# Rapid Memory-Aware Selection of Hardware Accelerators in Programmable SoC Design

Alok Prakash, *Member, IEEE,* Christopher T Clarke, *Member, IEEE,* Siew-Kei Lam, *Member, IEEE,* and Thambipillai Srikanthan, *Senior Member, IEEE*

***Abstract*—Programmable Systems-on-Chips (SoCs) are expected to incorporate a larger number of application-specific hardware accelerators with tightly integrated memories in order to meet stringent performance-power requirements of embedded systems. As data sharing between the accelerator memories and the processor is inevitable, it is of paramount importance that the selection of application segments for hardware acceleration must be undertaken such that the communication overhead of data transfers do not impede the advantages of the accelerators. In this paper, we propose a novel memory-aware selection algorithm that is based on an iterative approach to rapidly recommend a set of hardware accelerators that will provide high performance gain under varying area constraint. In order to significantly reduce the algorithm runtime while still guaranteeing near-optimal solutions, we propose a heuristic to estimate the penalties incurred when the processor accesses the accelerator memories. In each iteration of the proposed algorithm, a two-pass method is employed where a set of good hardware accelerator candidates is selected using a greedy approach in the first pass, and a 'sliding window' approach is used in the second pass to refine the solution. The two-pass method is iteratively performed on a bounded set of candidate hardware accelerators to limit the search space and to avoid local maxima. In order to validate the benefits of the proposed selection algorithm, an exhaustive search algorithm is also developed. Experimental results using the popular CHStone benchmark suite show that the performance achieved by the accelerators recommended by the proposed algorithm closely matches the performance of the exhaustive algorithm, with close to 99% accuracy, while being orders of magnitude faster.**

## I. Introduction

Embedded systems have undergone significant changes in recent years. In last five years alone, the performance demand from such systems has risen exponentially. At the same time, the area and power constraints have become more rigorous than ever. Such stringent and conflicting requirements have led to the adoption of application specific customization by adding suitable hardware accelerators to realize high performance systems with strict power and area budgets. For example the last three generations of Apple™ system-on-chip (SoC) dedicated more than half of its die area to application-specific hardware accelerators [1]. However, these SoCs typically face prohibitively high non-recurring engineering (NRE) and time-to market (TTM) pressures which make them unsuitable for low-to-mid volume products. Programmable SoCs, such as Xilinx Zynq platform [2] and Altera SoC field programmable gata array (FPGA) [3] address the NRE and TTM challenges while also providing the flexibility for future upgrades through the use of reconfigurable logic. However, they suffer from lower power-performance efficiency as opposed to ASIC designs and therefore require extensive customization to meet the design constraints. The complex design processes required for these SoCs and the lack of tools, however, have proven to be a significant hindrance in their wider adoption. This challenge is further exacerbated by the data sharing between the processor and the application-specific accelerators, which if ignored, can significantly impede the advantage of the accelerators. Hence, rapid tools and design methodologies are essential to generate application-specific custom designs based on such Programmable SoCs in order to meet the various design constraints.

Hameed et al. [4] highlighted the importance of incorporating custom storage elements that are tightly fused with the hardware accelerators in order to offset the performance overheads incurred by data fetches. As such, state-of-the-art accelerators often include tightly coupled on-chip memories with dedicated address space e.g. scratchpad memory to store the accelerator's local data. Unlike caches, the use of dedicated memories like scratchpads leads to benefits such as lower performance overheads (as it does not require tag comparison and address translation) and deterministic memory access latencies. While these accelerator memories should primarily serve specific hardware accelerators, data sharing between the accelerator memories and processor is inevitable. Existing approaches for data sharing between processor and accelerators requires cumbersome software invocations of direct memory access (DMA) engines, which imposes additional burdens on programmers [1].

In our earlier work [5] [6], we demonstrated a technique on Altera FPGA platform to addresses the challenge of using hardware accelerators that do not require DMA invocations or cache-coherence systems for data sharing between the accelerator memories and the processor. In order to achieve maximal performance gains in this architecture model, there is a need for careful analysis of the application in order to identify the best segments of application code for hardware acceleration. Specifically, the implications of data sharing between the accelerator memories and processor must be accounted for during hardware-software partitioning process. The significance of a memory-aware approach for selecting hardware accelerators is shown in Table I, which reports the normalized execution time of the SHA application from MiBench [7] benchmark suite with different combinations of basic blocks (BBs) implemented as hardware accelerators. A basic block is defined as a straight-line of application code segment with no branches in except to enter and no branches out except to exit the code segment [8]. The reason behind accelerating at the basic block level is discussed in Section II.

A. Prakash, S.K. Lam and T. Srikanthan are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore e-mail: (alok, assklam, astsrikan@ntu.edu.sg).

C.T. Clarke is with the Department of Electronic and Electrical Engineering, University of Bath, United Kingdom. e-mail: (eesctc@bath.ac.uk).

TABLE I: Normalized execution time of the SHA application from MiBench [7] with various combinations of basic blocks implemented as hardware accelerators

| Basic Blocks Implemented as Hardware Accelerators | Normalized Runtime (against software only implementation) |
|---|---|
| BB3 | 0.84 |
| BB3+BB6 | 0.95 |
| BB3+BB6+BB10 | 0.81 |
| BB3+BB6+BB10+BB13 | 0.67 |
| BB3+BB6+BB10+BB13+BB16 | 0.53 |

These basic block level hardware accelerators are implemented on the Altera FPGA fabric while the remaining application was executed on the soft-core NIOS II processor. It can be observed that the normalized runtime increases from the case of BB3 to BB3+BB6 due to the memory access penalties (caused by data dependencies between BB6 and BB10) that are incurred when BB6 is introduced as a hardware accelerator. The penalties are eliminated when BB10 is also included as hardware accelerator. Therefore, given a tight area constraint for hardware accelerators, a better solution can be achieved by porting only BB3 to hardware instead of BB3+BB6.

Hence, a more complex selection criteria must be employed in order to consider the complex data dependencies between the various hardware and software sections of the application code. The problem of memory aware selection of hardware accelerators is non-trivial especially with the growing complexity of embedded applications, which leads to a large design space ($2^N$ different hardware/software partitions with $N$ non-overlapping and unique candidate blocks) [9]. This prohibits the use of exhaustive methods for selecting hardware accelerators with high performance gains.

In this paper, an automated selection algorithm is presented that uses a mathematical model to account for the memory dependencies of different hardware accelerators. Combined with the traditional selection metrics such as performance gain, required hardware area etc., the proposed algorithm rapidly recommends the most profitable basic blocks for acceleration with varying area constraint. While the runtime and complexity of an exhaustive search increases exponentially, our method scales almost linearly with increasing number of hardware accelerator candidates. Moreover, the performance achieved by the candidates recommended by the proposed framework is shown to be close to 99% of the exhaustive search results.

The rest of the paper is organized as follows. In the next section, we review the related work in this area. This is followed by a discussion on the selection of hardware accelerator candidates in Section III. Section IV describes in detail the mathematical model to estimate the overhead of accessing local memory blocks in hardware accelerators. The proposed greedy selection algorithm that relies on a selection heuristic is discussed in the next section. We then extend the proposed algorithm in Sections VI and VII to improve the accuracy of the selection results. Section IX concludes this paper with discussions on future work.

## II. Related Work

Hardware accelerators at various levels of granularity are widely used to improve application performance and power consumption. Sun et al. in [10] showed the benefits of using

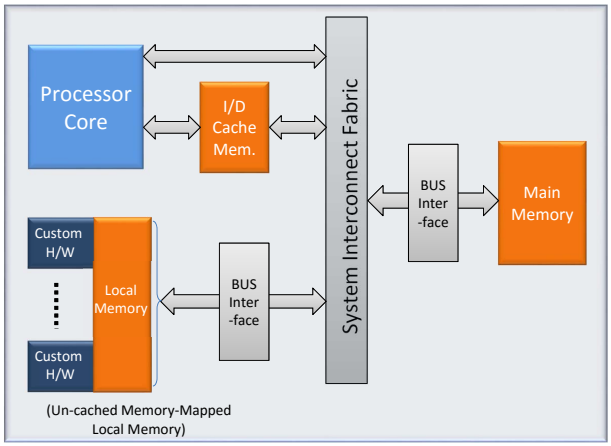



Fig. 1: Architecture model[5]

hardware accelerators of varying granularity including "traditional custom instructions" on Xtensa processors [11], a commercial configurable processor from Tensilica.

