Abstract—Current handhelds incorporate a variety of accelerators/IPs for improving their performance and energy efficiency. While these IPs are extremely useful for accelerating parts of a computation, the CPU still expends a significant amount of time and energy in the overall execution. Coarse grain customized hardware of Android APIs and methods, though widely useful, is also not an option due to the high hardware costs. Instead, we propose a fine-grain sequence of instructions, called a Load-to-Store (LOST) sequence, for hardware customization. A LOST sequence starts with a load and ends with a store, including dependent instructions in between. Unlike prior approaches to customization, a LOST sequence is defined based on a sequence of opcodes rather than a sequence of PC addresses or operands. We identify such commonly occurring LOST sequences within several popular apps and propose a design to integrate these customized hardware sequences as macro functional units into the CPU data-path. Detailed evaluation shows that such customized LOST sequences can provide an average of 25% CPU speedup, or 12% speedup for the entire system.

Index Terms—Mobile computing, Mobile SoC

I. INTRODUCTION

The debate between customized hardware/accelerators vs. general purpose hardware is back in the limelight [1]. It is well understood [2] that customized hardware accelerators or IPs are highly energy and performance efficient towards meeting the required computational needs of specific apps, without incurring the overheads of a general purpose von Neumann architecture. Accelerators such as video codecs, crypto engines and graphics (GFX) are extensively used in embedded and mobile devices for their specialized needs. Accelerators for some emerging functionalities such as machine learning, convolution neural networks for computer vision are also starting to make their way into high end computing systems. However, while such accelerators are very efficient for the specific purposes in some apps, they fail to fulfill the needs of many other common apps. In the very important domain of handheld devices (cell phones, tablets, etc.), the hardware already incorporates several of these IPs, which are important for apps such as Youtube (codec), virtual-reality apps/video recording (ISP) and gaming (GFX/ISP). Unfortunately, these accelerators are not as beneficial for numerous other immensely popular apps such as social networking, browsers, maps, photo, PDF, PPT viewers etc., mainly due to their coarse-grained offerings.

Even for apps that substantially utilize coarse grain accelerators, recent studies [3], [4] have reported high CPU utilization. We also find that the CPU contributes to as much as 27% of the execution time and 30% of the energy for apps such as Youtube (Fig.1, where IP utilizations are also as high as 46%), suggesting that there is still room for improvement. Towards optimizing CPU execution, a lot of past research have proposed to accelerate CPU execution by offloading hot code to specialized hardware [5], [9], custom hardware units from DFGs [10]–[12], µOps [13], [14], etc. These optimizations target throughput-oriented apps in scientific domain such as SPEC [15]. In contrast, handheld apps are user-oriented wherein there is a great deal of user interactions based on which events are processed. Note that, these events also repeat over time, giving rise to frequently executed and common functionalities. So, we first validate whether the existing methods such as hot function offloading is applicable for this handheld domain or not. Specifically, we characterize that while a few widely used Android APIs and methods do constitute a significant portion of the execution across these apps, they are not conducive for direct hardware realization due to the high associated costs. This suggests a need for drilling down to finer granularities for hardware realization.

Finer granularities, such as instruction sequences, may yield higher levels of commonalities not just across apps, but even within a single app’s execution profile. Towards exploiting the benefits from such opportunities, we seek to answer some of the largely unexplored questions in the context of mobile/handheld devices such as: How to identify these sequences? Do these sequences occur frequently enough for sufficient benefits? Can they be offloaded to a hardware without significant data transfer and synchronization costs? In general, CPUs fetch data (LOADs) from memory and perform a sequence of operations on it before sending output to memory (STOREs). Ignoring memory operations, the sequences of operations between the loads to stores are acceleration opportunities. These sequences, known as Load-to-Store (LOST) sequences in this paper, have the trade-offs below:

- Hardware realizations of very small lengths from these sequences could offer many more re-use (commonality) opportunities (either within or across apps), but the cost of data transfer and synchronization between the main core and this hardware may negate any benefits. In fact, the inability to support some functionalities in the hardware (e.g., LOAD/STORE ops are not supported in DySer [7]), can also limit the sequence lengths. As we will see, unlike

- Work was done as a student at The Pennsylvania State University.

This research is supported in part by NSF grants 1213052, 1302557, 1317560, 1320478, 1409095, 1439021, 1439057, 1626251, 1629129, 1629915, 1714389, 1526750, and Intel. We would also like to thank Jack Sampson for his feedback on this paper.
SPEC, PARSEC workloads used in [7], [10], [16], [17]. the mobile/handheld apps not only have LOADS frequently interspersed within the code, and hence preloading the data for offloading small sequences into DySer hardware is not beneficial. Also, many of these LOADS (effective addresses) are data-dependent, and it is infeasible to pre-load the data and sustain a longer chain in the accelerator as well. Hence, in our approach, the offloaded LOST sequences can include multiple (data dependent) LOADS and STORES, thereby allowing longer sequences to get accelerated with the necessary functional units for those operations.

