Abstract—High-level-synthesis (HLS) tools generate accelerators from software programs to ease the task of building hardware. Unfortunately, current HLS tools have limited support for concurrency, which impacts the speedup achievable with the generated accelerator. Current approaches only target fixed static patterns (e.g., pipeline, data-parallel kernels). This constraints the ability of software programmers to express concurrency. Moreover, the generated accelerator loses a key benefit of parallel hardware, dynamic asynchrony, and the potential to hide long latency and cache misses.

We have developed TAPAS, an HLS toolchain for generating parallel accelerators from programs with dynamic parallelism. TAPAS is built on top of Tapir [22], [39], which embeds fork-join parallelism into the compiler’s intermediate-representation. TAPAS leverages the compiler IR to identify parallelism and synthesizes the hardware logic. TAPAS provides first-class architecture support for spawning, coordinating and synchronizing tasks during accelerator execution. We demonstrate TAPAS can generate accelerators for concurrent programs with heterogeneous, nested and recursive parallelism. Our evaluation on Intel-Altera DE1-SoC and Arria-10 boards demonstrates that TAPAS generated accelerators achieve 20× the power efficiency of an Intel Xeon, while maintaining comparable performance. We also show that TAPAS enables lightweight tasks that can be spawned in ~10 cycles and enables accelerators to exploit available fine-grain parallelism. TAPAS is a complete HLS toolchain for synthesizing parallel programs to accelerators and is open-sourced.

Index Terms—High-level Synthesis, LLVM, Chisel, HLS, Cilk, TAPAS, Hardware accelerator, Power efficiency, Dynamic parallelism, FPGA

I. Introduction

Industry and academia realize that hardware customization is required to continue performance scaling as semiconductor scaling tapers off. Amazon EC2 [23] and Huawei have made FPGAs available to the public through the cloud. Microsoft [35] is also exploiting FPGAs for accelerating datacenter services. To address the challenges of developing application or domain specific hardware, high-level-synthesis (HLS) tools have been introduced. HLS translates a program in high-level language (e.g., C, C++) to an RTL circuit specification. It is an open question whether HLS tools have enough flexibility to permit software engineers to design high performance hardware.

A key limitation of HLS tools is their approach to concurrency. Accelerators attain high performance by instantiating multiple execution units that effectively support both coarse-grain and fine-grain concurrency [1], [5], [34] (relative to software). Unfortunately, current HLS tools do not effectively support concurrent languages. HLS tools also require an extensive set of annotations to generate parallel architectures. concurrency [10]. High-level-synthesis (HLS) tools with C interface typically analyze loops and employ techniques such as unrolling and pipelining [1]. Both Intel and Xilinx have targeted HLS at data parallelism [26].

HLS tools were aware of the challenges introduced by concurrency and have sought to exploit higher-level parallelism. LegUp [2], [11] includes support for a subset of the OpenMP and pthread APIs, and seeks to benefit from thread-level parallelism. IBM’s liquid metal [5] supported streaming kernel parallelism. Both toolchains are limited to static concurrency patterns, i.e. the parallelism structures are known during hardware generation and the structures cannot change during execution.

Recent works and industry-standard HLS tools have adopted fixed hardware templates that target specific concurrency patterns. Common templates include data parallelism, loop parallelism and loop pipelining [29], [33], [42]. The application programmer is expected to annotate and modify the application to fit the template. Template-based HLS adopts a construct-and-run approach in which the concurrency and operations are scheduled statically at hardware generation time. Unfortunately, in many concurrent programs the parallelism evolves as the program runs, either due to control flow [30], or run time non-determinism [14], [15] (see example in Figure 1).

Current HLS tools are built on a sequential compiler i.e., compiler intermediate representation and passes restricted to a sequential program-dependence-graph. Hence, prior tools largely focused on programs with static parallelism that can be expressed through templates (e.g., pragma pipeline) or library calls (e.g., OpenCL). Our work focuses on programs with irregular fine-grain parallelism expressed implicitly within the program and it has been built on a parallel compiler released in 2017 [39]. We demonstrate that for programs with dynamic concurrency, FPGAs can achieve higher performance/watt than a multicore.
Our Approach

Our work focuses on synthesis of hardware accelerators from parallel programs that contain on-the-fly or dynamic parallelism. TAPAS is a complete HLS framework that leverages parallel IR \cite{41} to generate the RTL for a parallel task-based accelerator. The accelerator architecture includes support for spawning and synchronizing both homogeneous and heterogeneous tasks at run time. TAPAS leverages the parallelism markers embedded by Tapir to generate RTL in two stages. The first-stage analyzes the parallel IR to infer the task dependencies, required synchronization, and generates a top-level architecture at the granularity of tasks. In the second stage, it generates the dataflow execution logic for each task; we permit arbitrary control flow (including loops) and memory operations. The microarchitecture generated by TAPAS is specified in parameterized Chisel \cite{4} and permits the designer to vary the number of tiles dedicated-per task, resource per task (e.g., queue depth, registers, scratchpad) and memory system capacity.

