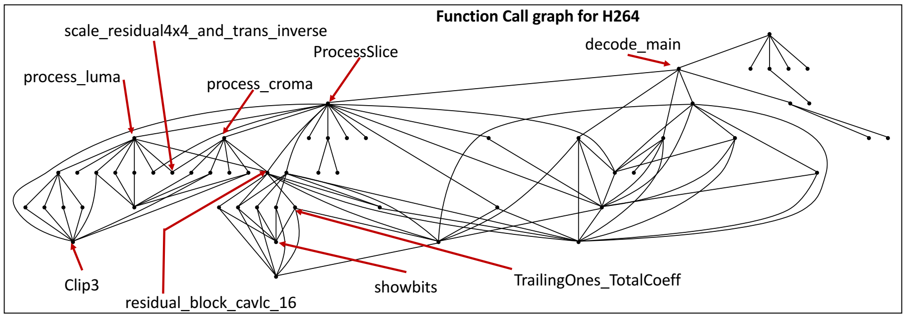


Fig. 7. Call graph of H.264, with some function names highlighted.

dedicated to the execution of the software (non-accelerated) parts of the considered benchmark.

Baselines for comparative evaluation. We compared the quality of the choice of accelerators given by `AccelSeeker` with the ones a designer would obtain when guided solely by a software profiling tool instead. For such baseline solutions, we rely on the *gprof* tool [29]. Gprof retrieves the software execution time of all functions, but provides no support for the estimation of hardware execution times, hardware area, nor I/O and invocation overheads. Mimicking the possible strategies a designer would follow based on profiling data, we considered three possible alternatives:

- In a breadth-first approach (termed *gprof1*), the leaf function with the highest computing time is selected first. Further functions are considered for hardware execution recursively, as the ones a) having the highest computation time in software, and b) that are either leaves in the call graph, or, in case they have callees, those have all been already selected in previous steps. After synthesis, a candidate is implemented in hardware if its inclusion in the accelerator set does not violate the area constraint.
- Conversely, *gprof2* adopts a depth-first stance. It also starts from the most compute-intensive leaf in the application call graph. It then traverses it by iteratively considering the parents of the current candidate, in order of decreasing workloads, selecting the highest-workload one which does not exceed the area budget.
- Finally, *gprof3* selects the most compute-intensive functions (without accounting for their callees) first, regardless of the call graph hierarchy.

Candidate	Validation Ranking	Estimation Ranking (`AccelSeeker`)	Estimation Ranking (gprof)
residual-block-cavlc-16	1	1	4
TrailingOnes-TotalCoeff	2	2	2
inter-prediction-chroma	3	3	5
scale-residual4x4	4	7	6
total-zeros	5	5	9
prediction-Chroma	6	10	13
IntraInfo	7	9	18
run-before	8	4	15
...	...	...	...
showbits	17	17	1
Clip3	18	18	3

TABLE I
RANKING OF ACCELCANDS, BASED ON APPLICATION SPEEDUP WHEN IMPLEMENTED AS ACCELERATORS ON THE ZYNQ PSOC IMPLEMENTATION (COLUMN 2), AS WELL AS ACCORDING TO EARLY ESTIMATION STRATEGIES (ACCELSEEKER, COLUMN 3, AND GPROF, COLUMN 4).

In all cases, these baselines disregard functions that contribute less than 0.5% to the total run-time, as these will not be of interest to a designer. In Section VII, we show that `AccelSeeker` outperforms the strategies above, outlining that the more comprehensive insights it offers are crucial towards pinpointing the AccelCands leading to higher speedups, and in defining higher-performance hardware/software partitionings under resource constraint.

Benchmark application. We perform the experiments on the H.264 video decoding benchmark released by University of Illinois [8], processing three video segments provided by the benchmark authors (in QCIF (176x144), CIF (352x288) and VGA (640x480) formats, respectively). The targeted implementation comprises 63 functions and more than 6000 lines of code. It is derived from the H.264 reference code described in [30], which was adapted to avoid non-synthesisable constructs.Its call graph is presented in Figure 7, along with the names of some of the functions.

