Leveraging Dynamic Task Parallelism in Hardware Accelerators

by

Steve Margerm

B.A.Sc., University of British Columbia, 1994

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in the School of Computing Science Faculty of Applied Science

© Steve Margerm 2018
SIMON FRASER UNIVERSITY
Summer 2018

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
Approval

Name: Steve Margerm

Degree: Master of Science (Computing Science)

Title: Leveraging Dynamic Task Parallelism in Hardware Accelerators

Examining Committee: Chair: Parmit Chilana
                        Assistant Professor

                        Arrvindh Shriraman
                        Senior Supervisor
                        Associate Professor

                        William N. Sumner
                        Supervisor
                        Assistant Professor

                        Keval Vora
                        Internal Examiner
                        Assistant Professor

Date Defended: July 4, 2018
Abstract

Industry and academia realize that hardware customization is required to continue improving compute performance and energy efficiency as semiconductor scaling tapers off. Generating custom accelerator hardware is typically a time consuming process requiring specialized training and tools. However, High-level synthesis (HLS) tools have been actively developed over the last decade that aim to ease hardware development by generating custom logic from traditional software languages such as C/C++. Unfortunately, current HLS tools lack generalized support for concurrency which impacts the speedup achievable with the generated accelerator. Current approaches tend to target a few fixed static concurrency patterns such as pipelining and data-parallel kernels. This makes it challenging for a software programmer to express task level and dynamic concurrency. As a result, some of the key benefits of parallel hardware such as asynchrony and the potential to hide long latency events are lost.

We present TAPAS, an HLS toolchain for generating parallel hardware accelerators from programs with dynamic parallelism. TAPAS is built on top of Tapir [17, 34], which embeds fork-join parallelism into the compiler’s intermediate-representation. TAPAS leverages the compiler IR to identify the parallelism patterns and synthesize the hardware logic for managing it. TAPAS provides first-class architecture support for spawning, coordinating and synchronizing tasks during accelerator execution. We demonstrate that dynamic tasks enable TAPAS to generate flexible architectures for concurrent programs with heterogeneous and nested parallelism, without being restricted to a specific pattern. Our evaluation using an Intel FPGA SoC demonstrates that TAPAS can generate accelerators that scale effectively. The accelerators can tap into the available concurrency and deliver well over 20× the performance/watt when compared to typical desktop microprocessor. We find that TAPAS enables lightweight tasks that can be spawned in a single clock cycle which allows accelerators to exploit fine-grain parallelism.

Keywords: Hardware accelerator; Computer architecture; Reconfigurable computing; FPGA; High-level synthesis (HLS)
Dedication

To Wendy
Acknowledgements

I would like to thank my senior supervisor, Dr. Arrvindh Shriraman, for his extensive support and motivation during my studies, and Dr. William N. Sumner for his frequent advice and seemingly endless patience when approached with random questions and curiosities. I’d also like to thank Amirali Sharifian and Apala Guha, for their help and hard work while working on the TAPAS project and my other lab mates for their friendship and support.

I would like to thank my parents Barb and Doug, without whom I would not be here today. And last but not least, I’d especially like to thank my wife Wendy for her support and encouragement when I had this crazy idea of going back to school midway through my career.
# Table of Contents

Approval .......................... ii
Abstract ............................ iii
Dedication ........................... iv
Acknowledgements ....................... v
Table of Contents ..................... vi
List of Tables ......................... viii
List of Figures ........................ ix

1 Introduction ........................ 1
   1.1 Contribution ........................ 2
   1.2 Dissertation Outline .................. 4

2 Background .......................... 5
   2.1 Static Parallelism ...................... 5
   2.2 Dynamic Parallelism .................... 6
   2.3 Related Work .......................... 8

3 TAPAS: Task Parallel Accelerators ....... 11
   3.1 High Level Transformations: Stages 1 and 2 ....................... 11
   3.2 Generating a Task Parallel Architecture: Stage 3 and 4 ............. 12
      3.2.1 Task Units .......................... 13
      3.2.2 Execution Example ................. 14
      3.2.3 Generating Task Execution Units (TXU) ..................... 15
      3.2.4 Task Memory Interface and Memory Model .................... 18
      3.2.5 Recursive Tasks ..................... 19

4 Evaluation .......................... 21
   4.1 Task Granularity ...................... 21
   4.2 Resource Utilization ................... 23
   4.3 Performance .......................... 23
4.3.1 Benchmarks ......................................................... 23
4.3.2 Results ............................................................. 28

5 Conclusion ............................................................. 31
  5.1 Conclusions .......................................................... 31
  5.2 Current limitations .................................................. 31
  5.3 Future Work ........................................................ 32

Bibliography ............................................................. 33
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Comparing the features of HLS tools</td>
<td>9</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>FPGA Utilization</td>
<td>23</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Benchmark Properties</td>
<td>24</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Benchmark Results on Cyclone V</td>
<td>30</td>
</tr>
</tbody>
</table>
# List of Figures

| Figure 1.1 | Pseudocode for PARSEC’s Dedup based on CILK-P (modified to enhance clarity) | 2 |
| Figure 1.2 | Parallel accelerator generated by TAPAS for dynamic pipeline parallelization of PARSEC’s Dedup | 3 |
| Figure 2.1 | Example of Regular vs Irregular Loop | 6 |
| Figure 2.2 | Loop Unrolling vs Dynamic Tasks | 6 |
| Figure 3.1 | TAPAS Compilation flow | 11 |
| Figure 3.2 | Tapir Detach, Reattach, Sync | 12 |
| Figure 3.3 | High Level Transformations: Stages 1 and 2 | 13 |
| Figure 3.4 | TAPAS Compilation Stages 3 and 4 | 13 |
| Figure 3.5 | TAPAS generated top-level Chisel HDL for nested loop | 14 |
| Figure 3.6 | Execution flow of Nested-loop accelerator generated by TAPAS | 14 |
| Figure 3.7 | Task-2’s Generated Chisel HDL Code | 17 |
| Figure 3.8 | Task-2’s Task Execution Unit (TXU) | 17 |
| Figure 3.9 | Multiple tasks simultaneously outstanding on TXU | 18 |
| Figure 3.10 | Memory Controller. Connects load and store nodes to a memory resource | 19 |
| Figure 3.11 | Fibonacci | 19 |
| Figure 4.1 | Scalability Test Code | 22 |
| Figure 4.2 | Performance Scaling with Workers | 22 |
| Figure 4.3 | ALM Utilization by Sub-block | 24 |
| Figure 4.4 | Matrix Addition | 25 |
| Figure 4.5 | Image Scaling | 25 |
| Figure 4.6 | SAXPY Accelerator. | 26 |
| Figure 4.7 | Stencil Code | 26 |
| Figure 4.8 | Stencil Microarchitecture | 27 |
| Figure 4.9 | Mergesort | 28 |
| Figure 4.10 | Normalized Performance | 29 |
| Figure 4.11 | Performance/watt Gain of Accelerator | 30 |
Chapter 1

Introduction

Industry and academia realize that hardware customization is required to continue performance scaling as semiconductor scaling tapers off. Amazon EC2 [18], Microsoft [29] and Huawei have made FPGAs available to the public through their cloud offerings. To address the challenges of developing application or domain specific hardware, high-level-synthesis (HLS) tools have been introduced. An HLS can automatically produce a register-transfer-level (RTL) circuit specification from a well known software programming language such as C or C++. However, it is an open question whether HLS tools have enough flexibility to permit software engineers with little hardware expertise to design high performance hardware.