We illustrate that the dynamic task-based accelerator has flexibility for realizing nested, heterogeneous, recursive, irregular or regular concurrency patterns. We briefly discuss how TAPAS handles the challenges of generating hardware for a dynamically pipelined program, Dedup from PARSEC (see Figure \ref{fig:1}); the figure includes the commented pseudo code. HLS tools find this particular code sample challenging and cannot generate an optimal microarchitecture. First, the stages in the pipeline change based on the inputs. As shown in the task graph, for some iterations stage-2 could be entirely skipped based on the results from stage-1. Second, the stages have different ordering constraints and exhibit nested parallelism. Stage-2 is embarrassingly parallel while stage-1 enforces ordering across each sequence. Finally, the pipeline termination condition (see line 4) needs to be evaluated at runtime and cannot be statically determined (e.g., bounded loop).

To handle dynamic parallel patterns TAPAS generates a hierarchical microarchitecture that includes first-class support for generic tasks. At the top-level the accelerator’s microarchitecture consists of a collection of unique atomic task units (one for each heterogeneous task in the system). Each task unit internally manages the dataflow logic for executing the task. The generated architecture has the following benefits: i) dynamic task spawning enables the program control to skip stages entirely and change the pipeline communication pattern, ii) the hierarchical task logic organization permits concurrent tasks to be nested. TAPAS permits Dedup’s stage-2 to be internally parallelized while ordering the tasks for stage-1. iii) The architecture eliminates dedicated communication ports, and allocates local RAM for communicating data between the tasks. This permits Dedup’s stage-1 to directly pass data to stage-3 when stage-2 is bypassed conditionally. iv) Finally, the architecture does not require any separate control for managing the task dependencies. TAPAS derives the concurrency control from the compiler IR and embeds it within the tasks e.g., pipeline exit function is the next_chunk() dataflow embedded within stage-0.

1) We have developed TAPAS, a complete open-source HLS tool that generates parallel hardware accelerators with support for dynamic task parallelism.
2) TAPAS’s framework is based on a parallel compiler intermediate-representation and includes support for arbitrarily nested parallelism and irregular task parallelism. It is language agnostic and has been tested using Cilk, Cilk-P and OpenMP.
3) We have developed a library of hardware components for spawning and synchronizing tasks, buffering tasks, and inter-task communication. We demonstrate that TAPAS HLS can compose these components to generate high performance parallel accelerators.

![Generated Pipeline Accelerator](image-url)
4) We evaluate the performance and flexibility of TAPAS on the Intel-Altera DE1-SoC and Arria 10 FPGA boards. TAPAS achieves \(\approx 20\times\) better performance/watt than an Intel Xeon quad core, while the performance is comparable to the multicore processor.

II. Background and Scope

There is a gap in the quality of the hardware generated by HLS and human-designs a) partly due to the inability of the HLS tool to comprehend and exploit the parallelism available in software, and b) partly due to underlying abstractions being not available in the hardware architecture. As resource abundant FPGAs appear in the market it becomes imperative for HLS to support dynamic parallelism where the user only specifies what tasks can run in parallel, instead of how the parallel tasks are mapped to execution units.

A. Why dynamic task parallelism in TAPAS?

A question that naturally arises when generating a parallel architecture is how does one specify parallelism to an HLS tool? There appears to be no consensus among current toolchains. This is primarily a result of there not being a standard framework to support concurrent execution, as is true for CPUs. Common frameworks such as OpenMP [2], [7] and Intel TBB are implemented using threads. However, it is unclear if the requirements of threads (e.g., precise register context, shared memory, per-thread stack) can be supported on non-CPU architectures at low overhead. Recent works have included support for threads in OpenMP loops [3], [11], [43].

We present an alternate vision based on the task abstraction. Please note that the notion of dynamic tasks [3], [22] we discuss here is different from the notion of static tasks explored in prior work [3]. Intuitively, a task is analogous to some encapsulated computation in software (not unlike a function call) which takes arguments and produces a value after running to completion. Every task is described as a three tuple \((f(), \text{args}, \text{sync})\). The function \(f()\) represents a scoped subset of the program dependence graph which implements the functionality. \(\text{Args}[\text{]}\) is a set of arguments passed to the function and \(\text{sync}\) is a run time field that represents other tasks that need to be synchronized. The key feature of tasks is that tasks can dynamically spawn new child tasks at run time. There has been extensive work in supporting static task parallelism in FPGAs [9], [9], [10], [21]. Prior works statically scheduled these tasks on the underlying execution units and relied on the task abstraction for understanding the static concurrency pattern. TAPAS provides direct support in hardware for creating and synchronizing tasks dynamically based on accelerator execution. TAPAS adopts a dynamic software task model that has been previously explored in the context of [21], [27], [38], vector architectures [40] and GPUs [32].

B. Tapas vs Industry-standard HLS

Dynamic Parallelism vs Static Parallelism

The term dynamic parallelism refers to enabling tasks to vary at run-time, both in terms of the type of child task spawned and the number of child tasks spawned. Prior HLS tools adopt static parallelism in which any concurrent thread/task is created and scheduled up-front. We use a commonly used parallel for-loop (Figure 2) to illustrate the differences between dynamic and static parallelism. The loop exhibits dynamic parallelism. First, the loop bounds are determined by a parameter \(\text{len}\) which is known only during execution and which varies the number of parallel loop iterations. Second, the \(\text{bar()}\) function is invoked only if \(\text{node}[i]\) is valid. The figure shows how TAPAS handles this pattern. TAPAS creates a root task for the loop control, which spawns a child task, \(f()\), only when required (i.e., \(\text{node}[i]\) is valid) and up to the dynamic maximum of \(\text{len}\). Current HLS tools will unroll the loop to exploit the static parallelism. When a loop is unrolled, the HLS tools create multiple hardware execution units onto which successive loop iterations are statically scheduled at the hardware construction time (in Figure 2 unroll factor is 2). Therefore, they must plan for the worst case and allocate resources for all possible iterations regardless of whether they are actually executed, and must handle corner cases (e.g. \(\text{len} \neq \text{unroll}\)).