### A. Custom Instructions with Memory Operations

The inclusion of memory operations into hardware accelerators necessitates a proper channel for these hardware accelerators to access the memory. The authors in [12], [13] proposed transferring data from the main memory to the local memory of the FPGA by a DMA transfer in a basic block before the hardware accelerator executes. After the hardware accelerator executes on the data in its local memory, the data is transferred back to main memory using another DMA transfer cycle. A significant portion of time savings achieved by executing the code segment in hardware, is eventually lost due to the DMA transfers. Another approach taken to facilitate data sharing between accelerators and the processor is to use caches for accelerator memories. Additional protocols are required to maintain cache coherence. Kluter et al. in [14] and [13] suggested modification to the cache controller to avoid using a rather costly hardware coherence protocol for solving the cache incoherence problem. In [15] the authors allow the accelerators to have direct access to the level one data-cache in the system and therefore able to initiate a memory transfer from the entire processor address space. All the write processes back to the memory only happens after the execution of the accelerator in order to ensure the validity of the data. In case of a Translation Lookaside Buffer (TLB) fault, the compiler introduces additional instructions to ensure the validity of the data. The authors have provided limited information of their work for proper evaluation.

Ensuring cache-coherence is becoming increasingly important, especially due to the growing popularity of multi-core systems. Therefore, companies such as ARM [16] and Synopsys [17] provide hardware IPs [18] [19] for easy integration while providing better performance and power efficiency. While opportunities exist to exploit the benefits of hardware-managed caches in accelerator designs, designing cache interfaces for accelerators is a very challenging problem [1].

In order to avoid DMA invocations or cache-coherence systems for data sharing between the accelerator memories and the processor, in our earlier work [5] we proposed an architecture model, shown in Figure 1 that uses hardware

accelerators with tightly coupled local memory blocks. Fine-grained hardware accelerators at the basic block level were used as custom instructions to show the efficiency of this architecture model. Hardware accelerators based on resource intensive functions in an application invariably waste precious hardware area, since they also accelerate code segments inside the function that do not execute as frequently as other segments and hence are not profitable. On the other hand, by definition, every instruction inside a basic block is executed with the same frequency. Hence, accelerators based on basic blocks, as presented in our earlier work [5] [6] allow designers to make the most efficient usage of the available hardware area by providing much finer control on the accelerated code segments. However, the usage of such fine grained accelerators results in a large design space for selecting the appropriate accelerators.

In [6], we used the architecture model from [5] to derive an equation to estimate the penalty of accessing data by the processor from the FPGA local memory blocks used to store data for hardware accelerators. It should also be noted that as presented in our earlier work, we propose to use hardware accelerators as custom instructions to the processor. This, unlike a co-processor based accelerator, ensures that the data is always synchronized between the local memory blocks associated with the hardware accelerators and the processor. In addition, the local memory blocks were initialized in the un-cached memory address space to the obviate the need for expensive cache-coherence protocols. However, a rapid technique for automatic memory-aware selection of these fine-grained accelerators for hardware-software (HW-SW) partitioning was not presented in [5] or [6].

### B. Selection of Hardware Accelerators

The authors in [9], [20] and [21] have proposed automatic partitioning strategies using hardware accelerators at varying levels of granularity. They used multi-grained accelerators, where the granularity of the accelerators ranged from basic blocks to a function or process level in their design, and the data communication between the hardware and software sections was done through the usage of shared memory space. In [9], they proposed the copying of relevant data in the shared memory space each time, prior to executing the software or hardware components of the application. In [20] the authors proposed an architecture, termed as KAHRISMA, where hardware accelerators of varying granularity could be added to the processor at runtime in order to leverage dynamic reconfiguration of modern high-end FPGAs and realize different processors. They proposed the addition of accelerators in the base processor ISA at runtime using dynamic reconfiguration to suit different applications. Similarly in [21], the authors proposed a run time system to incorporate different fine or coarse grained accelerators into the extensible processor according to application requirements. Based on their works discussed in [9], the data communication between the hardware and software components of the application requires data movement during application execution which incurs additional cost in terms of performance and power. Also, the selection of hardware accelerators was mainly based upon software and hardware performance of the code sections.

LegUp is a high level synthesis tool [22] that compiles a "C" program onto a hybrid architecture consisting of a MIPS based soft processor and custom hardware accelerators. The code sections implemented in hardware are selected manually and fed into the tool as one of the inputs for design space exploration. While the tool itself is extremely useful because of its end-to-end design flow, memory implications have not been considered during the hardware selection.

Yousuf et al. presented the design automation for partial reconfiguration (DAPR) design flow, for hardware-software co-designed systems. DAPRs design flow isolated low-level PR design complexities involved in analyzing PR designs with different performance parameters to make PR more amenable to designers [23]. Zhang et al. proposed a new MILP (Mixed Integer Linear Programming) formulation for hardware-software partitioning targeting MPSoC (Multi-core Processor System on Chip) with a DPR fabric [24].

Tang et al. formulated the optimization of HW-SW partitioning aimed at maximizing streaming throughput with predefined area constraint, targeted for multi-processor system with hardware accelerator sharing capability [25]. Greedy heuristic techniques were proposed to rapidly select appropriate hardware accelerators while achieving close to 94% of the performance obtained by accelerators recommended using an exact brute force approach. They also claim that the communication latencies between hardware and software is insignificant when compared to the computation time and hence can be safely ignored. As shown in our earlier works [5] [6] and also in this paper, ignoring the communication between the hardware and software components can even lead to worse performance as compared to the software only design. However the approaches in [23][24][25] fail to take this overhead into consideration.

In this paper, we build upon our previous work in [5] & [6] and propose a novel memory-aware HW-SW partitioning algorithm to rapidly recommend a set of hardware accelerators that provides high performance gain under varying area constraint. The proposed heuristic-based techniques mitigate the high design space complexity in the hardware software partitioning step and achieve rapid design space exploration and identification of profitable set of basic blocks for hardware acceleration. It is noteworthy that the algorithms proposed in this paper can be integrated into existing high level synthesis tools e.g. LegUp to provide for automated hardware-software co-design and design space exploration.

## III. Selecting Hardware Accelerator Candidates

In order to facilitate the selection of suitable basic blocks for hardware acceleration, we use the LLVM compiler infrastructure [26] for front end compilation of the application code to obtain a target-independent Intermediate Representation (IR). The IR is profiled to obtain the execution frequency of all the basic blocks in the application. The most frequently executed code sections or the "hot blocks" are usually the most preferred candidates for hardware acceleration. The basic blocks are arranged in decreasing order of execution frequency and the top most frequently executed basic blocks are chosen for further analysis. The number of blocks selected at this stage can be varied according to the user input. However, typically

only the top 20 - 30 basic blocks in an application have significant impact on the performance. This observation is in line with the Pareto principle or 90/10 rule [27], which states that 90% of the execution time is spent in 10% of the code.

Once the candidate basic blocks are shortlisted, we perform memory analysis to determine which of these basic blocks can be potentially executed in hardware. In our architecture model, we do not consider basic blocks with dynamically allocated memory for hardware acceleration. The shortlisted candidate basic blocks are therefore divided into two categories: *Implementable* and *Un-implementable* basic blocks.