• While larger sequences can amortize the set up and synchronization overheads, it would be more difficult to find exact matches of larger lengths repeating within or across apps. To address this concern, LOST sequences are defined purely based on the sequence of opcodes (i.e. functionality) rather than using code/PC addresses and operands. This allows more matches within/across apps than using PCs.

By profiling a diverse and popular set of mobile apps, we find promising opportunities for accelerating frequently (both within and across apps) occurring LOST sequences that have the potential to cover a significant portion of the app execution. There are diverse approaches to leveraging these sequences (static configurable accelerators, runtime reconfigurable accelerators, static customized execution unit realization, incorporating the accelerator as a separate unit/co-processor or within the main core’s data path, etc.), and we investigate one such option by synthesizing LOST sequences as a customized execution unit and incorporating this into the processor data path. By serving as one or more “macro” functional units in the processor data path, these accelerators incur very little set up and synchronization costs. Towards finding and exploiting fine-grain acceleration opportunities across diverse and popular apps, this paper makes the following contributions:

• We show the limitations of existing customized coarse-gain accelerators that still leave enough headroom for optimizations in many apps. We also show that even if Android APIs and methods do have widespread use across the apps, they are too coarse-grained for cost-effective hardware realization.

• We introduce the concept of a LOST sequence, that is fine grained enough to occur frequently within and across apps (by only specifying the sequence of operations rather than PC addresses, operands or data values), but is still long enough to not incur high setup and synchronization costs.

• Our characterization of LOST sequences shows a wide spectrum of sequence lengths and diversity in their occurrence frequencies across apps. We find five common sequences across the diverse set of apps and show that these common sequences account for as much as 54% of the dynamic instruction stream in some apps and on an average cover 31% of the dynamic instruction stream across all apps.

• We propose a hardware architecture that integrates five most frequent LOST sequences into the conventional processor data path as one or more “macro” functional units. This avoids high synchronization costs for accelerating them.

• A detailed evaluation with a common set of customized LOST sequence hardware shows CPU speedups as high as 44% with an average benefit of 25% across 15 apps.

II. COARSE GRAIN CUSTOMIZATION

In this section, we analyze a spectrum of ten diverse and popular Android apps, listed in Table V to understand which are the dominant contributors in hardware and software to subsequently optimize for performance and/or power. These apps have been ported to run on Gem5 [18] with IP models [19]. The configuration used is discussed in Sec. IV.

1) Hardware Characterization: Fig. 1 shows a breakdown of execution time and energy expended in different hardware components. We find that, while current IPs (e.g., codecs, GFX) are extremely useful in some apps (e.g., Youtube), they are not universally applicable to others (such as Maps) because of their coarse grain customization. This is evident from the overall breakdown where CPUs are still the dominant contributor of both execution time (48%) and energy (35%).

2) Software Characterization: We next present characterization results from the software perspective to examine which are the dominant software portions in the execution? and how common/frequent are these dominant portions not just within one app, but across apps? Prior studies have looked at profiling these APIs and/or individual methods during a single (or a group of related) app’s execution (e.g., [20]–[23]) in order to develop app/domain specific hardware [20], [24], whereas below we examine the execution characteristics at two granularities of the software namely, APIs and methods, in the ten selected apps. Beyond studying their contribution to each app, we are also interested in their importance across apps. We discuss these questions with the results in Table I that shows top APIs and methods by execution coverage for each app and the top APIs and methods common across apps (last row). In this paper, we use coverage to denote the percentage of total dynamic instructions executed by the app.

APIs: As can be seen, the top two APIs in Columns 2 and 4 has up to 36% (25% + 11%) coverage (in Browser). But entire Android APIs, albeit with reasonable coverage, are not amenable to ready hardware customization due to high code size (Column 5) and frequent changes to the repo (Column 2).

Methods: Even if a drill-down to individual methods is much more tractable (Column 3), without significant loss in coverage, there is no commonality in the most-executed set of methods and the most common set of methods (Columns 7 and 8) only offer 3-5% coverage (in addition to the periodic software revisions) making this alternative also un-attractive.
### TABLE I: Top 2 APIs by coverage for each app is shown in 2nd and 4th columns. 3rd column shows the 2 most invoked functions from APIs in 2nd column with their respective coverage. API Size (5th) and #Changes (6th) for both the top APIs is shown in order. The 7th and 8th columns denote the coverage from the 2 most common functions.

<table>
<thead>
<tr>
<th>Label</th>
<th>Dyn. Instruction Seq.</th>
<th>LOAD1</th>
<th>LOAD2</th>
<th>LOAD3</th>
<th>LOAD4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x = add (mem1)</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x = add x*b</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x = add ax</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x = load (mem2)</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>x = load (mem3)</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>x = load (mem4)</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>x = load x</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE II: Example to illustrate the properties of LOST. The ★ indicates the dynamic instruction is present in the corresponding sequence. LOST.2 and LOST.3 are identical sequences (load→add→store) as per our definition.

This motivates us to study the feasibility of identifying “smaller” functionalities (code blocks or even instruction sequences) in CPU execution with good coverage, and low cost of realizing an offload-hardware for them. On the surface, it may appear that the coverage offered by such small granularities, being parts of the methods studied above, would only be lower than their constituent methods. However, below we introduce a different paradigm of looking at “instruction sequences” that are identified purely on the basis of op-codes rather than the code segment/PC addresses, where these sequences reside. So, such sequences could occur at multiple places in the code, with a hardware realization offering a boost in coverage in all these occurrences, beyond that offered by the same sequence occurring at a specified code address within one method.