Static vs Dynamic Scheduling

Another limitation of current industry-standard HLS tools [16] is the lack of dynamic scheduling i.e., the ability for the dataflow to handle variations in instruction latency (e.g., cache misses). Even the HLS tools that support threads [12], only support static scheduling of instructions. Since memory instructions also need to have deterministic scheduling, prior HLS tools primarily support a streaming memory model in which data is loaded into a scratchpad ahead of invocation. While the combination of static scheduling, static concurrency, and streaming memory model leads to high efficiency, but it limits the type of
workloads that HLS can target. A pre-requisite for supporting dynamic task parallelism is shared memory and caches. Consequently, TAPAS needs to handle non-deterministic latency in memory operations (see Section III-C).

Concurrent work from Josipović et al. at FPGA 2018 has started investigating the benefits of dynamic scheduling of instructions. However, their work only exploits static parallelism from loops in sequential programs. TAPAS’s focus is on dynamic parallelism and parallel programs. TAPAS includes support for the task abstraction in the compiler that makes it feasible to target parallel languages such as Cilk. TAPAS also supports dynamic scheduling, however this is not our focus.

C. **TAPAS vs Prior Work**

<table>
<thead>
<tr>
<th>Base HLS</th>
<th>HLS+Kernels</th>
<th>Pattern</th>
<th>Parallel Prog.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Seq. Loops</td>
<td>Thread</td>
<td>Pattern</td>
</tr>
<tr>
<td>Target</td>
<td>Kernel</td>
<td>Pattern</td>
<td>Parallelize</td>
</tr>
<tr>
<td>Hints</td>
<td>#pragma Kernels</td>
<td>Pattern</td>
<td>Spawns/Sync</td>
</tr>
</tbody>
</table>

Table I summarizes the feature set of current HLS tools. Many HLS tools primarily target sequential programs and unroll loops to exploit instruction parallelism. A parallel architecture is often realized by using the HLS compiler to synthesize a single hardware core, and then typically requires an expert to manually instantiate multiple instances of the core in a hardware description language. To avoid this, both Xilinx Vivado and Intel HLS unroll and pipeline loops to convert loop parallelism to instruction level parallelism. Achieving efficient hardware requires the software developer to identify where it might be feasible to exploit loop parallelism and add additional hardware-oriented pragmas. HLS tools have anticipated the need to target higher levels of parallelism. Recent works have supported a subset of OpenCL, OpenMP or Pthreads. The primarily target is data parallel kernels. Current HLS tools schedule the concurrent operations statically and do not support dynamic spawning, asynchronous behavior, or nested parallelism. Furthermore, since the HLS tools statically schedule the memory operations, they require code annotations to help identify the streaming and FIFO access patterns between functions [13]. Finally, a promising avenue of research is HLS for domain-specific patterns. The hardware expert designs a parameterized template that targets a parallelism pattern (e.g., pipeline) and the software developer modifies the applications to ensure the program structure matches the pattern. Unfortunately, patterns always risk becoming obsolete.

**TAPAS** targets parallel programs (not a particular pattern) and only requires the programmer to identify concurrent tasks. It is built on a parallel compiler and leverages the information to automatically synthesize parallel hardware for arbitrary task graphs. The key novelty of TAPAS is that tasks can dynamically spawn and sync with other tasks. This enables TAPAS to handle a variety of common programming patterns including nested, recursive and heterogeneous parallelism. Finally, TAPAS supports dynamic scheduling of operations and handles non-determinism to enable a cache-based memory model.

### III. TAPAS: High-Level-Synthesizing Dynamic Parallel Accelerators

**TAPAS** is a hierarchical HLS toolchain for generating the RTL for a parallel application-specific accelerator (see Figure 3). **TAPAS** is language agnostic since it relies on Tapir-LLVM to parse the parallel program and generate compiler IR with additional markers indicating the parallelism. The input to TAPAS is a parallel program with markers for tasks and parallel loops; currently our infrastructure has been tested using Cilk, OpenMP and Cilk-P. Figure 3 shows the stages in TAPAS and RTL generation for a program with nested parallel loops. TAPAS consists of three stages. In Stage-1, (Section III-A) TAPAS analyzes the compiler IR, extracts the task dependencies, and generates the top-level RTL. The task units are declared and wired to the memory system. In Stage-2 (Section III-C) the program graph of each task is analyzed, and the RTL is generated for the dataflow of each task unit. Finally, in Stage-3 we configure and set the hardware parameters (e.g., number of execution cores) based on a specific deployment (e.g., LUTs available on the FPGA) and generate FPGA bitstream.