Apart from their execution frequency, the Implementable blocks are also annotated with other metrics such as their estimated execution time in software and hardware as well as the estimated hardware area. These metrics are necessary for selecting the most profitable blocks given an area constraint. In this work, we estimate the hardware area and execution time using the concepts proposed in [28] and the popular LegUp toolchain [22]. The software time can be estimated as the number of primitive operations in the basic block. Each of the primitive operation such as arithmetic, logical etc. typically take a single clock cycle for execution whereas memory operations generally need longer time to execute if the relevant data is not found in the cache.

We also develop a tool to extract the names, type and size of each variable manipulated in the candidate basic blocks. Since the total hardware area to accelerate a code segment in hardware also includes the memory space required to store the relevant variables, the memory size must also be estimated. We parse the LLVM IR to estimate the size of these memory blocks. Finally, the accesses frequency of all arrays and variables used in the candidate blocks is also identified.

## IV. Estimating Penalty For Using Accelerator Memory

Figure 1 shows the architecture model proposed in our earlier publication [5] that utilizes hardware accelerators with tightly coupled local memories that do not require expensive DMA invocations or cache coherent systems to enable data sharing with the processor.

Consider a block of code `Y` being executed in the processor that needs to process an array `A`. In a traditional memory hierarchy, `A` is stored in the main memory and will be brought into the processor's registers through the cache hierarchy. In an ideal situation, `A` is in the cache and therefore the access latency of this array is minimal. In the proposed model, `A` needs to be processed by the hardware accelerator `X`, and hence `A` is stored in the local memories that are tightly coupled to `X`. Now, if code block `Y` also needs to access `A`, the processor must access it from the local memory via the system bus instead of its data cache. This inevitably requires a longer access time for the processor compared to accessing from its cache. The difference in latency required by the processor to access the local memory instead of the data cache is referred to as the "access penalty". In order to quantify this penalty, the time taken for the processor to access the data from both the local memory and the data cache is modeled. The following terms are defined prior to formalizing this penalty model.

1) **Access Time from Processor to Local Memory ($T_{PM}$):** In the proposed model, the processor accesses the local memories that are tightly coupled with the accelerators through the system bus. The time taken for the processor to access these accelerator memories via the system bus is called $T_{PM}$.
2) **Access Time from Processor to Cache ($T_{PC}$):** The time taken for the processor to access the data from the cache memory is termed as $T_{PC}$. $T_{PC}$ can vary from the best case scenario of 1 clock cycle to multiple clock cycles in the worst case (due to the need of cache flushing etc.). In our experiments, we assume the best case scenario of 1 clock cycle for the software implementation.
3) **Access time from Hardware accelerator to the Local Memory ($T_{HM}$):** The time taken by the accelerator to access a data stored in the local memories is deterministic. One can safely assume that the data can be accessed in a clock cycle and this time is denoted by $T_{HM}$. In addition, multi-ported local memories can be used to minimize the overall access latencies by hardware accelerators through parallel data access.

### A. Penalty Model

Equation 1 models the number of clock cycles saved through implementing a code section `X` in hardware. The first term (*block advantage* of `X`) denotes the number of cycles saved by accelerating a code segment `X` in hardware. The second term is the latency difference between accessing a variable from the cache in case of software implementation, and accessing the same variable in the local memory by the accelerator during hardware implementation. In a typical case, when the processor accesses the data directly from cache, $T_{pc}$ is 1 clock cycle. Similarly, time required for the hardware accelerator to access the data stored in the block RAMs based local memory, is also typically 1 cycle. Hence, in this paper we ignore the second term during the penalty evaluation. In future, we will explore cache modeling techniques to estimate this term more accurately in order to further refine our selection algorithm.

$$\textit{Cycles saved} = \underbrace{(SW_{Time} - HW_{Time}) * Freq_X}_{\text{1st term}} + \underbrace{(T_{PC} - T_{HM}) * Freq_X}_{\text{2nd term}} - \underbrace{\sum_{(i=1)}^{(i=|U|)} (\alpha * m_i * Freq_i)}_{\text{3rd term}} \quad (1)$$

The third term is the maximum access penalty ($P_{max}$) associated with the processor accessing an accelerator memory. This penalty is analogous to the situation where the processor needs to access the data from the main memory instead of its cache hierarchy. Let us denote `U` as the set of basic blocks in software that need to access the local memories in the hardware implementation of `X`, and `X`$\notin$`U`. $m_i$ denotes the number of memory operations in basic block `i`$\in$`U` that is used to access the local memories in the hardware implementation of `X`, while $Freq_i$ is the execution frequency of basic block `i`. This penalty can be different across SoCs. $\alpha$ is ($T_{PM}$ - $T_{PC}$), which is fixed for a given SoC. For example, we have used the Altera SoC model with NIOS II processor in our experiments, where $\alpha = 5$ ($T_{PM}$ = 6, $T_{PC}$ = 1). The summation in this

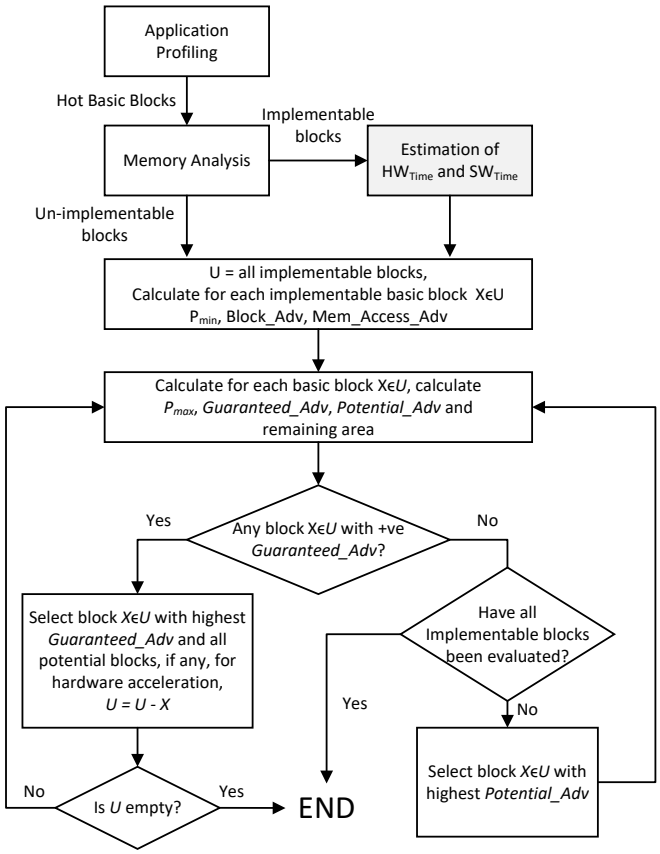



Fig. 2: Greedy selection algorithm

term denotes that we need to factor in the penalty incurred by accessing the data stored in the accelerator memories from all basic blocks currently being executed in software.

## V. Greedy Heuristic (GH) Selection Algorithm

*The main objective of our Greedy Heuristic (GH) selection algorithm is to select the best set of hardware accelerators that will minimize the overall memory access penalty, i.e. the third term in equation 1.* Ideally, all the basic blocks having common data-dependencies should be accelerated in hardware simultaneously, which then incurs zero penalty. This, however, may not be possible if there are Un-Implementable blocks or if there is a lack of available area to implement all the blocks in hardware simultaneously. Hence, in order to rapidly select a set of accelerators to minimize the overall memory access penalty, we propose our GH algorithm that aims to identify and select a set of basic blocks for hardware acceleration such that the selected accelerators have high intra-data dependency and low data-dependency with the rest of the application code.

### A. Definition of Important Terms

Before describing the proposed GH selection algorithm, we define the important terms used in the heuristic as follows:

1) **Max_Penalty ($P_{max}(\mathbf{X})$)**
The maximum access penalty ($P_{max}(\mathtt{X})$) has to be calculated for each implementable block X that is considered for hardware acceleration (taking into account the Implementable blocks that have been selected for hardware porting previously) using the third term in Equation 1.