A key limitation of HLS tools is their approach to supporting concurrency. Application-specific hardware attains high performance by instantiating multiple execution units that effectively support both coarse-grain and fine-grain concurrency [4, 28, 39] (relative to software). Unfortunately, current HLS tools do not effectively support concurrent languages and find it challenging to generate efficient parallel architectures. This can result in medium-quality accelerators with lower performance than software. HLS tools require sufficient information to generate hardware architectures that can exploit the available concurrency [9]. Tools with C interfaces typically analyze simple loops and deploy techniques such as unrolling, tiling and pipelining [39]. Both Intel (Altera) and Xilinx have targeted their HLS tools towards a regular program in OpenCL SDK [20]. Their approach is primarily aimed at targeting data parallelism. Both vendors offer only limited, proprietary support for OpenCL Pipes - a feature that would permit at least some access to higher level parallelism.

HLS tools have sought to expose techniques to access higher-level parallelism through two approaches i) thread parallelism, and, ii) patterns. Some tools include support for OpenMP loops and pthreads [1, 10], and in the past IBM’s liquid metal [4] supported streaming kernel parallelism. In the pattern-based approach, a hardware expert manually develops a fixed hardware template targeting a specific concurrency pattern, and the application programmer annotates and modifies the application code to fit this template. Common templates include data parallelism, loop parallelism, and loop pipelining [23, 27]. All of them adopt a “construct-and-run” approach, i.e., the HLS tool constructs the hardware statically and schedules concurrent operations during this process. Unfortunately, in many concurrent programs the parallelism evolves as the program runs, either due to control flow [24], or runtime non-determinism [11, 12] (see example in Figure 1.1). Past HLS tools do not support dynamically detected concurrency. Hence, they are
incapable of handling the dynamic concurrency patterns exposed in programs and also do not exploit the ability to run irregular and fine-grain parallelism workloads effectively in hardware.

```
while(!done) {
    spawn_pipe_stage() {
        chunk_t *chunk = get_next_chunk();
        if(chunk == NULL) { done = true; exit(0); }
    }
    spawn_pipe_stage() {
        chunk->is_dup = deduplicate(chunk);
    }
    if(!chunk->is_dup) {
        spawn_pipe_stage() {
            compress(chunk);
        }
    }
    spawn_pipe_stage() {
        write_to_buffer(chunk);
    }
}
```

Figure 1.1: Pseudocode for PARSEC’s Dedup based on CILK-P (modified to enhance clarity)

Observation
A dynamic, task-based accelerator has the needed flexibility for realizing generic, nested, heterogeneous, irregular or regular concurrency patterns. To illustrate this, we briefly discuss an example of the challenges of generating hardware for a dynamically pipelined program using Dedup from PARSEC (see Figures 1.1 and 1.2). The benchmark is parallelized using a streaming pipeline based on CILK-P [22]; the figure includes a commented pseudo code. Recently proposed HLS tools find this particular code sample challenging and cannot generate the microarchitecture based on a static analysis or restricted template based approach. First, the pipeline stages change based on the input data values. 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 a sequence. Finally, the pipeline termination condition needs to be evaluated at runtime and cannot be statically evaluate a priori (e.g., bounded loop). An HLS with the ability to create an arbitrary, flexible task pipeline with dynamically spawned stages would enable creating an efficient accelerator design for such an algorithm.

1.1 Contribution
This dissertation describes a high-level synthesis (HLS) tool chain, TAPAS\(^1\), for generating hardware accelerators from parallel programs that have on-the-fly or dynamic parallelism. TAPAS is a complete HLS framework, based on the Tapir-LLVM compiler that extends the compiler intermediate representation (IR) to include support for parallelism. TAPAS leverages the parallel IR to generate an RTL architecture for a parallel task-based accelerator. An RTL architecture is a design abstraction representing a synchronous

\(^1\)Task Parallel Accelerators
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:

1. Dynamic task spawning enables the program control to skip stages entirely (by not spawning the associated tasks) and change the pipeline communication pattern.
2. The hierarchical task logic organization permits concurrent tasks to be nested. Referring again to the Dedup example in Figure 1.2, TAPAS permits stage-2 to be internally parallelized while ordering the tasks for stage-1.

3. The architecture eliminates dedicated communication ports and allocates local RAM for flexibly communicating data between the tasks. This permits stage-1 to directly pass data to stage-3, when stage-2 is bypassed conditionally.

4. Finally, the architecture does not require any separate control for managing the task dependencies. TAPAS derives the concurrency control from the application and embeds it within the tasks. In the Dedup example, when to stop the pipeline is determined by the get_next_chunk() dataflow embedded within stage-0.

In summary, we have developed TAPAS, an HLS tool that generates parallel hardware accelerators with support for dynamic task parallelism. TAPAS’s framework is based on a parallel compiler intermediate-representation and includes support for arbitrarily nested parallelism and irregular (e.g. not visible until runtime) task parallelism. We have developed a flexible library of hardware components for spawning and synchronizing tasks, buffering tasks, and inter-task communication. We demonstrate that the TAPAS HLS tool can interface these components to generate high performance application-specific parallel hardware. We evaluate both the performance and flexibility of TAPAS on an Intel DE1 SoC Field Programmable Gate Array (FPGA) board and demonstrate that compared to optimized software, task-parallel accelerators can provide better scalability and power efficiency.

1.2 Dissertation Outline

The dissertation is organized as follow: Chapter 2 provides the background material and related ideas which inspired TAPAS. Chapter 3 describes the TAPAS tool chain flow and resulting accelerator architecture. Chapter 4 presents evaluation of TAPAS results when generating accelerators for a number of common parallel patterns. Conclusions and future work are outlined in chapter 5.
Chapter 2

Background

There exists a gap in the quality of the hardware designs generated by HLS tools and those coded by a designer using a lower level hardware description language (HDL) such as VHDL or Verilog. This is partly due to the inability of the HLS tool to comprehend and exploit the parallelism available in software, and partly due to underlying abstractions being unavailable in the hardware architecture. As more resource abundant FPGAs appear in the market, it becomes more feasible for an HLS to support a wider range of parallelism types where the user only has to specify “what tasks can run in parallel” instead of “how the parallel tasks are explicitly mapped to hardware.” In this work, we demonstrate that by adopting the abstraction of tasks and embedding in the architecture the ability to spawn and synchronize tasks, we can create parallel hardware accelerators for dynamic, nested (including recursive), and heterogeneous parallel programs. We map these hardware accelerators to an FPGA and evaluate the benefit of exploiting lightweight tasks.