TAPAS-generated accelerators support dynamic parallelism, dynamic scheduling, and caches. We restrict all communication between the ARM and the accelerator to occur through shared memory. Currently, TAPAS maps the accelerators to the FPGA on an SoC board. The ARM and the FPGA share a 512KB L2 cache. We synthesize a 16K L1 cache for the accelerator which is kept coherent with the L2 through AXI. TAPAS generates a binary for the program regions/functions that cannot be offloaded (e.g., due to system calls) and they run on the ARM. TAPAS does not rely on any hard logic in the FPGA and synthesizes the logic required to support the parallelism. This enables a flexible execution model that is independent of the processor and enables TAPAS to target different FPGA boards.
A. Stage 1: Task Parallel Architecture

TAPAS relies on Tapir \cite{39} to comprehend the semantics required by the task-based accelerator architecture. Tapir adds three instructions to LLVM IR, detach, reattach and sync, to express fork-join parallel programs. Using these three instructions TAPIR can support dynamic task spawning (create a concurrent task) and sync (synchronize parent and child). We describe the front-end task compiler pass in more detail in \textbf{Stage 1} and focus on the hardware generation itself once the task dependencies are known. The generated accelerator consists of multiple task units at the top-level, and each task unit represents a unique task. Figure \ref{fig:tapas} illustrates the top-level RTL, the interface and the parameters associated with the interface. TAPAS supports time multiplexing (equivalent to simultaneous multithreading) of multiple tasks on an execution unit, dynamic tiling and assignment of tasks at runtime to different execution units (equivalent to multicore). A task unit is an execution engine for a single task and serves as the basic building block in the architecture. The accelerator can consist of any number of task units interacting to create different task graphs. There are four main components within each atomic task unit: i) The task queue which manages spawned tasks, ii) Parent task interface (Spawn/Synchronization ports), iii) Child task interface (Spawn/Synchronization ports), and iv) Task Execution Unit (TXU) which represents a pipelined dataflow execution unit.

B. Execution Example

We describe the functionality of each of the components in the task unit by considering the implementation of the nested loop example (see Figure \ref{fig:tapas}). The task graph in the figure illustrates three tasks, T0 and T1. T0 is the outer loop control and spawner of N instances of inner loop. T1 is the inner loop control and spawner of N instances of T2, the body. Finally T2 performs the actual work, reading elements from the A[i][j], B[i][j] and adding them. In this application, N dynamic instances of task T1 will be created (for each iteration of the outer loop) and each dynamic instance of T1 will create N instances of T2 (total: $N^2$ instances).

A task in the queue can be in of the following states

- READY: spawned, but not allocated a TXU
- EXEC: TXU allocated, but task has not complete
- COMPLETE: execution complete and need to synchronize with the parent
- SYNC: Waiting on synchronization with child tasks. The task queue metadata consists of a child join counter (Child#), the ParentID and Args[] RAM (argument RAM).

Fig. 3: Overview of TAPAS. Top: Compilation flow. Bottom: Generated output at each stage.

Fig. 4: TAPAS generated microarchitecture in Chisel \cite{4], the A[i][j], B[i][j] and adding them. In this application, N dynamic instances of task T1 will be created (for each iteration of the outer loop) and each dynamic instance of T1 will create N instances of T2 (total: $N^2$ instances).

A task in the queue can be in of the following states

- READY: spawned, but not allocated a TXU
- EXEC: TXU allocated, but task has not complete
- COMPLETE: execution complete and need to synchronize with the parent
- SYNC: Waiting on synchronization with child tasks. The task queue metadata consists of a child join counter (Child#), the ParentID and Args[] RAM (argument RAM).
[SID, DyID]. The SID refers to the name of the parent task (in this instance T0) and the DyID corresponds to the task queue entry allocated to the instance of the parent task (index 0 here). This corresponds to dynamic task T0:0 spawning an instance of T1 (a j-loop, when \(i = 0\)). The ParentID metadata available in the spawn is noted down in the allocated to the spawned T1 task and will be used during synchronization. In (2) the dynamic instance T1:0 (corresponding to the inner j-loop with \(i = 0\)) creates N instances of the inner body T2. The field C# (Child#), in task T1:0’s queue entry corresponds to the count of the children tasks that are created by dynamic task T1:0. In this example, N instances of T2 are created corresponding to loop iterations \(i=0,j=0...N-1\). Note that, task T0 may concurrently create other instances T1:1,T1:2,... (inner j loop for \(i=1,i=2,...\text{iteration}\) if there is enough queuing available. The task unit asynchronously assigns task execution units for the ready tasks.

In (3) as the instances of T2 complete they synchronize with their parent task that spawned them. Each task will only synchronize and join with the parent task that created it. Here, the T2 instances T2:0...T2:N-1 (corresponding to tasks \(i=0,j=0...N-1\)) will join on completion with the dynamic instance of their parent T1:0 (\(i=0, j\)-loop control). Joining entails decrementing the counter in the queue entry (index 0) corresponding to T1:0. The purpose of noting down the SID and DyID when the T2 tasks were spawned is clear now. The SID permits composability and allows heterogeneous task units to communicate with and dynamically spawn a shared task. The SID serves as the network id of the parent task unit to route back on a join. The DyID serves as the index into the queue within the task unit corresponding to the SID. Finally in (4), once T1:0 has joined with all its spawned children, it proceeds to move from Sync to Complete status and reattaches back with its parent, T0. The task queue interfaces decouple task creation from task execution. The spawn and sync are asynchronous, and employ ready-valid signals. The asynchronous design permits us to vary the resource parameters per task without having to reschedule the tasks to deal with changes in latency.