2) **Min_Penalty ($P_{min}(\mathbf{X})$)**
The minimum access penalty ($P_{min}(\mathtt{X})$) signifies the minimum overhead incurred from the un-implementable blocks when all the implementable blocks have been ported to hardware. Hence, $P_{min}(\mathtt{X})$ needs to be calculated for each of the implementable block X only once (assuming all the other implementable blocks have been ported to hardware) using the third term in Equation 1.

3) **Block Advantage ($Block\_Adv(\mathbf{X})$)**
This is the general advantage for implementing a section of code X in hardware and is calculated as the difference of its software and hardware runtimes multiplied by its execution frequency using the first term in Equation 1.

4) **Memory Access Advantage ($Mem\_Access\_Adv(\mathbf{X})$)**
This quantifies the penalty reduction of moving a basic block X to hardware, as X can now access existing accelerator memories from previously selected hardware blocks. This is calculated using Equation 2 for each block X that is selected for hardware execution. We denote S as the set of previously selected hardware blocks that has local memories which can be accessed by X.

$$Mem\_Access\_Adv(X) = \sum_{(i=1)}^{(i=|S|)} (\alpha * m_{X \cap i} * Freq_X) \quad (2)$$

,where $m_{X \cap i}$ & $Freq_X$ denote the number of memory operations shared between basic blocks X and i, i∈S and the frequency of basic block X, respectively.

5) **Guaranteed Advantage ($Guaranteed\_Adv(\mathbf{X})$)**
This is the net advantage of accelerating a basic block X in hardware while considering all the penalties and advantages:

$$\begin{aligned} Guaranteed\_Adv(\mathtt{X}) = Block\_Adv(\mathtt{X}) + \\ Mem\_Access\_Adv(\mathtt{X}) - P_{max}(\mathtt{X}) \end{aligned} \quad (3)$$

6) **Potential Advantage ($Potential\_Adv(\mathbf{X})$)**
This is the maximum advantage that can be obtained by implementing a basic block X in hardware. It assumes that all the Implementable blocks have been selected for hardware implementation and is calculated as follows:

$$Potential\_Adv(\mathtt{X}) = Block\_Adv(\mathtt{X}) - P_{min}(\mathtt{X}) \quad (4)$$

### B. Greedy Heuristic (GH) Algorithm

In this subsection, we discuss the proposed greedy heuristic algorithm for selecting hardware accelerators. The proposed algorithm is described with the help of the Secure Hash Algorithm (SHA) from the MiBench benchmark suite [7]. Figure 2 shows the flowchart for the proposed algorithm.

TABLE II: Most frequently executed basic blocks in SHA

| Basic Block Name | Dynamic Frequency |
|---|---|
| sha_tranform_bb3 | 311872 |
| sha_tranform_bb6 | 97460 |
| sha_tranform_bb10 | 97460 |
| sha_tranform_bb13 | 97460 |
| sha_tranform_bb16 | 97460 |
| sha_tranform_bb5 | 4873 |

TABLE III: Hot blocks in SHA application and their metrics

| Information about the Hardware Block: Name, Frequency, SW/HW time, HW Size etc | | | | | | Memories Accessed with their size | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Basic block Name | Short Name | Frequency | HW Time | SW Time | HW Size (Logic Blocks) | W | A | B | C | D | E |
| sha_tranform_bb3 | Block 3 | 311872 | 12 | 21 | 10 | 5 | 0 | 0 | 0 | 0 | 0 |
| sha_tranform_bb6 | Block 6 | 97460 | 6 | 25 | 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| sha_tranform_bb10 | Block 10 | 97460 | 6 | 25 | 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| sha_tranform_bb13 | Block 13 | 97460 | 6 | 25 | 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| sha_tranform_bb16 | Block 16 | 97460 | 6 | 25 | 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| | | | | | | | | | | | |
| sha_tranform_bb5 | Block 5 | 4873 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| Mem_Size | | 0 | 0 | 0 | 0 | 320 | 32 | 32 | 32 | 32 | 32 |

The SHA application is first profiled to identify the most frequently executed basic blocks. Table II lists the 6 most frequently executed basic blocks in the SHA application, which constitute 87.85% of the total runtime.

Next, memory analysis is performed to identify the Implementable and Un-Implementable basic blocks. The first 5 hot basic blocks only contain statically allocated arrays and therefore are identified as Implementable blocks. The last basic block, sha_transform_bb5, needs data to be loaded from dynamically resolved locations, and hence is considered an Un-Implementable basic block in the proposed technique.

Various metrics for the Implementable basic blocks such as their hardware and software execution time and size of various arrays/variables are obtained using existing approaches described in [22] and [28]. Table III shows the SHA application with its most frequently executed basic blocks, estimated metrics, various arrays/variables, their size and the static access frequencies.

The proposed GH algorithm aims to select an Implementable basic block in every iteration with the highest $Guaranteed\_Adv$ for hardware implementation. If there are no basic blocks with positive $Guaranteed\_Adv$, then the basic block with the highest potential advantage ($Potential\_Adv$) is temporarily selected as a potential block. In the subsequent iterations, the potential basic blocks are assumed to have been ported to hardware and the algorithm attempts to find other suitable basic blocks for hardware implementation. Once a new basic block with positive $Guaranteed\_Adv$ is selected, the potential basic blocks are also selected along for hardware acceleration. At the end of each iteration the overall performance and area utilization of the selected hardware accelerators are calculated. The process continues till all the Implementable basic blocks have been evaluated for positive $Guaranteed\_Adv$ with or without the help of potential basic blocks in the previous iterations. It should be noted that it is indeed possible to evaluate all Implementable basic blocks and still find that some blocks do not have any positive $Guaranteed\_Adv$ even after considering all the potential basic blocks in the previous iterations. In such cases, these basic blocks will not be selected for hardware acceleration. This is the main strength of our work since the proposed algorithm evaluates these scenarios and avoids selecting blocks that can eventually result in worse performance. At the end of the selection algorithm, a performance-area curve is obtained to aid the designer in making an informed hardware-software partitioning decision.

The selection outcomes for the SHA application in every iteration is shown in Table IV. Based on the memory dependency across basic blocks as shown in Table III, it is evident that Block 3 is selected in the first step since it has the least memory dependency. In the second iteration, no basic block is selected, since there is an overwhelming memory access penalty and Block 16 is selected as a potential block. In the subsequent iterations, Block 16 is assumed to be a selected basic block and hence no memory access penalty is incurred for this block. In the third iteration, even after selecting two potential basic blocks, i.e. Block 13 and 16, the penalty is still more than the overall advantage and therefore no additional blocks are selected for implementation. In the fourth iteration, Blocks 16, 13 and 10 can be selected together since their combined advantage exceeds the penalty from the other blocks. The last remaining basic block, Block 6, is selected in the last (fifth) iteration.

TABLE IV: Selected basic blocks in each iteration for SHA

| Iteration | Guaranteed Blocks | Potential Blocks | Selected blocks |
|---|---|---|---|
| 1 | Block 3 | NILL | Block 3 |
| 2 | NILL | Blocks 16 | Block 3 |
| 3 | NILL | Blocks 13 & 16 | Block 3 |
| 4 | Blocks 13, 16 & 10 | NILL | Blocks 3, 16, 13 & 10 |
| 5 | Block 6 | NILL | Blocks 3, 16, 13, 10 & 6 |

**Exhaustive Search Algorithm:** In order to evaluate the benefits of the proposed GH algorithm, we implemented an exhaustive search algorithm that obtains the *exact* results while considering the data dependencies as proposed in our work. In the exhaustive algorithm, combination of *upto* $n$ *blocks* are evaluated in the `n`-th iteration. For example in the first iteration, one block with the best positive performance is selected. This selection is based on the assumption that the rest of the blocks are in software, which may incur some memory access penalty. Only when the advantage of implementing this combination of blocks in hardware is greater than the penalties from all the other blocks, we select the block for hardware implementation. The exhaustive search proceeds by evaluating all combinations of upto `n` basic blocks in the `n`-th iteration in the same manner, and choosing the best combination in each iteration. It should be noted that if no combination of `n` blocks provides better performance than the combination of blocks selected in the `(n-1)`-th iteration, then no new block is selected in the `n`-th iteration.