VII. EXPERIMENTAL RESULTS

A. Ranking of acceleration candidates

Herein, we showcase the effectiveness of `AccelSeeker` in identifying the AccelCands most amenable to hardware

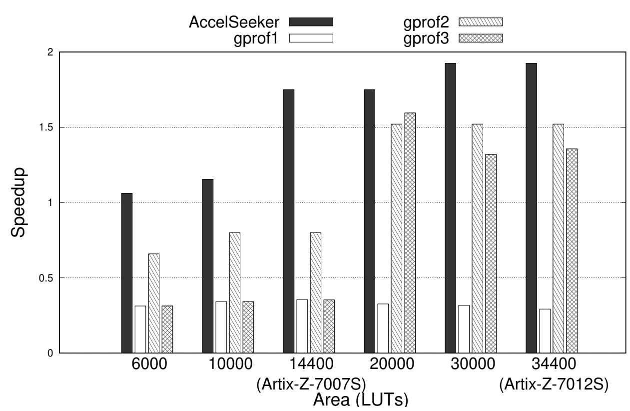


Fig. 8. Speedup obtained over the whole runtime of H.264 decoder by implementing, as hardware accelerators, the candidate sets obtained with `AccelSeeker` and the ones retrieved by gprof1, gprof2 and gprof3 profiling strategies (as detailed in VII-B), varying the area constraint.

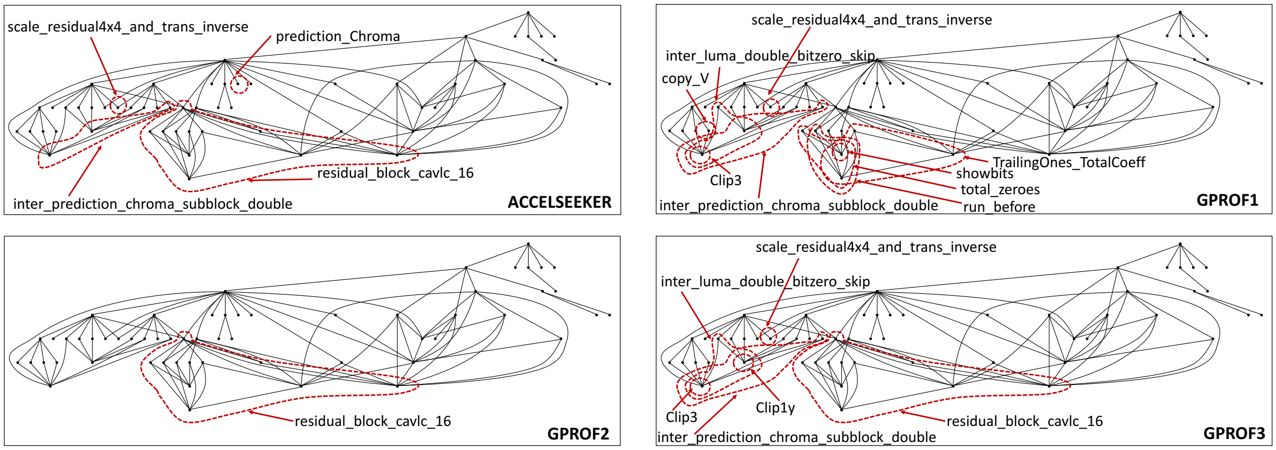


Fig. 9. H.264 call-graphs highlighting the acceleration candidates selected by by `AccelSeeker`, and by the three gprof strategies, for a 30k LUTs area budget.

Max LUTs	**`AccelSeeker`**	**gprof1**	**gprof2**	**gprof3**
6 000	TrailingOnes_TotalCoeff	showbits TrailingOnes_TotalCoeff Clip3 write_luma Clip1y	showbits TrailingOnes_TotalCoeff	showbits TrailingOnes_TotalCoeff Clip3 write_luma Clip1y
10 000	inter_prediction_chroma_subblock_double scale_residual4x4_and_trans_inverse	showbits TrailingOnes_TotalCoeff Clip3 scale_residual4x4_and_trans_inverse Clip1y total_zeros	showbits TrailingOnes_TotalCoeff Clip3	showbits TrailingOnes_TotalCoeff scale_residual4x4_and_trans_inverse Clip1y total_zeros
20 000	TrailingOnes_TotalCoeff inter_prediction_chroma_subblock_double scale_residual4x4_and_trans_inverse	showbits TrailingOnes_TotalCoeff Clip3 scale_residual4x4_and_trans_inverse inter_prediction_chroma_subblock_double inter_luma_double_bizero_skip total_zeros	residual_block_cavlc_16	Clip3 residual_block_cavlc_16 scale_residual4x4_and_trans_inverse Clip1y
30 000	residual_block_cavlc_16 inter_prediction_chroma_subblock_double scale_residual4x4_and_trans_inverse prediction_Chroma	showbits TrailingOnes_TotalCoeff Clip3 scale_residual4x4_and_trans_inverse inter_prediction_chroma_subblock_double inter_luma_double_bizero_skip total_zeros copy_V run_before	residual_block_cavlc_16	Clip3 residual_block_cavlc_16 scale_residual4x4_and_trans_inverse Clip1y inter_prediction_chroma_subblock_double inter_luma_double_bizero_skip