C. Stage 2: Generating Task Exe Unit (TXU)
TXU is the representation of execution engine within each task unit. Each TXU is a fully pipeline execution unit which permits multiple dynamic instances of a task execute simultaneously. The TXUs only communicate at the task boundaries with each other. All the inter-TXU communication is marshaled through a shared scratchpad or the cache.

TAPAS generates the logic for the TXUs based on the per-task sub-program-dependence-graph earmarked by our compiler. Each TXU is a dataflow that enables fine-grain instruction level parallelism to be mined. TAPAS HLS dynamically schedules the operations in the TXU. An automatic pipelining process introduces latency insensitive ready-valid interfaces between each operation in the dataflow. A dataflow graph mapped to the TXU may contain nodes with multi-cycle latency (e.g. floating point operations), and non-deterministic latency (e.g., memory operations). This approach is in contrast to current industry strength HLS tools which try to schedule the timing all operations statically; concurrent work in FPGAs has begun to analyze the potential of dynamic scheduling [35].

Figure 7 shows the add function unit from the file created by TAPAS, from $C[i][j] = A[i][j] + B[i][j]$. The add function unit communicates with Load $A[i][j]$, Load $B[i][j]$, + and Store $C[i][j]$ via decoupled handshaking signals which contain ready and valid signals in addition to data. The handshaking interface is governed by a simple state machine. This dataflow permits multiple concurrent T2 tasks to be outstanding at the same time on the execution unit. Task pipelining is illustrated in Figure 7. The dynamic task ids correspond to the queue index allocated at run time. Note that the pipeline of a TXU is in dataflow order and tasks complete in order of issue. Any load stalls cause the pipelined dataflow to throttle and eventually stall; however this leads to a simpler implementation compared to dynamic dataflow [20].

D. Stage 3: Parameterized Accelerator

TAPAS is a parameterized hardware generator and seeks to permit late stage parameter binding. As hardware designs grow in complexity, modularity becomes necessary. The asynchrony and latency insensitivity permits each of the task units to be parameterized independently. As shown in Figure 4 every task unit provides the mechanism for passing parameters prior to hardware elaboration and bitstream generation. While each tile has multiple parameters that can be set including the width and types of the args RAM, there are primarily two parameters that are set at this stage in toolchain, the task queue size ($N_{tasks}$) and the number of task execution units ($N_{tiles}$). We permit the user to vary the parameters on a per-task basis. The latency of the individual tasks and task dependencies will determine $N_{tasks}$. Determining $N_{tiles}$ is more involved as it depends on the the processing rate required of that particular task unit and how many active tasks are required to potentially hide memory latency.

E. Task Memory interface and Memory Model

In this study, we consider a heterogeneous SoC where both processing cores and accelerator are integrated into a single chip. Each of the accelerator’s caches is connected to the last-level cache, which is shared with the ARM processor over the AXI bus. A key question is how does the memory model look like on the accelerator side. TAPAS permits arbitrary task graph patterns to be converted into accelerators and thus needs to support a more flexible cache-like interface.

Fig. 7: Multiple tasks simultaneously outstanding on TXU.

Fig. 8: Data Box. Interfaces with memory operations in logic box and transfers operations to/from a cache or scratchpad.

In TAPAS all the task units share an L1 cache; it is conceivable that this model is most suitable for handling a programming model most familiar to software programmers [36], [37]. Generating an optimal cache hierarchy is beyond
the scope of this paper and we primarily focus on how to route values from the cache to the TXUs.

The data box (see Figure 8) connects a memory operation in the TXU to a memory interface. We currently support both cache and scratchpad (we only evaluate the cache memory model in this paper). We choose to group the common logic for multiple memory operations (e.g., misalignment) into the data box to minimize resource requirements. Figure 8 shows the architecture of the data box. Each data box consists of the following parameterized microarchitecture components: i) an in-arbiter tree network that arbitrates amongst requests to the memory interface, ii) an out demux network that routes responses back to the memory operations in the TXU’s dataflow, and iii) a table of staging buffers that contain the actual logic for reading the required bytes from the cache/AXI memory interface (which only supports word granularity accesses). Both the request and response networks are statically routed.

F. Compiler Front-end: Tasks from IR

TAPAS is language agnostic and relies on the parallel IR introduced by Tapir [39]. Tapir provides the front-end language bindings that translates Cilk/OpenMP programs to the LLVM IR. Tapir adds three instructions to the LLVM IR, detach (or spawn), reattach and sync. Spawn and reattach together delineate a task. A detach instruction terminates the block that contains it, spawns a new task starting from the target block and continues execution in parallel from the continuation. The reattach terminates the task spawned by a preceding detach instruction. Since Tapir assumes a generic threadpool execution model, it leaves the markers in place in the original PDG (Program Dependency Graph). We leverage the markers to perform reachability analysis and extract an explicit task graph, which is the architecture blueprint for our parallel accelerator. Nesting loops and irregular flows are analyzed in this stage and in the resulting task graph all task relations and the basic blocks that constitute a task are explicitly specified. We perform live variable analysis to extract and create the requisite arguments that need to be passed between tasks; these are used to parameterize the spawn port and args RAM for each task unit.