TABLE V: Hot blocks in a sample application and their metrics

| Information regarding the Hardware Block: Name, Frequency, SW/HW time, HW Size etc | | | | | Name of memory blocks accessed with their size | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Frequency | HW Time | SW Time | HW Size (Logic Blocks) | A | B | C | D | E | F |
| Block 0 | 311872 | 12 | 21 | 320 | 5 | 0 | 0 | 0 | 0 | 0 |
| Block 1 | 207460 | 6 | 25 | 192 | 1 | 1 | 0 | 0 | 1 | 0 |
| Block 2 | 157460 | 6 | 25 | 192 | 0 | 1 | 1 | 0 | 0 | 0 |
| Block 3 | 97460 | 6 | 25 | 192 | 0 | 0 | 1 | 1 | 0 | 0 |
| Block 4 | 97460 | 6 | 25 | 192 | 2 | 2 | 0 | 0 | 1 | 1 |
| Block 5 | 37460 | 6 | 25 | 192 | 10 | 0 | 0 | 0 | 0 | 5 |
| | | | | | | | | | | |
| Block 6 | 4873 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| Mem_Size | | | | | 320 | 32 | 32 | 32 | 32 | 32 |

Figure 3 shows the performance comparison for the SHA application from the MiBench [7] benchmark suite using the proposed GH and the exhaustive algorithms. These results are obtained by using equation 1 to estimate the number of clock cycles saved after using the accelerators recommended by the proposed GH and the exhaustive algorithms. In iterations 2 and 3, no new blocks are selected due to the huge penalties incurred. Overall the GH algorithm accurately tracks the exhaustive results, while being significantly faster.

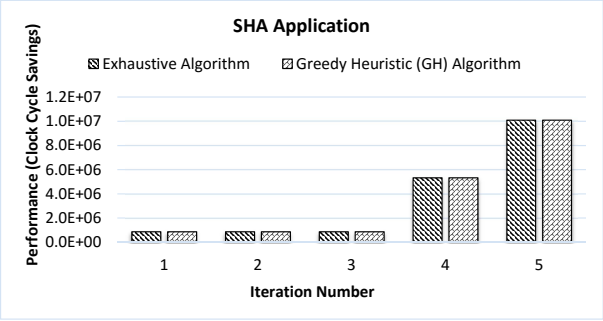



Fig. 3: Performance of Exhaustive and GH algorithms

## VI. Two-Pass Greedy Heuristic (TP-GH) Selection Algorithm

While the proposed GH algorithm in the previous section can identify suitable accelerators orders of magnitude faster than the exhaustive approach, it fails to provide accurate results in certain scenarios. In this section, we use a sample application to first demonstrate this problem and highlight the cause of this inaccuracy in the GH algorithm. Finally, we propose an extension, Two-Pass Greedy Heuristic (TP-GH) algorithm, to the original GH algorithm in an effort to overcome this limitation.

Table V shows the hot blocks of a sample application, with 6 Implementable blocks and 1 Un-implementable block. There are a total of 6 memory variables (*A, B, C, D, E* and *F*) that these blocks can access. The corresponding memory access frequencies are also shown in the table. Table VI compares the performance (clock cycle savings) of the proposed GH and exhaustive algorithms for the sample application. It can be observed that after the second iteration, the performance between the two methods begin to deviate, i.e. the GH algorithm fails to recommend the best set of hardware accelerators. In order to explain this behavior let us consider the blocks selected by

TABLE VI: Performance comparison of exhaustive and greedy algorithm

| | | | | | | |
|---|---|---|---|---|---|---|
| Exhaustive | 1.02E+6 | 2.76E+6 | 4.93E+6 | 8.45E+6 | 1.17E+7 | 1.40E+7 |
| Greedy | 1.02E+6 | 2.76E+6 | 2.76E+6 | 7.19E+6 | 1.05E+7 | 1.40E+7 |

TABLE VII: Selected basic blocks in each iteration

| Iteration | Two-Pass | | | Exhaustive |
|---|---|---|---|---|
| | Guaranteed Basic Blocks | Potential Blocks | Selected Blocks | Selected Blocks |
| 1 | Blk 3 | NILL | Blk 3 | Blk 3 |
| 2 | Blk 2 | NILL | Blks 3,2 | Blks 3,2 |
| 3 | Blk 0 | NILL | Blks 3,2,0 | Blks 0,1,4 |
| 4 | Blk 1 | NILL | Blks 3, 2, 0 & 1 | Blks 0, 1, 4 & 5 |
| 5 | Blk 4 | NILL | Blks 3, 2, 0, 1 & 4 | Blks 0, 1, 2, 4 & 5 |
| 6 | Blk 5 | NILL | Blks 3, 2, 0, 1, 4 & 5 | Blks 3, 2, 0, 1, 4 & 5 |

the GH and exhaustive search algorithms in each iteration as shown in Table VII.

The GH algorithm selects Block 3, Block 2 and Block 0 in the first three iterations. Based on Table VII, we can see that in the third iteration, the GH algorithm selects Blocks 0, 2 and 3, while the exhaustive method selects Blocks 0, 1 and 4. This means that the exhaustive search is able to select a set with different blocks leading to better performance as compared to the GH algorithm. The GH algorithm fails to select the best set as it has committed to the selection of Block 3 in the first iteration, thereby leading to a local maxima. In order to avoid getting stuck in such local maximas, we propose to use a second pass that refines the results from the greedy algorithm. The next subsection discusses this two-pass method that leads to better selection results.

### A. Two-pass Greedy Heuristic (TP-GH) Algorithm

This section presents our two-pass greedy heuristic (TP-GH) search algorithm to overcome the limitation of the GH algorithm proposed in Section V-B.

$$\boxed{\text{Blk 3} \Rightarrow \text{Blk 2} \Rightarrow \text{Blk 0} \Rightarrow \text{Blk 1} \Rightarrow \text{Blk 4} \Rightarrow \text{Blk 5}}$$

Fig. 4: Thread of sample application from greedy algorithm

In each iteration, the original GH algorithm selects a new guaranteed and/or potential block that reduces the overall memory penalties taking into account the selected blocks in the previous iterations. Let us denote a *"thread" as the sequence*

TABLE VIII: Guaranteed advantage of all blocks in the sample application

| Name | Freq. | HW Time | SW Time | HW Size (Logic Blocks) | Block Adv. | Max. Penalty | Guaranteed Advantage | Min. Penalty | Potential Advantage |
|---|---|---|---|---|---|---|---|---|---|
| Block 0 | 311872 | 12 | 21 | 320 | 2806848 | 3884900 | -1078052 | 0 | 2806848 |
| Block 1 | 207460 | 6 | 25 | 192 | 3941740 | 12942330 | -9000590 | 48730 | 3893010 |
| Block 2 | 157460 | 6 | 25 | 192 | 2991740 | 2547930 | 443810 | 48730 | 2943010 |
| Block 3 | 97460 | 6 | 25 | 192 | 1851740 | 836030 | 1015710 | 48730 | 1803010 |
| Block 4 | 97460 | 6 | 25 | 192 | 1851740 | 14578595 | -12726855 | 73095 | 1778645 |
| Block 5 | 37460 | 6 | 25 | 192 | 711740 | 10320365 | -9608625 | 24365 | 687375 |
| | | | | | | | | | |
| Block 6 | 4873 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Mem_Size | | | | | 320 | 32 | 32 | 32 | 32 |

TABLE IX: Combinations of blocks evaluated in each iteration using Two-Pass and exhaustive algorithms

| Iteration | Two-Pass | | Exhaustive |
|---|---|---|---|
| | Combinations of Nodes Evaluated | Selected Nodes | Selected Nodes |
| 1 | (3); (2); (0); (1); (4); (5) | Block 3 | Block 3 |
| 2 | (3,2); (2,0); (0,1); (1,4); (4,5) | Blocks 3,2 | Blocks 3,2 |
| 3 | (3,2,0); (2,0,1); (0,1,4); (1,4,5) | Blocks 0,1,4 | Blocks 0,1,4 |
| 4 | (3,2,0,1); (2,0,1,4); (0,1,4,5) | Blocks 0, 1, 4 & 5 | Blocks 0, 1, 4 & 5 |
| 5 | (3,2,0,1,4); (2,0,1,4,5) | Blocks 0, 1, 2, 4 & 5 | Blocks 0, 1, 2, 4 & 5 |
| 6 | (3,2,0,1,4,5) | Blocks 3, 2, 0, 1, 4 & 5 | Blocks 3, 2, 0, 1, 4 & 5 |

*of guaranteed and potential basic blocks that are selected by the GH algorithm. We also denote each basic block in the thread as a "node"*. It is evident that generally, the consecutive nodes in the thread have larger memory dependencies than the nodes which are further apart. Based on Table VII, the thread for the sample application is shown in Figure 4.