TABLE II
ROOT FUNCTION OF THE SELECTED H.264 CANDIDATES, FROM THE REFERENCE CODE IN [8], FOR DIFFERENT METHODS AND RESOURCE BUDGETS.

acceleration. For this round of experiments, we implemented the best suggested candidates either by gprof or by `AccelSeeker`, disregarding those which are too large to be mapped in the programmable logic of the employed test system (Xilinx Zynq XCZU9EG). In Table I, AccelCands are ordered by the speedup they provide on the Zynq PSoC when implemented as accelerators (column 2), compared to a fully software execution. `AccelSeeker` estimates a very similar ranking (reported in column 3), with only minor differences. Instead, a ranking based on profiling-only information such as gprof (column 4) badly correlates with actual achievable speedups. Indeed, some candidates suggested (e.g.: *Clip3()* and *showbits()*) actually present a larger run-time in hardware than in software, and are ranked poorly both by `AccelSeeker` and by validation. Results refer to the QCIF test video. Very similar outcomes were retrieved using the CIF and VGA inputs: 9 out of 10 of the highest-merit candidates are the same, with almost identical ranking.

B. Performance of resource constrained accelerator selections

To measure the performance of the proposed method, we compare the application speedups of the hardware-accelerated systems selected by `AccelSeeker`, under different C_{max} constraints, to those selected by the baseline methods. We express such constraint as a maximum number of LUTs dedicated to the accelerators implementation (including that of two real-world PSoCs, namely Xilinx Artix Z-7007S and Z-7012S [31]); similar considerations could be derived by instead limiting BRAMs or DSPs, or combinations of the three.

Figure 8 shows these results, again for the QCIF test input. The speedups are obtained by comparing the run-time of the benchmark application on accelerated systems (where selected AccelCands are executed in hardware) with the non-accelerated one (where all parts are run on the PSoC processor). The figure comparatively reports also the speedups obtained when using the three profiling-based strategies outlined in Section VI. These results show that our approach returns a performance increase even for very low area constraints, and in a 1.9X speedup for an area budget of 34 400 look-up tables (the amount available on the mid-range Artix Z-7012S).

On the other hand, the candidates identified by all profiling strategies fail to save any run-time (leading instead to slowdowns) for tight areas, because the advantages of hardware acceleration are dwarfed by invocation and data transfer overheads, which are not estimated by tools based only on profiling data. Even when some performance enhancement is achieved, as is the case for *gprof2* and *gprof3* for more lenient constraints, the retrieved selections are of inferior quality with respect to the `AccelSeeker` ones. Moreover, in gprof strategies an increase in the resources dedicated to hardware acceleration may even worsen the actual performance of the system, since more and more ill-performing candidates are earmarked for hardware execution. Conversely, the sets of AccelCands selected by `AccelSeeker` monotonically increase in performance as the C_{max} constraint is relaxed.

Further detailing the outcomes of our methodology and the considered profiling-based baselines, Table II reports the root function of the AccelCands selected for hardware acceleration under different area constraints, while Figure 9 depicts them on the H.264 call graph for a budget of 30K LUTs. This experimental evidence highlights that the breadth-first *gprof1* approach tends to select a large number of small, leaf functions which, due to the high implied overheads, fail to achieve high performance. A depth-first stance (embodied in *gprof2*) may instead select too few candidates, as it is restricted to focus only on a single branch of the function-call graph. Speedup opportunities are also missed by disregarding the call graph hierarchy entirely, as done in *gprof3*. Ultimately, higher performance can be obtained through the non-obvious selection of accelerator sets identified by `AccelSeeker`.