2.1 Static Parallelism

Consider the loop listed in Figure 2.1(a). Current HLS tools are optimized to parallelize regular loops such as this efficiently since their behavior can be statically analyzed and scheduled. A regular loop has a fixed number of iterations at compile time. If a loop is regular and contains operations with fixed timing, an HLS tool can schedule the execution time of each operation at compile time such that the loop completes as quickly and efficiently as possible. If a loop has unknown bounds, additional exit checking logic needs to be added which increases the amount of logic required and the iteration overhead. A standard method to increase the parallelism of a loop is to fully or partially unroll it. When unrolling, the loop body (e.g. the function bar()) is replicated thereby increasing the opportunities for a compiler or a hardware synthesis tool to exploit fine grained instruction and data level parallelism. HLS tools can fully unroll a loop or partially unroll it to the extent requested by the designer or to the extent allowed by device resources. When a loop is unrolled in hardware, multiple physical copies of the loop body are produced. These can be thought of as separate execution units onto which successive loop iterations are statically scheduled. Additionally, each hardware execution unit can potentially be pipelined to further increase parallelism. Loop pipelining allows multiple iterations of the loop to execute concurrently on a single execution unit. If the loop body is implemented as a series of dataflow operations separated by register stages, then multiple iterations of the loop can proceed through the register stage pipeline concurrently.
The rate at which new iterations can be started is called the initiation interval (II) and will be limited by any loop carried dependencies.

```c
void foo(node_t node[MAX]) {
    int i;
    for (i = 0; i < MAX; i++) {
        bar(&node[i]);
    }
    post();
}
```

(a) Regular Loop

```c
void foo(node_t node[MAX], int len) {
    int i;
    for (i = 0; i < len; i++) {
        if (node[i].valid)
            bar(&node[i]);
    }
    post();
}
```

(b) Irregular Loop

Figure 2.1: Example of Regular vs Irregular Loop

Now consider the more complex, irregular loop listed in Figure 2.1(b). This loop exhibits dynamic parallelism that isn’t fully expressed until runtime. First, the loop bounds are determined by a parameter `len` passed into the function so the number of loop iterations may not be known at compile time. Second, the `bar()` function is only called if the node’s `valid` field has been set true. When an HLS tool tries to unroll this code it must account for the dynamic loop bounds by inserting additional exit logic in the loop body. Conditional logic is also required to check the node’s `valid` status. While the unrolling may allow more parallelism to be exploited, the additional logic consumes both execution time, power, and resources when the unrolled loop is synthesized to hardware. Figure 2.2(a) illustrates unrolling the irregular loop four times creating four copies of the loop body.

![Unrolled Loop Diagram](image)

(a) Unrolled Loop

Figure 2.2: Loop Unrolling vs Dynamic Tasks

2.2 Dynamic Parallelism

The parallelism of the irregular loop in Figure 2.1(b) could be more efficiently exploited if the `bar()` function was dynamically spawned as a separate task only when needed. In that way, `bar()` would only be executed once for each valid node to a maximum of `len` times. The reduced number of dynamic tasks could still be executed concurrently by pipelining a task execution unit and by optionally tiling multiple execution units. Unlike the unrolling, the dynamic spawning would ensure that all execution units are kept busy until all tasks are complete. By unrolling and statically scheduling resources, an HLS
tool must always \textit{plan for the worst case}. It must allocate time and resources for all possible execution flows regardless of whether they are actually executed. An illustration of this unrolling vs dynamic task spawning is shown in Figure 2.2(b).

The ability to dynamically spawn tasks enables more than just improved support for irregular loops. It also enables efficient processing of hierarchical data structures such as adaptive grids and trees. Additionally, recursive algorithms that expose parallelism at each level recursion can also benefit. 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 may be due to a lack of a standard framework to support concurrent execution in hardware like there is in CPUs. Common parallelization frameworks such as OpenMP [1, 6] and Intel Thread Building Blocks (TBB) all build on the abstraction of threads (e.g. pthreads) available across multiple platforms. 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 with low overhead.

It is an open question as to how one best creates an accelerator that supports dynamic spawning. One approach might be to synthesize the run time library of an existing software framework with an HLS. While this may be theoretically possible, the resulting design may be very large and inefficient as the library would have been heavily optimized for a completely different type of architecture. In addition, the framework would likely require support for common OS features, such as dynamic memory allocation, which are not supported by existing HLS tools.

Perhaps the most flexible, feasible approach to build such an accelerator, would be to implement one or more general purpose microprocessor cores in an FPGA and run the target software directly. While such soft-core designs exist, their performance is significantly lower than a dedicated processor. Current FPGA's typically operate at clock frequencies between 100-400 MHz - an order of magnitude slower than a dedicated microprocessor [35]. While FPGA devices have the similar silicon feature sizes as a processor, their generic logic blocks, and in particular, their configurable routing limit the maximum clock speed. Even out of order and hyper-threaded processor soft-cores cannot exploit sufficient parallelism to overcome the clock speed deficit.

Somewhere between a fully synthesized library approach and general purpose processor cores, is the use of custom designed blocks or templates tailored to a particular parallelism pattern such as a pipeline. For example, a library of blocks optimized for fetching, buffering and moving data streams efficiently could be combined with some sort of pipelined task template. The design specific task blocks could be generated by an HLS tool, but the task and data management would use optimized, hand designed blocks and follow the template guidelines. The difficulty with such a strategy is that it can only be used when the program structure closely matches a pipelined pattern.

We present an alternate vision based on abstraction of tasks. Note that the notion of dynamic tasks [2, 17] we discuss in this work is quite different from the static notion of tasks explored in prior work [4]. 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 program dependence graph which implements the functionality. \text{Args[]} is a set of arguments passed to the function and \text{sync} is a runtime 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 runtime. The spawned child tasks will eventually be explicitly synchronized back with the parent so that the continuation and computation dependent on the child can proceed. TAPAS relies on the task abstraction at the compiler level to enable a generalized approach for HLS to generate parallel accelerators that manage the parallelism on-the-fly at runtime.