IV. TAPAS-generated Accelerators

There are no standard benchmark suites that target only dynamic parallelism. Table II gives a brief summary of the accelerator benchmarks and shows the characteristics of each application. Our emphasis is on being able to implement accelerators for these workloads without requiring additional effort from the programmer. Here we study applications that use common software patterns that current HLS tools either find challenging and may throw errors.

A. Nested Parallel and Conditional Loops

Related benchmarks: Matrix addition, Stencil, Image scaling, Saxpy Stencil

Multiple workloads employ the similar pattern of nested loops. However, there are variations based on the parallelism of the loop nests, loop depth and conditional loop entry/exit. Here, we briefly discuss Stencil. Stencil is an iterative kernel (Figure 10) that updates array elements in a loop. While the loop is embarrassingly parallel, the loop bounds are variable which introduces dynamic parallelism.

<table>
<thead>
<tr>
<th>Name</th>
<th>HLS Challenge</th>
<th>Memory Pattern</th>
<th>Per-Task # Inst</th>
<th># Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Add</td>
<td>Nested loops</td>
<td>Regular</td>
<td>49</td>
<td>21</td>
</tr>
<tr>
<td>Saxpy</td>
<td>Dynamic exit loops</td>
<td>Regular</td>
<td>29</td>
<td>16</td>
</tr>
<tr>
<td>Stencil</td>
<td>Nested parallel/serial</td>
<td>Regular</td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>Dedup</td>
<td>Task Pipeline</td>
<td>Irregular</td>
<td>180</td>
<td>72</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Recursive parallel</td>
<td>Regular</td>
<td>36</td>
<td>52</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>Recursive parallel</td>
<td>Regular</td>
<td>26</td>
<td>19</td>
</tr>
</tbody>
</table>

B. Pipeline Parallelism

Dedup code and hardware accelerator are outlined in Figure 11. Dedup has an irregular pipeline pattern. The main challenges posed by Dedup are:

- Task-Level Pipeline: HLS tools have limited support for task-level pipelines and primarily target loop
Mergesort Dynamic Tasks

Root task

spawn
sync
Mergesort

inner loop 1 (serial)
for(nr=0;...)

inner loop 2 (serial)
for(nc=0;...)

void stencil () {  
  /* Parallel for loop */  
  cilk_for (pos = 0; pos < NROWS * NCOLS; pos ← ++ +) {  
    /* Serial for loop */  
    for (nr = 0; nr <= 2*NBRROWS; nr ++ ) {  
      /* Serial for loop */  
      for (nc = 0; nc <= 2*NBCOLS; nc ++ ) {  
        int row = (pos/NCOLS) + nr - NBRROWS;  
        int col = (pos & (NCOLS-1)) + nc ← − NBRROWS;  
        if ((row < NROWS)) {  
          if ((col < NCOLS)) {  
          ...  
          cilk_sync;
          /* Parent waits for children */
          cilk_spawn mergeSort(list, mid + 1, end);
          /* Spawn to sort 2nd half */
          cilk_spawn mergeSort(list, start, mid);
          /* Spawn self to sort 1st half */
          cilk_spawn self to sort 1st half */
          /* Parent waits for children */
          if (start < end) {  
            int mid = start + ((end - start) / 2);
          }  
          merge(list, start, mid, end)  
        }
      }  
    }  
  }
}

Fig. 10: Stencil Accelerator

dedup is parallelized using tasks that have have non-trivial entry and exit logic, making it challenging to convert them to loops.

- **Conditional stages:** Emerging research [11] has sought to support functional pipelines using FIFO queues that require the program to be rewritten. Unfortunately, dedup also has conditional pipeline stages (see S2 in Figure 1) which FIFO queues cannot support. FIFO queues fix producer and consumer stages and cannot handle conditional pipelines.

- **Intra-stage parallelism:** The FIFO ports are ordered and this would lead to dedup losing parallelism in the S2 stage, which permits out-of-order chunk processing (see task-dependencies in Figure 1).

- **Pipeline control:** Finally, the pipeline termination condition is dynamically determined by an exit function (get_next_chunk). HLS tools do not support dynamic exits, since they statically schedule operations.

**TAPAS** does not suffer from these limitations since it supports dynamic spawning/syncing of tasks and execution units are assigned at runtime. Furthermore, all tasks communicate with each other through shared memory and the parallelism is not limited by extraneous hardware structures such as FIFO.

**C. Recursive Parallelism**

**TAPAS** can effectively generate accelerators for recursively parallel programs. HLS tools have traditionally not supported recursion [11], [19] due to the lack of a program stack. Nothing precludes the addition of a stack, but it would require changes to the HLS compilation framework. Figure 11 illustrates how **TAPAS** can support recursively parallel mergesort.

In mergesort, the primary function employs a divide and conquer strategy. It partitions an array into two halves, and recurses on each half in parallel. The parent function then waits on the children and merges the sorted halves. To implement recursion it must be possible for more than single invocation of the same function to exist at the run time. Further, the data also has to be implicitly passed via a stack. **TAPAS** achieves this through the following: i) **TAPAS** precisely captures the state needed by a recursive task from the LLVM IR and implicitly manages the stack frames in a scratchpad. ii) The task controller supports dynamic scheduling and asynchronous queuing, which permits a task to spawn itself without logic loops. iii) The task controller tracks the dynamic instances to support implicit parent-child synchronization iv) Finally, all return values from the recursion are passed through shared cache.