In the two-pass greedy heuristic approach, a thread is first created using the GH algorithm described earlier in section V-B. In the second pass, a "sliding window" is moved along this thread to identify the best combination of blocks in an iterative manner. The size of the window is kept equal to corresponding iteration. For example, in the third iteration, the length of the sliding window is three in order to identify all combinations of three nodes. It is noteworthy that *since the first pass aligns the blocks based on their memory dependency, the second pass has a much greater probability of selecting the best set of candidates for any combination*.

Table IX shows the combinations of nodes obtained in each iteration. The combination are selected from nodes that lie *adjacent* along the thread in Figure 4 generated from the first pass. In each iteration, the combination that leads to the best performance is selected. It can be observed from Table IX, that the blocks selected using the two-pass approach matches the results of the exhaustive algorithm for the sample application.

## VII. Multi-thread Two-Pass Greedy Heuristic (MT-TP-GH) Selection Algorithm

The two-pass TP-GH approach discussed in the previous section overcomes the problem of local maxima in the GH algorithm to some extent. However, it is still dependent on a single thread created by the GH algorithm. Hence, if the thread created using the GH algorithm in the first pass is grossly erroneous, then the second pass may not be able to find the best solutions. In order to overcome this limitation, we propose a Multi-thread Two-Pass Greedy Heuristic (MT-TP-GH) selection algorithm in this section that applies the second pass on multiple threads to identify the best candidates for acceleration.

We will describe the multi-thread approach based on the sample application discussed in Section VI. This approach works in three steps. In the first step, all the implementable basic blocks are arranged in decreasing order of their guaranteed advantage. Table VIII shows the guaranteed advantage of the various blocks for the sample application. Hence, from Table VIII, the sequence of sorted basic blocks is 3, 2, 0, 1, 4, 5. This sequence, $\mathcal{S}$, of nodes will form the starting node of new threads in the second step.

The second step of the proposed MT-TP-GH algorithm works iteratively. In each iteration, the GH algorithm is used to create a new thread by using one node from sequence $\mathcal{S}$, obtained from the first step, as the first node in the new thread. For the sample application, the various threads generated in the six iterations are shown in Figure 5. It should be noted that the maximum number of threads is equal to the number of Implementable blocks in the application.

| | |
|---|---|
| Thread 1 | Blk 3 ⇒ Blk 2 ⇒ Blk 0 ⇒ Blk 1 ⇒ Blk 4 ⇒ Blk 5 |
| Thread 2 | Blk 2 ⇒ Blk 3 ⇒ Blk 0 ⇒ Blk 1 ⇒ Blk 4 ⇒ Blk 5 |
| Thread 3 | Blk 0 ⇒ Blk 1 ⇒ Blk 4 ⇒ Blk 5 ⇒ Blk 2 ⇒ Blk 3 |
| Thread 4 | Blk 1 ⇒ Blk 4 ⇒ Blk 0 ⇒ Blk 5 ⇒ Blk 2 ⇒ Blk 3 |
| Thread 5 | Blk 5 ⇒ Blk 0 ⇒ Blk 1 ⇒ Blk 4 ⇒ Blk 2 ⇒ Blk 3 |
| Thread 6 | Blk 4 ⇒ Blk 1 ⇒ Blk 0 ⇒ Blk 5 ⇒ Blk 2 ⇒ Blk 3 |

Fig. 5: Multiple threads of the sample application

After generating all the threads, the third step of the MT-TP-GH algorithm applies the 'sliding window' concept proposed in the TP-GH algorithm (Section VI) on each of the threads to obtain the best combination of hardware accelerators of different size, for example, best combination of two blocks, three blocks, etc. Lastly, the best combination of each size amongst all the threads is identified as the combination with the best performance of a given size and is taken as the final solution for that size.

## VIII. RESULTS

We used applications from the popular CHStone [29] benchmark suite to evaluate the proposed heuristic-based selection algorithms. All experiments were executed on a virtual machine running Ubuntu 12.04, 32bit OS on a single core of Intel Xeon CPU Es-1650 V2 at 3.5GHz with 2GB RAM.

In order to compare the accuracy of the proposed method, we also implemented two other selection algorithms namely, Exhaustive and Knapsack. Figure 6 plots the performance, estimated using equation 1, in terms of clock cycles saved after accelerating a set of blocks in hardware, as recommended by the three selection methods:

1) Exhaustive Algorithm (Blue line): As explained in Section V), the Exhaustive algorithm evaluates combinations of *upto* `n` *blocks* in the `n`-th iteration to identify the best set of blocks with highest performance while considering the data dependencies. Since, the exhaustive algorithm provides *exact* results, the outcome from this algorithm serves as the baseline to compare the accuracy of the results obtained from the other heuristic-based methods.
2) Knapsack Algorithm (Grey line): Knapsack is a well known heuristic-based selection algorithm that uses the available area to select the best set of candidates for hardware acceleration [30] [31]. In this work, we used the area constraint for every iteration equal to the area required by the blocks recommended by the Exhaustive algorithm for the corresponding iteration. For example, during iteration 4, where a maximum of 4 blocks can be selected for acceleration in the Exhaustive method, we obtained the area required to implement the blocks recommended by the Exhaustive method. This area value was used in the Knapsack to identify a set of blocks for acceleration while considering the Block Advantage term from equation 1. Hence, the Knapsack algorithm is oblivious to data dependencies across candidate blocks.
3) Proposed Multi-thread Two-Pass Greedy Heuristic Algorithm (Orange line): This is the MT-TP-GH algorithm proposed method after using the Multi-Pass technique proposed in Section VII.

The X-axis shows the maximum number of hardware blocks selected in every iteration, as explained in Section V. The grey curve in the plots depicts the results obtained from the Knapsack algorithm. As can be observed from Figure 6, the blocks recommended by the Knapsack algorithm do not achieve similar performance as the ones selected by the Exhaustive or the Proposed methods. In this heuristic, the data dependencies between the candidate accelerator blocks is not considered during selection. Hence, the chosen set of accelerators frequently encounter more penalties than the advantage gained by hardware acceleration. As seen in Figure 6, the Knapsack results often show negative cycles saved, implying that the penalties from the blocks executing in software outweigh any advantage gained through hardware acceleration. This in turn implies that the overall execution time of the application can be even slower than that of a pure software-only solution.

The orange curve shows the results obtained by the proposed MT-TP-GH algorithm explained in Section VII. It can be observed that in all the applications and for every iteration, the blocks recommended by the proposed method closely match the performance obtained by the blocks recommended by the time-consuming Exhaustive method. Columns 2 and 3 of Table X list the average relative performance (with respect to the Exhaustive algorithm), for each application, as achieved by the accelerators from Knapsack and Proposed algorithms, respectively. The performance achieved by the accelerators recommended by Knapsack algorithm varies widely as confirmed by this table as well as the grey curves in Figure 6. As discussed earlier, a negative relative performance implies that the overall execution time after using the accelerators is worse than even a software-only solution. On the other hand, the results in Column 3 of Table X confirm that on average for all applications, the MT-TP-GH algorithm achieves close to 99% of the performance achieved by the Exhaustive algorithm.