The simple parallel for-loop (Figure 2.1(b)) illustrates the benefits of adopting tasks as the fundamental unit for implementation in accelerators:

1. **Optimized Context:** Representing tasks within the compiler IR permits TAPAS to determine on a per-task basis what state is allocated to registers and optimize the generated hardware and communication between tasks. For instance, in this particular for loop example we only need to pass the address of node[i].

2. **Dynamic parallelism:** Permitting tasks to dynamically spawn other tasks helps to support flexible parallel patterns. In this loop example, the total number of parallel tasks depends on len argument which needs to dynamically evaluated before triggering a parallel iteration. Furthermore, the body of the loop contains a branch. Unrolling the loop and statically scheduling on separate threads requires additional branch logic and exit check logic. This increases the required loop resources, power consumption and delay.

3. **Simple Synchronization:** Finally, unlike threads which permit arbitrary interactions, tasks implicitly specify synchronization and communicate data only at the boundary. This enables TAPAS to generate accelerators architectures with precise semantics, irrespective of the parallel pattern.

### 2.3 Related Work

While recent works have included limited support for thread creation and OpenMP loops, full support of existing software concurrency frameworks such as CilkPlus, TBB and X10 [2, 37, 40] with rich feature sets seems impractical in the near term due to resource limitations of current FPGAs. For example, the LegUp HLS [7] supports a subset of pthread and OpenMP functionality. However, each software thread is statically mapped to a separate, parallel hardware accelerator block. This severely limits the number of threads that can be created.

There has been extensive work in the context of heterogeneous computing for supporting task parallelism. Asynchronous and task parallel programming has been supported by multiple FPGA toolchains. Prior work largely relied on task abstraction for understanding program structure and then statically scheduled these tasks on the underlying execution units. Several works such as PATS [5] use a software runtime to dynamically schedule tasks on to a hardware accelerator. Similarly, a software Run Time System Manager (RTSM) [8, 9] dynamically spawns software tasks on an available processor, and hardware tasks on an available FPGA resource. OmpSs [16] extends OpenMP to support the spawning of tasks to heterogeneous platforms such as processors, GPUs, and FPGAs. OmpSs creates a fixed set of heterogeneous parallel resources (thread pool) at initiation and software later dynamically decides where a task is to be executed. Unlike prior works, TAPAS provides direct support in hardware for creating and synchronizing tasks dynamically based on accelerator execution. Our work supports dynamic scheduling in which the tasks are assigned in hardware to execution units at runtime. Finally, the dependencies
between tasks in TAPAS (as in software) are enforced through explicit synchronization, while in prior work data dependencies served as synchronization events.

There has been work that supports hardware assistance in the dynamic spawning of tasks for multicore processors. Swarm [21] describes a hardware scheduler that speculatively spawns short tasks to many-core architectures to exploit ordered, irregular parallelism. Carbon [19] adds a hardware scheduler to chip multiprocessors targeting the recognition, mining, and synthesis application domain. Hardware support for asynchronous direct messages (ADM) that bypass memory for communication between threads has been proposed [33] as a method to accelerate a range of software based task manager implementations. XLOOPS [36] proposes specialized loop parallelism instructions and hardware to support for accelerating vector architectures. A channel abstraction was implemented on GPUs [26] to facilitate dynamically aggregating asynchronous tasks into course grained tasks. In contrast, our work focuses on supporting task spawning of synthesized hardware tasks within application-specific accelerators. TAPAS also synthesizes the control logic for the concurrent tasks from the program, while in prior work the control logic continues to run in software or requires manually designed control logic.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Seq. Kernel Pattern</td>
<td>Pattern Tasks</td>
<td>Parallel Prog.</td>
<td></td>
</tr>
<tr>
<td>Hints</td>
<td>#pragma Kernels Pattern</td>
<td>Spawn/Sync</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Loops</td>
<td>Unroll/Pipeline Parallelize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heterogeneity</td>
<td>— Limited</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested parallel</td>
<td>— Limited</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lightweight</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multithreading</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicore</td>
<td>✓ Multiple execution units</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Comparing the features of HLS tools

Table 2.1 summarizes the feature set of current HLS tools. Industry standard HLS tools Vivado [39] and Intel-HLS primarily target sequential programs. A parallel architecture is often realized by using an HLS tool to synthesize a single hardware core and then requiring a developer to manually instantiate multiple instances of that core in a hardware description language. Both Vivado and Intel-HLS unroll and pipeline loops to convert loop parallelism to instruction level parallelism. Achieving efficient hardware with HLS tools often requires a software developer to identify where it might be feasible to apply these specific optimization techniques and add additional pragmas to the program, which is challenging for those less familiar with the intricacies of hardware design. HLS tools primarily target data parallel kernels. However, they have adopted a kernel-based approach and a subset of OpenCL, OpenMP or Pthread specifications to allow programmers to convey some types of higher level parallelism beyond just data or instruction parallelism. Support for higher level parallelism is limited though, because HLS tools construct the hardware statically and do not support dynamic spawning, asynchronous, or nested parallelism. Even this restricted parallelism support typically requires numerous, proprietary user annotations (e.g. pragmas) to help the tools discover parallelism and determine when various hardware structures can be used.
Finally, a promising avenue of research is HLS for domain-specific patterns. This approach requires the hardware expert to design a parameterized template targeting a program pattern (e.g., pipeline). The software developer can deploy HLS within the application only when the program structure matches the target pattern. For each new pattern from other domains a hardware expert has to develop new concurrency controllers. Concurrency patterns also risk becoming obsolete and hence toolchains typically target higher level domain-specific hints whose utility is well defined.

*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. A key novelty of tasks in *TAPAS* is that they can dynamically spawn and sync with other tasks. This enables *TAPAS* to handle a variety of common programming patterns including nested parallelism (e.g. nested loops) and heterogeneous parallelism (e.g. concurrent heterogeneous tasks). Further, *TAPAS* with dynamic tasks is able to handle underlying non-determinism in the memory system.
Chapter 3

TAPAS: Task Parallel Accelerators

TAPAS takes a hierarchical approach for generating the RTL for the dynamic task-based microarchitecture. Here we describe a series of compilation steps which automate the process of generating an efficient hardware architecture and low-level RTL from a compiler intermediate representation (IR) that includes parallelism markers. We develop TAPAS HLS as a separate LLVM backend that processes and transforms its parallel compiler IR into a suitable microarchitecture through a series of steps. As seen in Figure 3.1, these steps fall into two categories: high level concurrent task transformations in Stages 1,2 (Section 3.2), and low level analyses and hardware task execution generation in Stages 3,4 (Section 3.2.3). In the first two stages a parallel compiler is used to convert source code to LLVM’s intermediate representation (IR). Then a number of LLVM optimization passes are run to generate a program dependence graph and delineate parallel tasks. In the last two stages the accelerator is constructed from a library of highly configurable building blocks written in Chisel HDL. The majority of the library building blocks match one-for-one with LLVM IR instructions (e.g. add, compare, multiply blocks) with some additional complex blocks for controlling loops and dynamic task management. The architecture can then be synthesized using the Chisel compiler and the target SoC’s backend tools.