#### III. LOAD-TO-STORE (LOST) SEQUENCES

Rather than set granularities for hardware customization based on pre-determined software boundaries (APIs/methods), we start afresh to find out what exactly the CPU cores perform with the data (coming from memory) given to them, before they produce some output (which again percolates into memory). Understanding this functionality from the input to the output stages can better help determine granularities, rather than be governed by pre-defined software-boundaries, that are intended for other purposes. Towards this goal, we define a Load-to-Store (LOST) sequence that a core performs, starting with a LOAD from memory, until it produces output in the form of a STORE into memory, with all the intermediate functional operations performed between them (in the data flow) becoming part of this sequence. Such a sequence offers the opportunity for offloading those functionalities between the LOAD and the STORE to an accelerator. However, the number of such sequences and the size/length of these sequences can become inordinately large to be implemented in hardware. To address this concern, we pick the ones that yields the most benefit at low realization costs.

More formally, a LOST sequence starts with a LOAD instruction, and ends with a STORE instruction that is on the dependence chain of the LOAD. In between, it includes all the instructions on this dependence chain, in the same order of dependence. For instance, Table II gives 4 LOST sequences for the dynamic instruction sequence I1 to I10. The salient characteristics of a LOST chain are discussed below:

- Each subsequent instruction of a LOST chain (after the first LOAD) is dependent on the prior instruction of that chain.
- Adjacent instructions of a LOST chain do not need to be spatially adjacent either in the code segment or temporally adjacent in the dynamic execution sequence (e.g. I3 and I4 are not part of LOST.2). Hence, a single sequence (or even adjacent instructions of a sequence) can cross basic block and/or method/function boundaries.
- An instruction can simultaneously belong to multiple LOST chains (e.g. I2 belongs to both LOST.1 and LOST.2), i.e. sequences can intersect. In fact, one can even be a subset of another (e.g. LOST.3 is a subset of LOST.4).
- The same sequence of CPU functionality (i.e. add, eor, mov/shift [25], etc.) can occur in multiple places in the code segment, even if the source of the operands (and not just the data) are different. Due to our definition of an instruction (see Footnote 1), our approach will tag all these occurrences as belonging to the same LOST sequence. (e.g. LOST.2 and LOST.3 are the same sequence). Such flexibility allows greater coverage without requiring additional hardware cost.

1Note that an instruction, in our discussions, refers to the operation (i.e. opcode) performed by the CPU, and not the PC value of that instruction, or even the operands used. E.g., add i3, r2, r1 and add i4, r5, r6 are treated as the same instruction for our purposes since we are only interested in accelerating the functionality once the input data is made available.
While there has been prior work [10], [17], [26], [27] on tracking dependence chains for different optimization, to our knowledge, we are unique in discounting the PC, data and operand sources for our intended purposes. Our LOST sequence concept allows the tracking of arbitrarily long data flow amongst the computations within the CPU without regard to software boundaries. Further, without attaching PC values, data values, or even operand sources, we can identify more commonality for re-use of such functionalities across disparate pieces of code - not just within/across basic blocks, but also across methods of an app, or even across apps.

A. Methodology for Extracting LOSTs

We collect instruction traces of the app run (for roughly 2-3 minutes of execution time) in Android using the Android emulator [28]. Even an offline analysis of this execution trace is non-trivial due to the following reasons: (i) There are typically more than $10^9$ LOST sequences in each app’s trace (the number of sequences can be higher than the number of dynamic instructions); and (ii) Storage structures become voluminous in order to track not just registers, but also memory addresses, that can cause dependencies across instructions.

In the interest of space, we are not detailing all the techniques and optimizations used to address these challenges. Fig. 2a summarizes the overall Map-Reduce framework that we use to parallelize the LOST extraction process. We divide the execution trace into equal “snippets” that are farmed out to Mapper threads. Each Mapper locally finds and accumulates the LOST sequences for its portion. A subsequent Reduce phase histograms the sequences from different Mappers (similar to the well-known word count Map-Reduce app) to accumulate the frequency of LOST occurrences across the entire execution trace. We should point out that there is a possibility of a LOST sequence not being tracked when it spans multiple mappers (as in the first instruction of Fig. 2a which does not find its matching Store in the same mapper). However, we have verified that this is a reasonable tradeoff since it only misses out on catching less than 0.0015% sequences (the mappers are relatively coarse grained working with around 200K instructions each), and such sequences do not occur frequently either. Further, the hardware supports ARMv7 ISA, which encompasses ≈2k opcodes with T1ARM, T2, T2EE, NeonSIMD and VFP instruction formats. So at the mappers, we build an instruction parser supporting ARMv7 ISA to process all the opcodes seen in the execution traces.