The reasons behind this superiority are twofold. Firstly, `AccelSeeker` is not only guided by execution frequency, as profiling is: it can instead account for the potential speedup that can be harnessed via hardware execution, and for the traded-off *overhead* due to transferring data between processors and accelerators (see Section IV-C). It can then evaluate this in the light of the resource *cost* that a dedicated hardware unit entails (see Section IV-B). Secondly, `AccelSeeker` is empowered by the selection algorithm described in Section III-C, which solves the *Accel Selection* Problem, maximising merit under cost constraint. Given an instance such as H264, with a call-graph of 63 functions, and resulting in a conflict graph of 63 nodes and 361 edges, it is evident that the problem should not be left to be solved manually by designers. As opposed to an approach based on profiling only, then, our compiler-based strategy is well-suited to guide this complex challenge.

C. Design effort analysis

A single invocation of `AccelSeeker` retrieves an entire set of acceleration candidates, focusing on those that can best leverage hardware acceleration. Conversely, all profiling-based

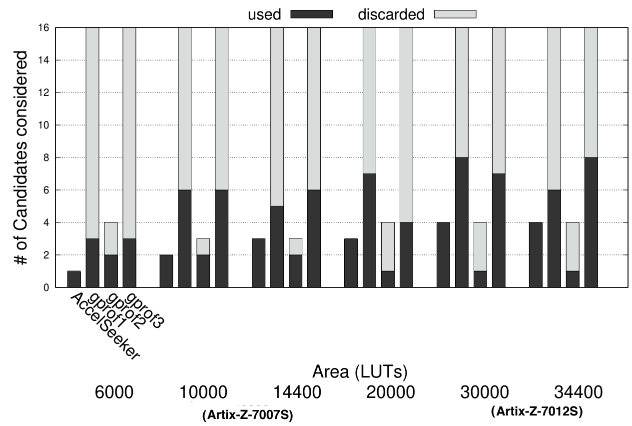


Fig. 10. Number of candidates selected by `AccelSeeker` and, for comparison, by the gprof-based strategies, while varying the area constraints. Candidate accelerators selected by gprof exceeding resource constraints can only be discarded after their implementation.

baselines necessitate a trial-and-error stance, because resource estimations are not available and cannot be relied upon to discard up-front AccelCands that exceed available budgets. Therefore, these strategies either mandate a large number of synthesis runs for many possible choices (*gprof1*, *gprof3*) or overly restrict the set of possible acceleration candidates, thereby hampering the resulting speedups (as is the case of *gprof2*). Indeed, this effort is reported in Figure 10: the majority of the candidates identified by profiling ultimately violate the resource constraints, across different strategies and amounts of available resources. The synthesis of such candidates is avoided by instead employing `AccelSeeker`, hence greatly reducing the design effort towards the selection of highly effective hardware/software partitionings. Indeed, the collection of all `AccelSeeker` phases took a time in the order of milliseconds for the experiments in Figures 8 and 10.

VIII. Conclusions

We presented `AccelSeeker`, a framework for assisting system architects during the early design phase of hardware-accelerated systems. By automatically assessing the potential speedup of different hardware acceleration choices, in their default implementation , as well as the hardware resources they demand, `AccelSeeker` allows architects to pinpoint the code sections that are worthy targets for further, more detailed analysis and optimisation.

`AccelSeeker` performs the identification of candidate accelerators, as well as their area and speedup estimations, through compiler analysis passes implemented within the LLVM compiler, without requiring lengthy and detailed evaluations of each acceleration candidate individually. It then automatically selects the set of candidates that maximise estimated speedup under a given resource constraint. Experimental evidence highlights that the hardware/software partitionings selected by `AccelSeeker` vastly outperform choices that are solely based on profiling information.

REFERENCES

[1] M. Cacciotti, V. Camus, J. Schlachter, A. Pezzotta, and C. Enz, "Hardware acceleration of HDR-image tone mapping on an FPGA-CPU platform through high-level synthesis," in *International System-on-Chip Conference*. IEEE, Sep. 2018, pp. 158–162.