**V. Evaluation**

It is challenging to find a fair baseline since dynamic parallelism is not supported by existing HLS toolchains. Running our benchmarks on the FPGA would entail changing the algorithm and memory model, and a program re-write. Even finding a baseline CPU is challenging since the Cilk and Tapir are currently x86-only. The x86 multicore’s cache hierarchy is deeper and larger than the FPGAs, which makes it challenging to understand the impact of dynamic parallelism independent of the memory system. We answer the following questions: i) Can **TAPAS** support fine-grain tasks? How fine-grain can the tasks be? (§V-A) ii) Is the performance improvement attributable to low task spawn latency or speeding up individual tasks with dataflow execution? (§V-C) iii) What is the baseline performance compared to an Intel i7 quad core. (§V-C) iv) What is the energy consumption and performance/watt benefit compared to an Intel i7 (§V-D). In all the cases, we use the same unmodified Cilk programs. v) How does
static parallelism with prior HLS tools compare against
dynamic parallelism in TAPAS (%V-E)

A. Parallel Task Overhead

Result: The overhead of spawning a task on an FPGA
is significantly less than a software. This enables small, fine
gain tasks to scale better.

```
1 void scale(int *a, int n) {
2   int i;
3   cilk_for(i=0;i<n;++i) {
4       a[i]++;
5   }
6   return;
7 }
```

(a) Test Code

(b) Parallel Task Tiling

![Fig. 12: Scalability Test Code](image)

Fig. 12: Scalability Test Code

Fig. 13: Performance Scaling with Tiles

The microbenchmark in Figure 12(a) was synthesized
to see how fast tasks can be spawned. Figure 12(b) provides a top-view of the generated architecture. We
incrementally varied the amount of work ("+" operations)
in the loop body. The performance is plotted against an
increasing number of worker tiles (Figure 13). On a Arria
10 (∼300MHz) target device, we achieve a maximum spawn
rate of 40 million spawns/second. Even for fine-grain tasks
(50 instructions), the performance scales monotonically
with the addition of parallel worker tiles. ‘Software’ in the
plot (Figure 13) refers to spawning a task of 50 increments;
the program was run on a Intel i7-3.4Ghz, 8MB L2 (four
cores). At such fine-granularity, the software runtime for
Cilk provides zero benefit due to task spawning overheads.
TAPAS exploits the low overhead of task spawning on an
FPGA and enables fine-grain parallelism not exposed in
software.

B. Resource Utilization

Result: Arria 10 FPGA board can support ∼100 parallel
tasks each containing 100 integer operations.

![Fig. 14: ALM Utilization by Sub-block](image)

Fig. 14 shows the relative amount of ALM resources
(aka. LUTs and registers) used by each sub-block of the
design. In the extreme case (1 operation/task), 60% of
the logic is non-compute overhead; at 50 operations/task,
the overheads is ∼20%). As the number of execution tiles
increases, the overhead of the control logic is amortized
and at 10 tiles the control overhead is reduced to 3%. The
memory network required to support shared memory access
is less than 10% of overall chip resources. The network
is primarily needed to support dynamic scheduling and
routing values back and forth from the shared cache to the
internal nodes in the task execution unit.

C. Scalability and Performance

Result 1: TAPAS generated accelerators exploit all the
available parallelism exposed by the applications and scale
with increasing hardware resources (1.5–6×)

Result 2: To improve accelerator performance compared
to an Intel i7 multicore a better cache hierarchy is required.

Figure 15 plots the performance when varying the number
of execution tiles per task. A performance increase
from the baseline is seen in all examples with the exception
of Dedup. In Dedup even the baseline case (1 tile) has
four heterogeneous task units (Figure 3) organized as a
pipeline, with one execution tile per task unit. Any further
improvement with increasing tiles/task is feasible only if
the pipeline stages are unbalanced (not the case here).
The saxpy and matrix addition improve with the addition of a second tile, but the benchmarks quickly saturate the cache bandwidth as their inner loops are small and dominated by memory reads and writes. In contrast, the Stencil benchmark is more computationally intense and consequently scales well even up to 8 tiles and beyond.

Q2. In Figure [10] we compare the execution time of accelerators against an Intel i7 quad-core (3.4GHz,8MB L2,21GB/s DRAM). Identical Cilk benchmarks were used for both the i7 runs and TAPAS. We set the concurrency to be identical (four cores for i7 and four tiles for TAPAS). Accelerator designs were generated for both Cyclone V SoC FPGA and Arria 10 SoC FPGA. On the Cyclone V the accelerators performed at approximately 50% of the multicore (even achieving speedup in a few cases). On the Arria 10, the generated accelerators performed on par with the i7 as a result of higher frequency (300MHz vs 150MHz for the Cyclone V). The Dedup accelerator achieved best speedup since the accelerator implemented the pipeline more efficiently than software. The mergesort accelerator performed poorly in comparison to the Intel i7 since it is completely memory bound and limited by the memory system on the FPGA.

D. Energy Consumption

Result: TAPAS-generated accelerators exceed the energy efficiency of the multicore often by 20×.