B. Characteristics of LOST Sequences

With an offline analysis of the apps’ dynamic execution, we observe several interesting insights on LOST sequences:

- Fig. 2a plots the number of occurrences and the corresponding coverage of LOST sequences of different lengths in the Music Player app. In the interest of clarity, the x-axis only shows sequences of powers of 2, though there are sequences of all lengths in-between. We show contributions of the top 3 sequences for each length, along with the contribution of all other sequences of that length. As can be seen, we do find a wide spectrum of sequence lengths, ranging from the small single digits to as high 200k, though lengths up to 2K are shown in the Figure. There are many more sequences of smaller lengths compared to those of larger lengths, and the former has cumulatively higher contribution to the overall coverage because of their much more frequent occurrences.

- Fig. 2c shows the difference between our definition of an instruction in a sequence (i.e. without regard to PC values) compared to the traditional way [7], [10], [17], [29] of associating an address with an instruction. In the latter, the same opcode sequence appearing in different places/functions would get counted as separate sequences (shown as stacked bars for each of these sequences), and the benefits of a hardware realization for that sequence may not be as apparent. Our approach on the other hand exploits much more commonality across the entire code segment.

- Note that because of the property that one LOST sequence can be a proper subset of another, there is a tradeoff in the benefits vs. costs in realizing a superset sequence vs. a subset sequence in hardware. The former may incur a higher hardware realization cost, but may not benefit as much from the effort. In Fig. 2b we plot the number of occurrences, and the corresponding coverage, for the most common LOST sequence and its supersets (i.e. the sequences get larger as we move to the right) for Music player. There are two counter-acting factors affecting the coverage - the number of occurrences decreases as we move to supersets, but since each sequence is longer, each invocation would contribute to a larger coverage. Consequently, we see a sweet spot in each of these LOST sequences, and we have used this optimal point in our hardware realizations as will be explained next.

C. Top 5 LOST Sequences from All Apps

In Table III, we list the top 5 LOST (by coverage) sequences in the apps. We identify 5 categories of LOST sequences...
(segregated by rows) and describe their functionality in their respective app source codes in the fourth column. Recall that, one LOST may occur at different PCs or functions in the source code (Fig. 2c). So, we identify the functionality of LOSTs as the most commonly occurring ones in this table. They are network-based (Net), data-transfer between kernel and user spaces (DataTransfer), event listeners (EventListener), orientation listeners (OrientationListener), and iterators (Iterator). The instruction mnemonics for each of the above LOSTs (in third column), range from 2 to 15 instructions. But, by considering instructions as opcodes, we are able to find several LOST sequences with high coverage in many apps, with values reaching as high as 54%, making even each of them an individually attractive design choice. Note that, all these LOSTs such as Net, EventListener, etc., map to the hot functions such as SensorEventListenerImp, ReceiverDispatched, in Table I as well as other functions with similar instruction sequences. Table III also lists overlapping LOST chains from the same functionality. For example, DataTransfer(Kernel) has 7 frequently executed paths (or DFG), along with their respective coverages. This detail allows us the flexibility to optimize for only the most relevant LOSTs for all the apps and not the infrequent paths of a DFG. In the interest of space, we concentrate on our goal towards a generic acceleration unit for all apps by picking the 5 LOST sequences that are common across all the 10 apps (and are not overlapping with each other in the same DFG) in our training set shown in Table III with blue-text. We validate their generality by observing their coverage in a different set of 5 popular apps as listed in Fig. 3a. Note that, the average coverage by these "generic" LOST sequences varies between 4% to 45% and vary in length between 2 to 5 instructions only. So, any acceleration mechanism targeting these relatively small sequence of instructions can derive substantial performance benefits. We next explore the mechanisms to accelerate these LOSTs and their impacts on app performance.

### Table III: Top 5 LOST sequence across all apps:

| Category      | Name            | Details of Instruction Mnemonics | Description | Apps benefited: Underlined = top LOSTs for app
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Net</td>
<td>Net.1</td>
<td>id-st</td>
<td>Common action: All apps, top LOST for Youtube</td>
<td>All apps</td>
</tr>
<tr>
<td></td>
<td>Net.2</td>
<td>ld-mov-lid-st</td>
<td>Facebook,Maps</td>
<td>Facebook,Maps, Office, Photo, Youtube</td>
</tr>
<tr>
<td></td>
<td>Net.3</td>
<td>4% ld-mov-st</td>
<td>Facebook</td>
<td>Facebook, Maps, Office, Photo, Youtube</td>
</tr>
<tr>
<td></td>
<td>Net.4</td>
<td>ld-mov-st-lid-mov-lid-st</td>
<td>Facebook</td>
<td>Facebook, Maps, Office, Photo, Youtube</td>
</tr>
<tr>
<td></td>
<td>NetReceive</td>
<td>ld-mov-st-lid-mov-lid-st</td>
<td>Facebook</td>
<td>Facebook, Maps, Office, Photo, Youtube</td>
</tr>
</tbody>
</table>

### IV. LOST HARDWARE CUSTOMIZATION

The LOST instruction sequences, which each individually constitute as much as 54% of the dynamic instruction stream of these apps, are simple enough for hardware implementation, while simultaneously being generic enough to be useful in several popular apps. In this context, we address five pertinent questions: (i) What are the bottlenecks in the execution of a LOST sequence? This would suggest us which stage should be targeted. (ii) What is the most suitable solution for a LOST hardware acceleration? (iii) What other issues such as interspersed non-LOST instructions in a LOST sequence and control divergence are addressed? (iv) How to supply required operands for executing a LOST sequence? and (v) How to invoke the proposed LOST hardware during runtime?