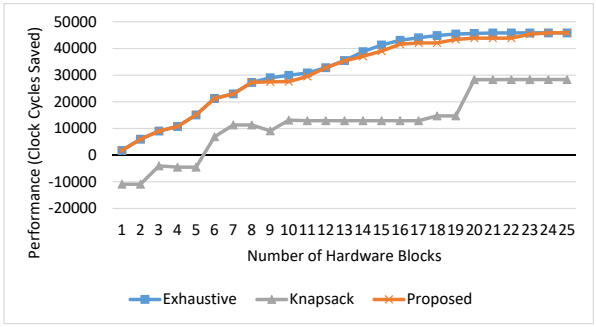
TABLE X: Relative Performance (%) of Knapsack and Proposed Algorithms & Execution Time (in seconds) of Exhaustive, Knapsack and Proposed Algorithms.

| | **Relative Performance (%)** | | **Execution Time (s)** | | |
|---|---|---|---|---|---|
| Application | **Knapsack** | **Proposed** | **Exhaustive** | **Knapsack** | **Proposed** |
| **ADPCM** | -2 | 98 | 5901.488 | 0.035 | 2.491 |
| **AES** | -208 | 100 | 2467.06 | 0.023 | 1.035 |
| **BLOWFISH** | 66 | 93 | 2348.731 | 0.038 | 0.985 |
| **DFADD** | 34 | 99 | 3825.721 | 0.014 | 1.592 |
| **DFDIV** | -39 | 99 | 3465.581 | 0.006 | 1.456 |
| **DFMUL** | -62 | 100 | 2598.657 | 0.005 | 1.087 |
| **DFSIN** | 36 | 99 | 11594.4 | 0.005 | 4.899 |
| **GSM** | 19 | 100 | 7729.674 | 0.008 | 3.262 |
| **MOTION** | -107 | 100 | 966.964 | 0.062 | 0.723 |
| **SHA** | -53 | 100 | 2837.592 | 0.023 | 1.182 |
| **Average** | **-32** | **99** | **4373.586** | **0.0219** | **1.8712** |

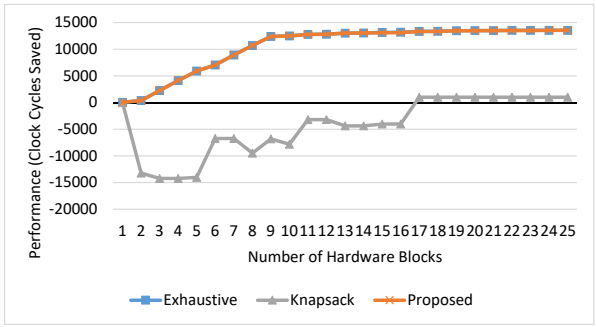
To compare the runtime of these selection algorithms, we measured their execution time on our experimental platform for all the iterations shown in Figure 6. Columns 4 through 6 in Table X show this execution time comparison. As expected, the Knapsack and Proposed algorithms run significantly faster than the Exhaustive algorithm. However, while the Knapsack algorithm runs faster than even the Proposed algorithm, it can be observed from Columns 2 and 3 of Table X as well as Figure 6, that the blocks selected by the Knapsack algorithm achieve significantly worse performance than the Proposed method. On the other hand, the Proposed algorithm achieves similar performance as the Exhaustive algorithm but at a significantly faster rate. This proves the effectiveness of our Multi-thread Two-Pass Greedy Heuristic algorithm.

## IX. CONCLUSION

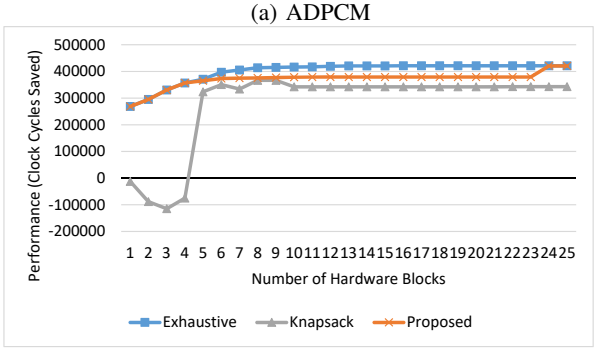
We proposed a novel algorithm to automatically and rapidly select suitable application segments for hardware acceleration while taking into consideration their data dependencies. The architecture model adopted in this paper enables the implementation of accelerator memories that do not require DMA invocations and cache coherency systems, hence reducing the complexity of data sharing between the processor and accelerator memories. To rapidly select profitable hardware accelerators, we proposed a heuristic-based greedy selection algorithm and further refinements to closely match the accuracy of an Exhaustive algorithm. Experimental results using applications from the CHStone benchmark suite show that the proposed MT-TP-GH approach recommends hardware accelerators that achieve comparable results to an Exhaustive search method