Figure 3.1: TAPAS Compilation flow

3.1 High Level Transformations: Stages 1 and 2

To comprehend the semantics required for a task-based accelerator architecture TAPAS relies on Tapir [34]. Tapir adds three instructions to demarcate and synchronize concurrent regions in the task graph: detach, reattach, and sync. Detach and reattach together delineate a concurrent task. A detach instruction terminates the block that contains it, spaws 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. The sync instruction dynamically waits for all tasks in a parallel context
to complete. Since Tapir assumes a generic threadpool execution model that can execute on top of any runtime, it leaves the markers in place in the original PDG (Program Dependency Graph). Figure 3.2 illustrates the control flow graph for a simple spawn generated by Tapir. The dotted det.achd basic block contains the parallel task delineated by the detach/reattach pair. Note that Tapir ensures that any inputs to the detached task are stored in shared memory. Likewise, outputs from the detached task must be explicitly loaded from shared memory in any continuation blocks.

```c
int simple_spawn(int i, int j) {
    cilk_spawn{i++;}
    j--;
    cilk_sync;
    return (i+j);
}
```

(a) Code

(b) Hardware

Figure 3.2: Tapir Detach, Reattach, Sync

Since we do not support generic threadpools in hardware we must customize the execution units for each task. We leverage the markers to perform reachability analysis and extract an explicit task graph of task nodes and dependency edges. This becomes the architectural blueprint for our parallel accelerator. Nesting, loops, and irregular flows are analyzed in this stage and in the resulting task graph all task relations are explicitly defined. The edges in the task graph carry information about the values being passed from the parent task to the child task. TAPAS incorporates standard definition-use and live analysis routines to create the requisite arguments that need to be passed between tasks.

Figure 3.3 illustrates a nested cilk_for loop (a) and the resulting program dependency graph and dynamic task graph (b). The logic to implement each level of the nested for loops, as well as the loop body addition are each extracted explicit tasks.

### 3.2 Generating a Task Parallel Architecture: Stage 3 and 4

Once parallel tasks have been delineated and the program dependence graph created in Stage 1 and 2 (§ 3.1), the hardware accelerator architecture is constructed. The generated accelerator consists of multiple task units at the top-level with each task unit representing a unique task type. Figure 3.4 illustrates the interconnection of three task units corresponding to T0, T1, and T2 of the nested loop example in the


previous section. Figure 3.5 illustrates an excerpt from the the top-level Chisel HDL code and associated parameters for the same example.

### 3.2.1 Task Units

*TAPAS* supports time multiplexing (equivalent to simultaneous multithreading) of multiple tasks on an execution unit, tiling of execution units (equivalent to multicore), and dynamic assignment of tasks at runtime to different execution units. A task unit serves as a basic building block in the architecture. It is a fully-functional execution engine for a single task. 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) 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.

---

**Figure 3.3: High Level Transformations: Stages 1 and 2**

**Figure 3.4: TAPAS Compilation Stages 3 and 4**
class NestedAccelerator(implicit p: Parameters) extends NestedAcceleratorIO {

/******* Instantiate Cache and Memories ***********/
val SharedL1cache = Module(new Cache)
// Scratchpad SRAM
val SharedScratchpad = Module(new ScratchMem(size=256))

/******* Instantiate Task Units *************/
val Task-0 = Module(new TaskController(Nt=32, Nr=32, ParentPort = 1, ChildPort=1, new T0-DF()))
val Task-1 = Module(new TaskController(Nt=32, Nr=32, ParentPort = 1, ChildPort=1, new T1-DF()))
val Task-2 = Module(new TaskController(Nt=32, Nr=32, ParentPort = 1, ChildPort=1, new T2-DF()))

/******* Connect Task Units ***************/
Task-0.io.in <> io.in
io.out <> Task-0.io.out
Task-1.io.detach.in <> Task-0.io.spawn.out
Task-0.io.sync.in <> Task-0.io.out
Task-2.io.detach.in <> Task-1.io.spawn.out
Task-2.io.sync.in <> Task-1.io.out

Figure 3.5: TAPAS generated top-level Chisel HDL for nested loop.

Figure 3.6: Execution flow of Nested-loop accelerator generated by TAPAS

3.2.2 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 3.6). The task graph in the figure illustrates three tasks T0,
the outer loop control and spawner of N instances of inner loop. T1 is the inner loop control and spawner of N instances of body. Finally, T2 performs the actual work; reads elements from the A[i][j], B[i][j] and adds them. Note that in this application N dynamic instance 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 one of the following states: READY, spawned but not allocated to a TXU, EXE, TXU allocated and but not complete, COMPLETE, execution complete and need to synchronize with parent, and 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). (1) illustrates a spawn operation with T0 initiating the task corresponding to the inner loop iteration, T1. A spawn is a tuple, \((\text{Args}[,].\text{ParentID})\). The ParentID consists of SID:DyID. The SID refers to the name of the parent task (in this instance T0) and the DyID corresponds to the index of the task queue corresponding to the parent task (index 0 here). Hence this operation corresponds to dynamic task T0:0 spawning a dynamic instance of T1 (inner j loop for i = 0). The ParentID metadata available in the spawn is noted down in the task queue 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,...iteration) if there is enough queue space available in task unit T1. In (3) as the instances of T2 complete they synchronize with their parent task that spawned them. By the parallel semantics of the compiler IR, 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 corresponding to T1:0, entry 0 here. The purpose of noting down the SID and DyID when the T2 tasks were spawned is clear now. The SID permits composability and allows multiple tasks to invoke tasks on T2 and 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 with the spawn and sync of the dependent tasks and decouples task creation from task execution. The spawn and sync interfaces are latency insensitive and enables an accelerator designer to transparently parameterize the resources per task unit and transparently compose the task unit.

### 3.2.3 Generating Task Execution Units (TXU)

The TXU represents the execution engine within each task unit. The generator parameterizes and permits the number of TXUs to be varied per task unit. This tile-based architecture enables an accelerator designer to easily scale the amount of parallelism available in the hardware. TAPAS-generated TXUs are fully pipelined execution units permitting multiple dynamic instances of a task to simultaneously execute on the same TXU (à la simultaneous multithreading or Hyperthreading). The TXUs only communicate at task boundaries and even then do not directly communicate with each other. All inter-TXU communication is marshaled through a shared local memory or the cache.
TAPAS generates the logic for the TXUs based on the per-task sub-program-dependence-graph (sub-PDG) earmarked by our compiler. Each TXU is a dataflow executor that enables fine-grain instruction level parallelism to be mined. TAPAS’s pipelined dataflow model ensures that each TXU unit exports a latency insensitive interface and is composable with TXUs generated by other tasks. TAPAS decouples functionality from timing constraints when implementing each task-execution unit (TXU).