#### A. Execution Time Breakdown

To better understand what needs to be accelerated (for LOST sequences), we show the profile of LOST instructions on the baseline CPU in Fig. 3B. As can be seen, no one stage is overly dominant, making us simultaneously consider several issues for speedup. An instruction spends around 15% and 16% time in the Fetch and Decode stages respectively, reiterating the well-known observation that a von-Neumann architecture is highly inefficient for executing repetitive code. Further, we see that (i) stalling for availability of functional units, and (ii) stalling for data produced from other instructions (whether in the same LOST or not), contribute to as much as 20% and 32%, respectively. Consequently, a good acceleration...
option should consider all these 4 aspects - fetch, decode, availability of functional units, and data dependencies - for extracting the maximum performance from LOSTs.

B. Exploring Existing Acceleration Solutions

Below, we discuss the suitability of four alternatives from traditional solutions for accelerating LOSTs:

**SIMD:** Having separate SIMD units [30], [31] with many functional units can be one way to address the corresponding functional unit bottlenecks. Also, with one SIMD instruction fetch/decode, one could perform multiple parallel operations to avoid the fetch/decode bottleneck. However, our LOSTs are at best a few instructions long, with dependencies flowing from one to the next (otherwise they would be part of the same LOST). Hence, there is no data parallelism for SIMD to exploit; consequently, we discard this alternative.

**Configurable/programmable offload hardware:** In order to avoid explicit instruction fetch/decode, Coarse Grained Reconfigurable/programmable Arrays have been proposed [7], [14], [17], [29], where simple functional units get connected together based on the data flow by runtime configuration. In order to accommodate the overheads of such configuration (conservatively 64 cycles [7]), and also the explicit data transfers between the main core and the programmable hardware, the offloaded functionality needs to be relatively coarse grained and repetitive. However, any one execution of a LOST is only a few instructions long, and though repetitive in the overall app execution, the separation between successive invocations is temporally disparate and the corresponding control flow graph (CFG) for each such invocation may not be the same either. Hence, we discard this alternative as well.

**IPs:** IP cores/accelerators [32], [33], are instead the ideal mechanism to allow direct data flow between these successive LOST instructions, without the bottleneck of instruction fetch and decode. However, we need these to get integrated into the processor datapath, since the overheads would be very high to interface with a coprocessor-like entity with explicit calls and data transfers just to accelerate a few ($\leq 15$) instructions.

**On-core Customized Execution Units:** We would thus like to integrate the hardware entity that performs LOST acceleration as close to the CPU datapath as possible. This will allow us to (i) invoke the hardware with minimal overhead, and (ii) reduce any data transfer back-and-forth, by facilitating resource sharing (e.g. registers) between the two. Such on-core Customized Execution Units like BERET [10] have been proposed, though for very restrictive/specific functionalities such as add-shift. We propose to use such capability for additional, but simplified functionalities that mandates to execute the LOST sequences. Further, the prior proposals leverage such specialized units only when the instructions requiring them were spatially/temporally contiguous. In our case, as already noted, even successive instructions of a LOST sequence can be separated temporally/spatially.

C. Addressing Spatial Separation of LOST Instructions

Although we use spatially separated generic LOST sequences to obtain substantial app execution coverage, we need them to be spatially contiguous for a compiler to easily replace them with an offload command. To this end, we now employ hoisting instructions between LOSTs, a well-known technique employed by compilers for various other optimizations [34]. In detail, the spatial separation of the instructions in a LOST sequence can cause four types of dependence scenarios between the LOST sequence and the non-LOST instructions surrounding them as discussed below:

**InBound (non-LOST $\rightarrow$ LOST):** Fig. 4a(a) shows an example scenario for InBound, where a non-LOST instruction (I2) only inputs value to a LOST instruction (I3). In this case, the compiler can safely move the InBound instruction (I2) after the last LOST instruction (after I4) making the LOST sequence spatially contiguous. Also, the compiler needs to insert a hint after the LOST sequence to guarantee that the LOST sequence’s results are ready. On the hardware side, the CPU will only issue I2 when the LOST is ready to commit.

**Performance Impact:** As I2 is hoisted, its output is available earlier; I1’s start is also delayed. So, the execution of subsequent instructions may be affected based on these hoists.

**OutBound (LOST $\rightarrow$ non-LOST):** Fig. 4a(b) shows an example scenario for OutBound, where a LOST instruction (I1) only inputs value to a non-LOST instruction (I2). In this case, the compiler can safely move the OutBound instruction (I2) after the last LOST instruction (after I4) making the LOST sequence spatially contiguous. Also, the compiler needs to insert a hint after the LOST sequence to guarantee that the LOST sequence’s results are ready. On the hardware side, the CPU will only issue I2 when the LOST is ready to commit.