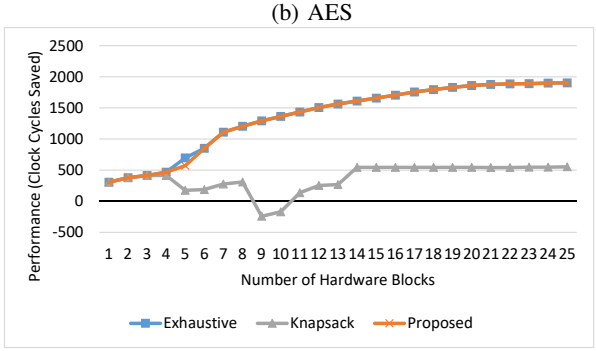

(a) ADPCM



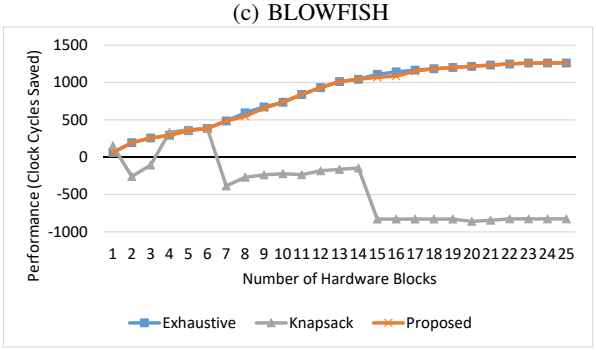

(b) AES



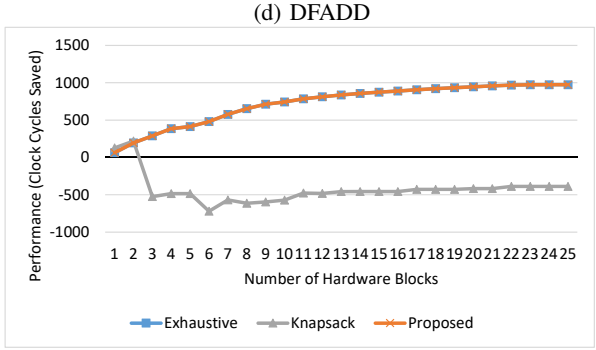

(c) BLOWFISH



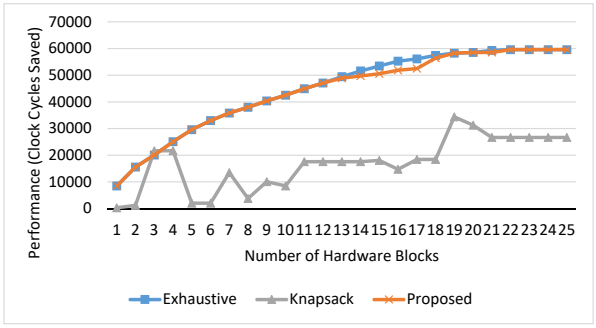

(d) DFADD



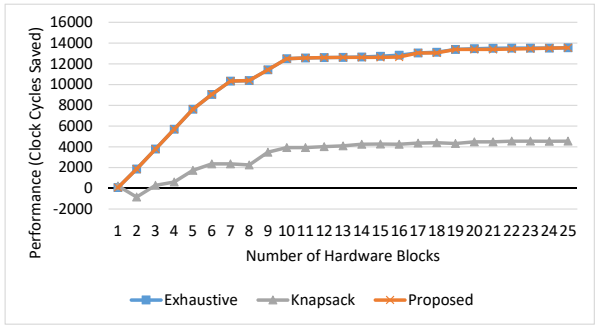

(e) DFDIV



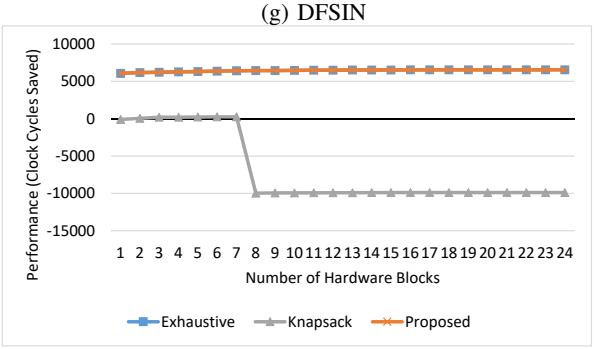

(f) DFMUL



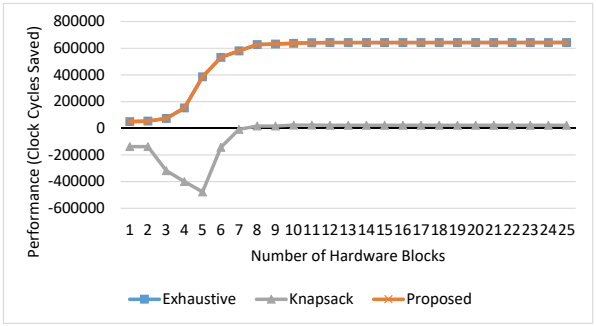

(g) DFSIN




(h) GSM




(i) MOTION




(j) SHA

Fig. 6: Performance comparison between various methods

with close to 99% accuracy while being orders of magnitude faster, hence lending itself well to rapid design space exploration at an early stage. In future, we plan to incorporate cache models to further refine our penalty estimation model.

## REFERENCES


[1] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward Cache-Friendly Hardware Accelerators," in *Proceedings of the Sensors to Cloud Architectures Workshop (SCAW), with HPCA, 2015*.

[2] "Zynq-7000 All Programmable SoC, [Online]. Available: (http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm)."

[3] "Altera SoC FPGAs, [Online]. Available: (http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html)."

[4] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding Sources of Inefficiency in General-purpose Chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 37–47.

[5] A. Prakash, C. Clarke, and T. Srikanthan, "Custom instructions with local memory elements without expensive dma transfers," in *Field Programmable Logic and Applications (FPL), 2012 22nd*.

[6] A. Prakash, C. Clarke, S. Lam, and T. Srikanthan, "Modelling communication overhead for accessing local memories in hardware accelerators," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, 2013, pp. 31–34.

[7] "Mibench 1.0, [Online]. Available:http://www.eecs.umich.edu/mibench/."

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[9] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 2, pp. 273–289, 2001.

[10] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 11, pp. 2035–2045, 2007.

[11] R. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE*, vol. 20, no. 2, pp. 60–70, Mar/Apr 2000.

[12] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," in *Design, Automation and Test in Europe, 2006. DATE '06.*, 2006.

[13] T. Kluter, S. Burri, P. Brisk, E. Charbon, and P. Ienne, "Virtual ways: efficient coherence for architecturally visible storage in automatic instruction set extensions," in *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 126–140.

[14] T. Kluter, P. Brisk, P. Ienne, and E. Charbon, "Speculative dma for architecturally visible storage in instruction set extensions," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, ser. CODES+ISSS, 2008, pp. 243–248.

[15] J. Cong and K. Gururaj, "Architecture support for custom instructions with memory operations," in *ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '13, 2013, pp. 231–234.

[16] "ARM Ltd. [Online]. Available:(http://www.arm.com)."

[17] "Synopsys. [Online]. Available:(http://www.synopsys.com/home.aspx)."

[18] "Introduction to AMBA 4 ACE. - Whitepaper, [Online]. Available: (http://www.arm.com/files/pdf/cachecoherencywhitepaper_6june2011.pdf)."

[19] "AMBA 4 ACE (AXI Coherency Extensions) Verification IP. [Online]. Available:(http://www.techdesignforums.com/practice/technique/amba4-ace-cache-coherency-vip/)."

[20] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "Kahrisma: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 819–824.

[21] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, "mRTS: Run-time system for reconfigurable processors with multi-grained instruction-set extensions," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*.

[22] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[23] S. Yousuf and A. Gordon-Ross, "An automated hardware/software co-design flow for partially reconfigurable fpgas," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2016, pp. 30–35.

[24] C. Zhang, Y. Ma, and W. Luk, "Hw/sw partitioning algorithm targeting mpsoc with dynamic partial reconfigurable fabric," in *2015 14th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)*, Aug 2015, pp. 240–241.

[25] J. W. Tang, Y. W. Hau, and M. Marsono, "Hardware/software partitioning of embedded system-on-chip applications," in *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2015, pp. 331–336.

[26] "LLVM, Compiler Infrastructure, [Online]. Available:(http://llvm.org/)."

[27] T. M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01, 2001, pp. 191–202.

[28] S.-K. Lam and T. Srikanthan, "Rapid design of area-efficient custom instructions for reconfigurable embedded processing," *J. Syst. Archit.,09*.

[29] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[30] A. Ray, W. Jigang, and S. Thambipillai, *Knapsack Model and Algorithm for HW/SW Partitioning Problem*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–205.

[31] J. Wu, T. Srikanthan, and G. Chen, "Algorithmic aspects of hardware/software partitioning: 1d search algorithms," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 532–544, April 2010.


**Alok Prakash** is currently a Research Fellow at the School of Computer Science and Engineering, Nanyang Technological University, Singapore, from where he also received his PhD degree in 2014. Prior to this, he was a Research Fellow at the School of Computing, National University of Singapore. His research interests include developing Customized Application Specific Platforms, Custom Instructions and Processor Selection & Customization.

**Christopher T Clarke** received a BEng degree in Engineering Electronics and a PhD degree in Computer Science from the University of Warwick in 1989 and 1994 respectively. From 1994 to 1997 he lectured at Nanyang Technological University in Singapore where he was a cofounder of the Centre for High Performance Embedded Systems (CHiPES). Since then he has spent time in industry, both as an in-house engineering manager and independent consultant for UK silicon design houses, system integrators and multinationals such as Philips Semiconductors. He later joined the Microelectronics and Optoelectronics research group in the Department of Electronic and Electrical Engineering at the University of Bath in March 2003 where he has been involved with many bio-medical and bio-mimetic European Union-funded research projects. He is a member of the Centre for Advanced Sensor Technologies (CAST).

**Siew-Kei Lam** is an Assistant Professor at the School of Computer Science and Engineering, Nanyang Technological University (NTU), Singapore. He received the B.A.Sc. (Hons.) degree and the M.Eng. degree in Computer Engineering from NTU, Singapore. His research interests include embedded system design algorithms and methodologies, algorithms-to-architectural translations, and high-speed arithmetic units.

**Thambipillai Srikanthan** joined Nanyang Technological University (NTU), Singapore in 1991. At present, he is a Professor and the Executive Director of the Cyber Security Research Centre @ NTU (CYSREN). Prior to this, he was the Chair of the School of Computer Engineering at NTU. He founded CHiPES in 1998 and elevated it to a university level research centre in February 2000. He has published more than 250 technical papers. His research interests include design methodologies for complex embedded systems, architectural translations of compute intensive algorithms, computer arithmetic, and high-speed techniques for image processing and dynamic routing.