An automatic pipelining process introduces latency insensitive ready-valid interfaces between each operation in the dataflow and can handle any non-determinism. As a result, a dataflow graph mapped to the TXU may contain nodes with multiple cycle latency (e.g., floating point operations), and unpredictable 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. The statically determined implementation in existing HLS tools is inherently fragile and changes in the anticipated program flow or operation latency may lead to a painful redesign and re-synthesis.

The translation from a sub-PDG composed of IR instructions to a TXU dataflow is direct. That is to say, the TXU is a direct manifestation of the program dependency graph. Each IR instruction node has a corresponding Chisel library node with equivalent semantics. Each data or control dependency edge becomes a control or data wire interconnecting the Chisel nodes. For example, an unsigned add instruction becomes an unsigned adder node, a signed division becomes a signed divider node, etc. The same is true for the control flow instructions such as conditional branch. This simple one-to-one translation greatly simplifies ensuring the semantic correctness of the resulting accelerator logic.

Even the more complicated Tapir detach, reattach, and sync instructions have direct hardware library equivalents. For example, the detach instruction terminates its current basic block and takes a detached block D and a continuation block C as its arguments. The detach instruction spawns the task starting at block B, allowing that task to execute in parallel with block C. Essentially it operates much like a conditional branch instruction except that both branches are activated simultaneously. Additionally, a detach instruction must signal to a sync instruction that a new task has been created. The corresponding Chisel detach node does the exact same thing, activating two downstream basic blocks in the dataflow logic as well as signal that a new task has been started to a Chisel sync node generated by a he corresponding sync IR instruction. Likewise, a Chisel reattach node signals the sync node upon the detached task completion (e.g. when it joins). The sync node actively tracks the state of detached tasks and only activates downstream logic when all spawned tasks are complete.

Tapir makes minimal assumptions about the consistency of concurrent memory accesses and assumes that memory is shared among parallel tasks and that any register state is local to each task. TAPAS supports this model by providing local registers and stack memory for each task control unit. Each concurrent task executing on the TXU blocks has its own stack frame within the STACK memory. Additionally, each TXU can access the shared stack memory of any parent task or even global memory via the cache.

An example of the top level Chisel HDL code generated by TAPAS is provided in Figure 3.7. It shows a portion the code generated by the line:

\[ C[i][j] = A[i][j] + B[i][j] \]

The required load, adder and store operators are declared as Chisel “Modules” at the top of the code. For example, LoadA is a module of the LoadNode type. Additionally, the module has parameters "ID=0"
/* Create load operators: load A[] and B[] */
val LoadA = Module(new LoadNode(ID=0,RouteID=0)(new DataBundle(32.W)))
val LoadB = Module(new LoadNode(ID=1,RouteID=1)(new DataBundle(32.W)))

/* Create 32bit Integer add operator */
val Add = Module(new ALU(ID=3,"SAdd")(new DataBundle(32.W)))

/* Create store operator: store C[] */
val StoreC = Module(new StoreNode(ID=2,RouteID=0)(new DataBundle(32.W)))

/* Wire up Load, Add, and Store nodes */
/* sink <> source */
Add.io.LeftIO <> LoadA.io.Out
Add.io.RightIO <> LoadB.io.Out
StoreC.io.inData <> Add.io.Out

Figure 3.7: Task-2’s Generated Chisel HDL Code

and "RouteID=0", and operates on the data type "DataBundle(32.W)" which is a 32 bit wide integer with a ready-valid interface. Further down in the code the module datapaths are connected.

Figure 3.8: Task-2’s Task Execution Unit (TXU)

Figure 3.8 shows the corresponding function units created by this Chisel HDL code. The add function unit communicates with Load A[][], Load B[][], + and Store C[][] via decoupled hand-shaking 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 3.9. The dynamic task ids corresponding to queue index are allocated at runtime. In the first cycle, the arguments i,j and base addresses *A, *B, *C are sent to the TXU. The first dynamic instance of T2 is T2:0, the arguments i=0, j=0, *a,*b,*c. These tokens are processed by the address calculator in the first cycle. In the second cycle, the load operations are fed the memory addresses for A[0][0] and B[0][0], while the second dynamic task, T2:1 is initiated with arguments i=0,j=1, *a,*b,*c. When the loads complete and the A[0][0] and
B[0][0] reach the + the third task is initiated. 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 where tasks can complete in any order depending on only data availability [15].

3.2.4 Task Memory Interface and Memory Model

In this dissertation, we consider using TAPAS to target a heterogeneous FPGA SoC where both the processing cores and accelerator logic are integrated into a single chip. In this context, the accelerator FPGA fabric is connected to the processor’s L2 cache over an AXI bus. As a result, accelerator designs have access to processor memory as well as their own internal SRAM memory resources. A key question is what is the accelerator’s model for accessing the processor’s memory? Unlike prior work which primarily targeted streaming memory models [27], TAPAS permits arbitrary task graph patterns to be converted into accelerators and thus needs to support a more flexible interface. TAPAS uses a simple local cache to provide flexible but high performance access to the processor’s memory. It is conceivable that the task parallelism could also help to effectively hide the non-deterministic latency of the cache while providing a programming model that software is familiar with [30–32].

In addition to the cache interface to the processor’s memory, the accelerator has three types of local memory for temporary storage: scratchpad RAM, stack RAM, and registers. All SSA variable assignments in the IR are converted to registers in the TXU dataflow. Any stack memory accesses are connected to the local Stack RAMs associated with each task control unit. Local scratch pad memory is used for any data structures used exclusively by accelerator blocks. Any loads and stores that can’t be confirmed as stack or of local scope are connected to the cache for the processor’s memory.

A key implementation issue is how to route values from the cache or local RAMs to the TXUs and furthermore the internal dataflow nodes within the TXU. The memory controller (see Figure 3.10 connects a memory operation in the TXU to a memory resource such as a cache, scratchpad RAM, or local stack. We provide full support for unaligned accesses and masking logic to handle half words and bytes. We chose to group the common logic for multiple memory operations into a memory controller block to minimize resource requirements. Figure 3.10 shows the architecture of the memory controller. It consists of the following parameterized microarchitecture components i) an in-arbiter tree network that arbitrates between requests to the memory interface, ii) an out demux network that routes responses back to the
requesting node, and iii) a table of staging buffers that contain the actual logic for reading the required bytes from the memory resource. Both the request and response networks are statically routed.