**Performance Impact:** As I1, I3, I4 are hoisted, their outputs are available earlier; I2’s start is also delayed. So, the execution of subsequent instructions may be affected based on these hoists.

**Bulge:** Fig. 4a(c) shows a scenario for Bulge, where a LOST instruction (I1) inputs value to a non-LOST instruction (I2), and I2 inputs value to a subsequent LOST instruction (I3). Here, I2 cannot be hoisted, and so we do not accelerate such scenarios.

**Branch:** There are three types of control divergence scenarios. First, there could be a branch from outside the LOST sequence, wherein existing CPU mechanisms like ROB squashes handle the correct path execution. Second, a branch in the LOST sequence also is acceleratable because the acceleration logic will take the required decision and change the control flow. For our five generic LOST sequences, this does not occur. The third case is where a non-LOST branch occurs between two instructions in a LOST sequence as depicted in Fig. 4a(d), where there is Branch, between two LOST instructions (I1 and I3). In this case, we use compiler support to profile the most frequent path in runtime, and convert the control-flow to assert-style data-flow as proposed in [35] and used by [10], [17] and apply the above steps as applicable.

The LOST sequence may now become contiguous in most cases and thus enables, one-shot execution in these scenarios. This implies that the LOST sequence can be merged as a macro instruction that can be invoked at runtime. But to facilitate this one-shot acceleration, we need the required
operands for all the instructions which is described next.

D. Encoding and Supplying Operands for LOST Execution

Recall that, LOST sequences are obtained by accounting for only the instruction opcodes without considering their operands. Although this helped us cover more instructions using a LOST sequence, to execute them, we need the corresponding LOST hardware unit to know the exact order of operands (both input/output of registers or memory addresses or immediate values) needed by each of its constituent instructions. We term this as operand-list in this work. To encode (and subsequently supply) the operands for LOST executions, there are two issues: (i) Diversity: There may be multiple variants of operand-lists for a LOST sequence. For example, \(I_1: \text{add } r1, r2, r3\) and \(I_2: \text{add } r3, r4, r1\), respectively. We count both \(I_1 \rightarrow \text{st}\) and \(I_2 \rightarrow \text{st}\) as occurrences of \(\text{ld} \rightarrow \text{add} \rightarrow \text{st}\), irrespective of their differences in operand-lists namely \(\langle r1, r2, r3 \rangle\) and \(\langle r3, r4, r1 \rangle\). (ii) Efficient encoding: Given that there could be many such operand-lists for the same LOST hardware, we need an efficient way of supplying the operand-list to a LOST hardware for each of its invocation.

To understand operand-list diversity, we profile the operand-lists across multiple apps and present our insights in handling their diversity using an example app (MusicPlayer). We show the top 100 operand-lists sorted by their respective LOST coverage for all the generic LOST sequences in Fig. 4b. As can be seen, there could be potentially up to 5k different varieties of operand-lists for each of the LOST sequences and thus, we need an efficient mechanism of encoding/supplying the operand-list for the entire LOST sequence. A naive way of doing this is by using the original list of instructions themselves. Although this method supplies all the required operand-list to the hardware, it is highly inefficient because it needs fetch and decode for each of such instructions.

Alternatively, we construct a simple hardware lookup table and populate it with the most commonly used operand-lists and use the table entries to supply inputs to LOST invocations (as shown in Fig. 4c steps 1 and 2). Consequently, the compiler can replace the spatially contiguous set of LOST instructions with one simple ARM co-processor invocation instruction namely, CDP2 \(25\) with an index to this lookup table that holds the operand-list for that corresponding invocation. Note that, the compiler will not synthesize the CDP2 call for the operand-lists other than the entries in the lookup table. We will quantify this lost opportunity in Sec. 5A.

In Step 3, Fig. 4c shows that the compiler can generate the CDP2 instruction by replacing an example sequence of our generic LOST DataTransfer(Kernel)6 sequence, which is subsequently used in looking up the hardware operand-list table for supplying the appropriate operand-list for LOST acceleration. Using this approach, we can make CDP2 call to invoke LOSTs of any length in one instruction, with a caveat that the operand-list table will increase by 10 bits per entry for encoding one instruction. We observe that 64 entries to be a sufficient operand-list table size for all the apps to capitalize on their respective LOST coverages and accelerate them.

E. Proposed Hardware Integration

**LOST Execution Support:** Fig. 4c shows the micro-architecture enhancements to the baseline Out-of-Order (OoO) CPU design to support LOST execution. They are: a) **Operand-List Table** with 64 entries (400 bytes per LOST); b) **Execution Units** attached to the Common Data Bus. Whenever there is a LOST invocation (e.g. CDP2 LOST.id, LOST.operand-list-index), the instruction decoder will use the LOST.id (DATA_TX_k6) and LOST.operand-list-index (54) fields (as a normal ARM instruction) to lookup the corresponding operand-list entry in the **LOSS Operand-list Table**. The operand-lists will supply operands to the ROB in the same way other instructions are inserted. The only difference is that these accelerator instructions have \(m\) input and \(n\) output registers, as opposed to 2 input registers and 1 output register in most of the conventional instructions (even in the current ARM ISA there are instructions such as STM and LDM which take multiple inputs/outputs). When all the input registers in the operand-list are ready (i.e., InBound instructions are...
ready to commit), the LOST instruction will be issued to the corresponding LOST Execution Unit. After that, it will execute and commit/squash as a normal CPU instruction.