[2] S. Nouri, J. Rettkowski, D. Göhringer, and J. Nurmi, "HW/SW co-design of an IEEE 802.11 a/g receiver on Xilinx Zynq SoC using high-level synthesis," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. ACM, Jun. 2017, pp. 1–6.

[3] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-aladdin," in *MICRO 49: Proceedings of the 46st Annual International Symposium on Microarchitecture*, Oct. 2016, pp. 1–12.

[4] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "SDSoC: A higher-level programming environment for Zynq SoC and Ultrascale+ MPSoC," in *Proceedings of the 2016 ACM/SIGDA 24th International Symposium on Field Programmable Gate Arrays*, Feb. 2016, pp. 4–4.

[5] "Xilinx embedded system tools reference manual," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1043-embedded-system-tools.pdf

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, Mar. 2004, pp. 75–88.

[7] B. A. Syrowik, B. Fort, and S. D. Brown, "Use of CPU performance counters for accelerator selection in HLS-generated CPU-accelerator systems," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, Jun. 2018, pp. 1–6.

[8] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "High level synthesis of complex applications: An h. 264 video decoder," in *Proceedings of the 2016 ACM/SIGDA 24th International Symposium on Field Programmable Gate Arrays*, Feb. 2016, pp. 224–233.

[9] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, Sep. 2012.

[10] Xilinx, "Vivado high-level synthesis," www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, Mar. 2017.

[11] Cadence, "Stratus high-level synthesis," www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html, Apr. 2016.

[12] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[13] C. Pilato and F. Ferrandi, "Bambu: A free framework for the high level synthesis of complex applications," Mar. 2012.

[14] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski *et al.*, "From software to accelerators with LegUp high-level synthesis," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. IEEE, Sep. 2013, p. 18.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Feb. 2011.

[16] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceedings of the 41st Annual International Symposium on Computer Architecture*. IEEE, Jul. 2014, pp. 97–108.

[17] B. C. Schafer and K. Wakabayashi, "Divide and conquer high-level synthesis design space exploration," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 3, pp. 29:1–29:19, Jun. 2012.

[18] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-traversing design space exploration for high level synthesis," in *Proceedings of the International Conference on Computer Design*. IEEE, Oct. 2018, pp. 210–217.

[19] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th Design Automation Conference*. IEEE, Jun. 2013, pp. 1–6.

[20] L. Ferretti, G. Ansaloni, and L. Pozzi, "Cluster-based heuristic for high level synthesis design space exploration," *IEEE Transactions on Emerging Topics in Computing*, no. 99, pp. 1–9, Jan 2018.

[21] L. Piccolboni, P. Mantovani, G. D. Guglielmo, and L. Carloni, "COSMOS: coordination of high-level synthesis and memory optimization for hardware accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 150:1–150:22, Sep. 2017.

[22] G. Zacharopoulos, A. Barbon, G. Ansaloni, and L. Pozzi, "Machine learning approach for loop unrolling factor prediction in high level synthesis," *2018 IEEE International Conference on High Performance Computing & Simulation (HPCS)*, pp. 91–97, 2018.

[23] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–29, Jul. 2006.

[24] E. Giaquinta, A. Mishra, and L. Pozzi, "Maximum convex subgraphs under I/O constraint for automatic identification of custom instructions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 3, pp. 483–494, 2015.

[25] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, and L. Pozzi, "RegionSeeker: Automatically identifying and selecting accelerators from application source code," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 741–754, Apr. 2019.

[26] J. Oppermann and A. Koch, "Detecting kernels suitable for C-based high-level hardware synthesis," in *Smart World Congress*. IEEE, Jul. 2016, pp. 1157–1164.

[27] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proceedings of the 52nd Design Automation Conference*. ACM, Jun. 2015, pp. 1–6.

[28] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," in *Communications ACM*, vol. 9, 1973, pp. 575–577.

[29] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: a call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, Jun. 1982, pp. 120–126.

[30] K. Suehring and al., "H.264/AVC reference software," http://iphome.hhi.de/suehring/tml/, May 2015.

[31] Xilinx, "Xilinx all programmable SoC portfolio," www.xilinx.com/products/silicon-devices/soc.html, Mar. 2017.