3.2.5 Recursive Tasks

TAPAS’s ability to dynamically spawn and queue tasks coupled with its support of stack memory allows recursive algorithms to be run on hardware accelerators. Typical HLS tools have limited support for recursion (like some forms of tail recursion only [39]) or do not support it at all [7]. TAPAS, however, can support a wider range of recursive code. Figure 3.11 uses the simple Fibonacci algorithm to illustrate a general form of a simple recursive algorithm with a base case and a recursive case. In the Fibonacci example the recursive case is multiply recursive. Any subsequent instructions after the recursive calls are deferred until the base case is executed. Tapir can support algorithms in such a form by spawning the recursive calls as parallel tasks.

```c
1 void fib(int n)
2 {
3 if (n < 2) {
4     return (n);
5 }
6    cilk_spawn {x=fib(n-1);}
7    cilk_spawn {y=fib(n-2);}
8    cilk_sync;
9    return (x+y);
10 }
```

(b) Fibonacci Example using Cilk

Figure 3.11: Fibonacci
The main body of the recursive function in Figure 3.11(b) is shaded in gray. This portion of the function is effectively non-blocking and so any dataflow implementation of this logic can be called repeatedly. It is non-blocking because it ends in either a return function for the base case, or a sync function for the recursive case. The sync function decouples the main body of fib() in the example from the deferred code \( \text{return}(x+y) \) and allows the deferred code to execute at a later point in time the base case has been found. Since the deferred code will be executed in reverse order to the main body of the code, it must be passed the result values \((x,y)\) via the stack rather than registers. As with any software implementation, TAPAS’s support for recursion depth is limited by available stack size. It is important to note that all recursive calls must be spawned as parallel tasks. Spawning any recursive calls allows them to be queued for future execution and lets the parent task continue to completion freeing the TXU dataflow logic. TAPAS does not support sequential recursive calls as in that case the parent task would not complete and free the TXU logic. This would result in a deadlock condition.
Chapter 4

Evaluation

This chapter evaluates the characteristics of several accelerator designs generated by the TAPIR infrastructure described Chapter 3. The evaluation investigates the following areas:

- **Scalability**: How does performance scale as parallel resources are increased? At what point is the parallelism too fine grained to be exploited with a dynamic task based model?

- **Resource Utilization**: How many resources are consumed by each benchmark? What percentage of the resources are lost to task management overhead?

- **Performance**: How does the performance/watt of an FPGA accelerator solution compare to a CPU based solution across several micro-benchmarks?

4.1 Task Granularity

The overhead of spawning a task on an FPGA is significantly less than in software. This increases the range of parallelism opportunities that can be successfully exploited through spawning a separate task. Since task creation requires only a single `detach` command even small, fine grained tasks can provide a performance improvement.

The code in Figure 4.1(a) was synthesized on an FPGA to see how fast tasks can be effectively spawned from inside a simple parallel `for` loop. A simplified diagram of the generated hardware structure is shown in (b). The maximum spawning rate for this example is determined by the sequential code that increments and tests the loop counter (currently 7 clock cycles). The spawned task body contains the load, addition, and store of a 32 bit integer \( a[i] \). Additionally, for the purposes of our testing, the generated hardware architecture was modified to allow easy insertion of additional adder stages. This provided a simple mechanism for increasing the duration and resources required for the spawned task. \(^1\)

The performance is plotted against an increasing number of worker tiles in Figure 4.2. Assuming a clock speed of 300 MHz (Arria 10) for the target device, the maximum possible spawn rate is 40 million spawns/second based on the loop iteration rate. As can be seen in the figure, even for small workloads the performance scales with the addition of parallel worker tiles. For a load of 50 adders the performance scales linearly up to 4 tiles before starting to flatten for 5 tiles. For a rough comparison to a software

\(^1\)Running multiple tasks concurrently on the same tile was explicitly disabled so that the overhead of using multiple task tiles for parallelism could be measured.
solution some equivalent code was run on an Intel I7 workstation\textsuperscript{2} using the Tapir Cilk compiler with its default optimization settings. The results for spawning a task of 50 additions are shown as ‘Software’ in the graph. As is expected, the performance is flat and Cilk spawning provides no benefit at this level of granularity due to the much higher spawning overhead. Our architecture allows an FPGA to exploit these fine grained opportunities for parallelism while still providing the flexibility of expression and support for dynamic parallelism found on a more traditional software platform.

\textsuperscript{2}Intel Core i7-3770 CPU @ 3.40GHz
4.2 Resource Utilization

Table 4.1 shows per instruction and per tile resource utilization of the accelerator. The test code was synthesized for two Intel SoC FPGAs to see how many worker tiles of a given size could be fit on a chip and what clock speed would result (see Table 4.1). For the older Cyclone V SoC, the largest test case of 10 tiles of 50 add instructions filled 85% of the chip with clock speed 153 MHz. The same design on a newer Arria 10 SoC reached a clock speed was 308 MHz and took approximately 12% of the chip. This suggests that the newer FPGAs in particular, have the potential to support dozens of parallel tasks with each containing hundreds of instructions. The single M10K block RAM consumed is for queuing the spawned tasks in the task controller logic. Enough memory was provided to support queuing and management of up to 32 simultaneous dynamic tasks.

<table>
<thead>
<tr>
<th>Freq (MHz)</th>
<th>Tile</th>
<th>Ins.</th>
<th>ALMs</th>
<th>Regs</th>
<th>M10K RAM</th>
<th>% Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclone V 5CSEMA5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>185.46</td>
<td>1</td>
<td>1</td>
<td>1314</td>
<td>1424</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>178.09</td>
<td>1</td>
<td>50</td>
<td>2955</td>
<td>3523</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>153.61</td>
<td>10</td>
<td>1</td>
<td>7107</td>
<td>8547</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>159.24</td>
<td>10</td>
<td>50</td>
<td>24738</td>
<td>27604</td>
<td>1</td>
<td>85</td>
</tr>
<tr>
<td>Arria 10 10AS066</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>308</td>
<td>10</td>
<td>50</td>
<td>28844</td>
<td>27659</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4.1: FPGA Utilization

Figure 4.3 shows the relative amount of ALM resources used by each sub-block of the design. In the first case the dataflow logic is very light and more than 50% of the logic has been used for control units and other part of the design. However, once the actual computation portion of the worker tiles grows to more than a few instructions the resources that are used for control logic becomes comparatively light.

There is another trade-off between the number of tiles and number of instructions for each tile. When comparing a single tile versus 10 for the case of a single instruction, we can see that the amount of resources needed to support the additional tiles (task controller, memory arbitration) is small. Another observation is that our overall logic needed to support arbitrated memory access on average is less than 10%. It shows that dynamically spawning a new task in our accelerator design can be achieved without a significant increase in logic and complexity.