**Example of a LOST Sequence Execution** We now illustrate the difference between a CPU execution and a LOST hardware execution using the timing diagram shown in Fig. 5. The upper part shows a list of nine CPU instructions, where there is an InBound scenario between I3 (non-LOST) to I5 (LOST), and a similar OutBound scenario between I5 (LOST) to I7 (non-LOST). If executed in a normal OoO CPU, it finishes in 14 cycles. The bottom part shows the code transformed (by the compiler) into five instructions, including hoisting the InBound (I3) above and moving the OutBound (I7) below the LOST instructions, and subsequently replacing LOST instruction by a single CDP2 instruction. In this setup, I1 and I3 are executed in the normal CPU pipeline. Next, the LOST invocation instruction (CDP2) is fetched as usual at 3rd cycle and waits for I3’s result. When the InBound (I3) is ready to commit, the CDP2 is issued at the 7th cycle. It subsequently finishes its execution and commits at the 9th cycle. I7 (OutBound) subsequently gets fetched at 4th cycle, and again waits for the LOST’s result to proceed and execute to commit at 12th cycle. Although the LOST acceleration helps the execution to finish 2 cycles ahead of the baseline CPU execution in this example, the performance impacts may vary depending on the extra waiting-time introduced between InBound/OutBound and LOSTs.

**V. EXPERIMENTAL EVALUATION**

**Mobile Apps Used** We start our LOST study using 10 popular Android apps and find 5 generic LOSTs in their execution. To validate their generality, (i) we use 5 more popular apps that were NOT used in the characterization effort towards generic LOSTs, and (ii) we test with two available android compilers viz., Dalvik and ART compilers with up to 22 optimization passes to rule out any compiler level artifacts. Table V details all 15 apps in terms of number of downloads, popularity (rated by users out of 5 ⭐️), and app domains. Also, to ensure that we capture the app behavior, we perform general tasks described for an app for a certain period of time, and its corresponding instruction footprint in Table V More details can be found in [http://csl.cse.psu.edu/lost](http://csl.cse.psu.edu/lost).

**Traces and Simulation Platform:** For each app, we supply the LOST and CPU traces to the Gem5 simulator (version 5.4 [18]) to model the CPU behavior for LOST execution and normal OoO CPU execution. Since Gem5 only models conventional ARM CPU, we use Vivado HLS and Synthesis tools [36] to model LOST units’ timing and power. We use HLS synthesis of LOST units as a conservative approach to model the timing of LOST units (also stated by [2]). We also simulate memory using DRAMsim2 [37] that comes bundled with Gem5. The details of evaluation are listed in Table IV.

**Schemes Evaluated:** For understanding the impact of LOST acceleration mechanism in optimizing the fetch, decode and execution stages of the CPU, we compare the CPU performance improvements through LOST acceleration with that of a baseline CPU execution and an ideal scheme, where the fetch, decode and execution stall times are set to zero. The ideal scheme also does not incur any of the overhead such as InBound/OutBound, bulges and branches.

We present our evaluation of the proposed LOST acceleration hardware by comparing it against a) baseline CPU execution and b) 0-stall (ideal) execution. All the results are normalized with respect to the baseline OoO execution.

**A. Performance benefits from LOST**

Fig. 6 plots the CPU speedup for all 15 apps. The results show that all apps benefit from LOST sequence acceleration and the benefits vary between 12% (Maps) and 44% (as seen in PlayStore). The results also show the individual contributions of the 5 LOST sequences across different app domains. For example, user-intensive apps such as Angrybirds and Facebook Messenger make good use of the EventListener1 acceleration (up to 9% speedup), while passive apps like Clock and Calculator achieve considerable performance gains from Iterator (10.3%) and Net1 (19%). With the ideal (0-stall) case, we see an average CPU speedup of 30% w.r.t the baseline OoO CPU execution with a maximum speedup of 52%. Our LOST acceleration, on the other hand, provides an average 25% speedup, thus lagging behind the 0-stall setup by only 5%. We can see that CPU intensive apps such as Acrobat, Browser, Email and Office perform within 3% of the ideal case. But, the gap widens when the coverage becomes higher in cases such as calculator when the latency to fetch/decode becomes dominant before invoking LOST (discussed later in Fig. 7a).

**B. Trade-offs with LOST acceleration**

**Where does it gain?** We next analyze the reasoning behind the LOST acceleration gains by breaking down the performance benefits across various CPU execution stages. Fig. 7a shows...
the LOST execution speedup breakdown across the Fetch, Decode, execution of LOST sequence, execution of CPU instructions, and OutBound scenarios. We do not show stalls due to InBounds because of its negligible contribution to the speedup. We gain performance speedup from all 4 stages, especially, from the CPU speedup(8%), because we offload 5 LOST sequence to LOST hardware (as high as 54% of the dynamic instructions). The performance of LOST sequence execution also boosts because we accelerate these LOST instructions. We observe 8% gain from fetch and decode stages due to reducing the LOST sequence into a single instruction.