In Table IV we report the absolute power consumption and resource utilization of the generated accelerators. The power is obtained using using Intel Quartus PowerPlay. It is an estimate of total power (static and dynamic) based on signal activity levels derived from gate-level simulation. The tabulated data shows how even with varied parallel patterns such as Stencil (nested loops) and Mergesort (recursive) we can effectively exploit the available resources (50% of the Cyclone V FPGA chip). Mergesort is the largest design using roughly half of the available chip resources and consuming approximately 1.5W of power. We compare the performance/watt (Figure 17) of the accelerator against a multicore; in both cases we set the concurrency level to four.

The power for the multicore is directly measured through the RAPL interfaces. TAPAS accelerators often achieve over 20× better performance/watt than the multicore.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Tile</th>
<th>MHz</th>
<th>ALMs</th>
<th>Regs</th>
<th>BRAM</th>
<th>Power(W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAXPY</td>
<td>5</td>
<td>149</td>
<td>7195</td>
<td>9414</td>
<td>3</td>
<td>0.957</td>
</tr>
<tr>
<td>Stencil</td>
<td>3</td>
<td>142</td>
<td>11927</td>
<td>11543</td>
<td>3</td>
<td>1.272</td>
</tr>
<tr>
<td>Matrix</td>
<td>3</td>
<td>223</td>
<td>4702</td>
<td>7025</td>
<td>3</td>
<td>0.677</td>
</tr>
<tr>
<td>Image</td>
<td>4</td>
<td>141</td>
<td>4442</td>
<td>5814</td>
<td>3</td>
<td>0.798</td>
</tr>
<tr>
<td>Dedup</td>
<td>3</td>
<td>153</td>
<td>10487</td>
<td>6509</td>
<td>3</td>
<td>1.014</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>4</td>
<td>120</td>
<td>5699</td>
<td>9887</td>
<td>62</td>
<td>1.155</td>
</tr>
<tr>
<td>Mergesort</td>
<td>4</td>
<td>134</td>
<td>14098</td>
<td>24775</td>
<td>74</td>
<td>1.491</td>
</tr>
</tbody>
</table>

E. Intel HLS vs TAPAS

An apples-to-apples comparison with prior HLS tools is challenging since: i) HLS tools only support static parallelism. It is not feasible to convert some applications to use static parallelism (e.g., recursive mergesort), and with others (e.g., Dedup) the conversion changes the algorithm entirely. ii) HLS tools deploy a streaming memory model since they statically schedule all instructions with known latencies. TAPAS employs caches and shared-memory, which are a pre-requisite for dynamic parallelism.

To attempt a quantitative comparison we use two benchmarks, SAXPY and Image scaling. Among our benchmarks these were amenable to static parallelism. We used the Intel HLS Compiler (v17.1) and employ the suggested streaming DRAM interface. We set up the DRAM latency for both the Intel HLS and TAPAS to 270ns (150MHz FPGA clock). We also set the same concurrency level. In HLS the loops was unrolled 3 times and TAPAS was configured to use 3 tiles. The results are listed in Table V. The results indicate that TAPAS is pretty competitive. It may be feasible to hand optimize the HLS implementation further, but we could also optimize TAPAS. The most notable difference is where the block RAMs are utilized. Intel HLS appears to generate large stream buffers in its load and store interfaces. In contrast, TAPAS uses a 16K L1 cache shared by all task units, but also expends block RAMs in the task queue.

The part3_ddr_masters.cpp example included with Intel HLS.
VI. Thoughts and Future Directions

Our results demonstrate that FPGAs and hardware accelerators have the potential to address a long outstanding challenge in concurrency, effective support for dynamic fine grain parallelism. The following facets need to be addressed to further improve performance.

- **Cache hierarchy:** To compete against a multicore processor we need to improve the overall cache hierarchy, both bandwidth and latency. The current cache macro-block we release as part of the toolchain is borrowed from the RISC-V cores with limited support for multiple outstanding cache misses. Our AXI implementation is also sub-optimal as we do not yet exploit all the burst options available in the protocol.

- **Task controllers:** The task controllers and queuing logic add latency to the critical path. In many workloads (e.g., bounded size matrix multiplication) there exist loop patterns that can be statically parallelized. TAPAS can benefit from statically scheduling such loops, and eliminating the task controllers. The challenge is identifying loops where this optimization may be feasible.

- **Opportunity for Dynamic Parallelism:** TAPAS relies on the compiler front-end (Tapir in this paper) to capture the parallelism intent of the workloads. TAPAS is currently capable of generating accelerators for the widely used fork-join parallelism. In our current workloads, apart from the initialization, all other functions are offloaded to the accelerator.

VII. Summary

TAPAS’s primary goal is to provide an intuitive HLS toolchain for software programmers to generate parallel accelerators. We have decoupled concurrency from parallelism; we use the task-based programming framework to convey what can run in parallel and generate an architecture that can dynamically explore the available parallelism at run time. We hope this will be an effective framework for those in the community to build and exchange parallel accelerators.

We have released TAPAS open source (github link redacted) which includes i) LLVM-based compiler back-end that translates parallel compiler IR to parallel accelerator architectures in Chisel, ii) a framework to convey concurrency and task parallelism to TAPAS, iii) Chisel libraries implementing support for task spawn/sync/reattach operations on an FPGA, iv) sample parallel accelerators (e.g., pipeline, nested loops, heterogeneous).

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd for suggestions and feedback that helped to improve this paper.

References