Where does it lose? The LOST acceleration has its limitations due to the following aspects: InBound stalls, OutBound stalls, Branches that are not favorable (infrequent), Bulges and the coverage limitations of a fixed 64 entry operand-list table. Fig. 7(a) shows the average occurrence of all these aspects on LOST acceleration. We see that, InBound stalls occur ≈1% of the total LOST coverage and hence, is not a major factor for performance loss. OutBound stalls on the other hand are more frequent (78%) and their effect on the app performance is still hidden by the gains from other execution stages (as explained above). The other contributing factors such as bulges, infrequent branch paths and operand-list based coverage limitation are not overly dominant and thus, can be addressed by more complex solutions to accelerate another 20% of LOST execution in future extensions to this work.

In summary, we systematically quantify the LOST acceleration benefits and in turn present an analysis of where and when LOST acceleration can gain the most as well places where it cannot gain much. The average 25% CPU performance acceleration translate to 12% system-wide performance improvement based on CPU utilization reported in Fig. 1. To complete our LOST acceleration hardware study, we also do a conservative energy estimation from our HLS models for the generic LOST sequences (average of 5.9nJ per invocation) and a CACTI [38] based energy estimation (0.0049nJ per read) for the Operand-list tables. Our estimations show that the LOST acceleration hardware can conservatively save up to 20% in terms of CPU energy, translating to 7% system energy savings.

VII. CONCLUSIONS

We present a fine-grain hardware acceleration mechanism for alleviating CPU involvement, thereby improving app performance in handhelds. In this context, we have developed a comprehensive framework to effectively identify and extract all LOST sequences of instructions from an app and pick the commonly occurring sequences within and across apps. Unlike prior approaches that identify instruction sequences

However, the basic concepts of hardware customization has been proposed much before that [9], [16], [41], [42]. Unlike our work, most of these designs are at a coarse-grain granularity targeted for a specific function.

A body of work like BERET [10], Chimaera [9], OneChip [16], and others [43], [44] integrates a reconfigurable execution unit into the CPU pipeline. These accelerate instruction sequences in the ranges of thousands of instructions. Works like programmable functional units (PFU) [11], specialization-engines for data-flows [12], work at the granularity of tens of instructions. During runtime, they track the instructions executed and dynamically reconfigure the execution unit. This unit is triggered with a customized/extended instruction set, where a commonly used sequence of instructions are replaced by one instruction. There are other works, which optimize the ISA for specific apps and domains, e.g., MMX, NEON, SSE, RISC-V [13] etc. This approach (adopted by [45], [46]) uses a SIMD architecture for accelerating regular code such as a loop-based computations. Thus, they are limited in scope for general purpose apps. Similarly, chained vector units (e.g. [47], [48]) exploit the strategy of optimizing instruction sequences based on data flow. They are beneficial for code, where dependencies are not just statically identifiable, but are also spatially/temporally proximate. However, as we observe, our LOST sequences have much larger spatial and temporal spans, than those exploited by chained vector units.

Compiler optimizations [4], [49] in the mobile space focus on reducing the amount of compute to achieve the same result or provide hints to the run-time for better memory/compute scheduling [50], [51]. Dalvik, a VM framework in Android OS, is one such framework which not only interprets the source code at runtime, but also identifies and compiles the hot-spot trace [49]. Our proposal can complement these proposals in finding more opportunities for hardware acceleration.

The most related to ours is DySER [7], which uses dynamically synthesized data-paths based on opportunities identified by the compiler. These paths are configured statically into CGRA cores at runtime. However, the offloaded functionality is restricted (i.e. cannot perform memory operations), thus requiring all memory data to be explicitly passed (no shared registers either) through FIFOs. While this may work in apps from SPEC, PARSEC, etc., we find mobile apps have many more data-dependent interspersed loads, making it difficult to pre-load all the required data needed for a reasonable amount of off-loaded computation. As shown, many of the frequently occurring LOST sequences (Table III) do contain such data dependent loads. To our knowledge, this is the first effort to analyze a wide variety of mobile apps and propose a fine-grain accelerator design that is more universally beneficial.

VI. RELATED WORK

Recently, different flavors of hardware acceleration have been explored for performance and energy optimizations in several app domains such as machine learning [6], [8], computer vision [5], speech recognition [39] and health care [40].

Unlike prior approaches that identify instruction sequences
based on PC addresses or operands, the LOST sequences are defined based on a sequence of opcodes that can even span method boundaries, effectively increasing coverage and the potential for finding opportunities for hardware acceleration. We also discuss solutions for handling critical dependence issues in hardware acceleration such as control flow divergence and operand supply to a LOST hardware unit and design a micro-architecture that integrates these customized hardware sequences as macro functional units into the CPU datapath. Experimental evaluations with fifteen mobile apps indicate that the proposed techniques can provide on an average 25% CPU speedup (12% overall system performance improvement). These improvements are significant considering the broad coverage across apps from different domains, as well as requiring just five very small LOST sequences.

REFERENCES