4.3 Performance

This section evaluates the performance of the accelerators relative to their energy consumption. A series of micro-benchmarks have been selected that exhibit a number of common parallel patterns. The benchmark performance of the accelerator is then compared to the same benchmark run using Cilk on a standard PC.

4.3.1 Benchmarks

Table 4.2 gives a brief summary of the accelerator benchmarks and shows the characteristics of each benchmark. In the following sections we give a brief description of each benchmark kernel; our emphasis

---

3 Adaptive Logic Module
Figure 4.3: ALM Utilization by Sub-block

is on being able to implement accelerators for these workloads without requiring additional effort from
the programmer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bench.</th>
<th>Parallel Pattern</th>
<th>Mem Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mat. Addition</td>
<td>In-house</td>
<td>Nested parallel loops</td>
<td>Regular</td>
</tr>
<tr>
<td>Image Scale</td>
<td>In-house</td>
<td>Parallel loop w/ conditional</td>
<td>Regular</td>
</tr>
<tr>
<td>Saxpy</td>
<td>BLAS</td>
<td>Parallel loop w/ dynamic bound</td>
<td>Regular</td>
</tr>
<tr>
<td>Stencil</td>
<td>MachSuite</td>
<td>Nested parallel/serial loops</td>
<td>Regular</td>
</tr>
<tr>
<td>Dedup</td>
<td>Parsec</td>
<td>Pipeline</td>
<td>Irregular</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>In-house</td>
<td>Recursion</td>
<td>Regular</td>
</tr>
<tr>
<td>Mergesort</td>
<td>In-house</td>
<td>Recursion</td>
<td>Regular</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmark Properties

**Matrix Addition - Nested parallelism**

Figure 4.4 shows the code for matrix addition. It has two nested parallel loops. A standard HLS can parallelize this in two ways: 1) flattening to a single level, or, 2) having the HLS tool only parallelize the innermost loop (which is not parallel in this case) while executing the other levels serially. In contrast, TAPAS, decomposes nested parallelism into multiple units, each with potentially multiple tiles, and leverages all parallelism levels without programmer intervention. Furthermore, it can transparently encapsulate the innermost serial for loop in a task and optimize the implementation without requiring any changes in the task controllers of the outer loops.

**Image Scaling - Imbalanced iterations**

Listing 4.5 shows vector scaling code for image manipulation. While the loop is parallel, iterations contain one-sided conditionals which lead to an imbalance in latency. If the loop iterations are unrolled into
```c
void mat_add(int **a, int **b, int **c) {
    cilk_for (int i = 0; i < NROWS; i++) {
        cilk_for (int j = 0; j < NCOLS; j++) {
            c[i][j]=a[i][j]+b[i][j];
        }
    }
}
```

Figure 4.4: Matrix Addition

multiple copies by an HLS, the copies may exhibit unbalanced latency which leads to a longer critical path. In this example, if a pixel value is negative it will be clipped to a zero value and a portion of the unrolled logic will remain idle until all other copies are complete. TAPAS, however, will not suffer from this imbalance because a worker tile will be dynamically assigned a new pixel to process as soon as it is free.

```c
void image_scale(int *a, int *c, int scale, int N) {
    cilk_for (int32_t i = 0; i < N; i++) {
        /* Clip pixel values */
        if (a[i] < 0) {
            c[i] = 0;
        } else {
            c[i] = (a[i] * scale) >> 8; /* scale is 24.8 bit fixed point */
            if (c[i] > 255) c[i] = 255;
        }
    }
}
```

Figure 4.5: Image Scaling

**SAXPY - Dynamic loop bound**

*SAXPY* is a linear algebra code. It is an embarrassingly parallel loop; however, the loop bound \( n \) is unknown statically as shown in (Figure 4.6(a)). In current HLS tools, the programmer would have to unroll the loop to mine parallelism. However, an unknown bound implies they can only partially unroll the loop by a constant factor supplied statically. Ideally, this factor should be large to mine more parallelism. However, the case where the bound \( n \) is not a multiple of this unroll factor must also be handled, using a sequential remainder loop or side exits. A larger unroll factor will lead to larger resource costs for side exits or longer time spent in remainder loops. In contrast, TAPAS naturally handles unknown bounds and does not waste resources or time, because it dynamically binds iterations to tiles. The generated hardware looks like Figure 4.6(b). The loop header task generates and queues a spawn to the body’s task unit. Note that the spawns themselves are sequential and header task evaluates the bounds check before spawning.
static void saxpy(int n, int a, int* x, int* y) {
    /* Root task */
    cilk_for (int i=0;i<n;++i)
        /* Workers */
        y[i] = a * x[i] + y[i];
}

Figure 4.6: SAXPY Accelerator.

Stencil - Nested Parallel and serial loops

Stencil is an iterative kernel (Figure 4.7) that updates array elements based on a fixed pattern. Stencil is a highly parallelizable kernel in that each array element is independent from the other elements. Our parallel implementation of stencil includes a parallel for-loop that iterates over a fixed length array and computes a stencil pattern for each element. The inner stencil pattern is calculated by two nested serial loops. It is worth noting that TAPAS supports arbitrary nesting of loops; serial and parallel loops can be nested in any order to any depth (resources permitting). Additionally, loops can optionally be unrolled using standard LLVM command line options or pragmas.

```c
void stencil () {
    /* Parallel for loop */
    cilk_for (unsigned pos = 0; pos < NROWS * NCOLS; pos++) {
        unsigned i = pos / NCOLS;
        unsigned j = pos & (NCOLS-1);
        /* Serial for loops */
        for (unsigned nr = 0; nr <= 2*NBRROWS; nr++) {
            for (unsigned nc = 0; nc <= 2*NBRCOLS; nc++) {
                unsigned row = i + nr - NBRROWS;
                unsigned col = j + nc - NBRCOLS;
                if ((row < NROWS)) {
                    if ((col < NCOLS)) {
                        out[i * NCOLS + j] += in[row * NCOLS + col];
                    }
                }
            }
        }
        out[i * NCOLS + j] =
            (out[i * NCOLS + j] + (FILTROWS * FILTCOLS)) /
            (FILTROWS * FILTCOLS);
    }
}
```

Figure 4.7: Stencil Code

Dedup - Pipeline parallelism

Dedup code is depicted in Figure 1.1, and described in Section 1. It demonstrates multiple advantages of using TAPAS. The first advantage is that dedup has an irregular pipeline pattern. It does not quite conform to a typical hardware pipeline template. HLS tools support pipelining, however, they have several
restrictions on the pipeline structure. Typically, they support linear pipelines connected by FIFO queues. While dedup could use FIFO queues, it does not have a strictly linear pipeline. \( S2 \) is a conditional stage which is not always executed. HLS pipelines would nevertheless require its data to be passed through the \( S2 \) stage even if there is no computation to be performed. TAPAS, with its decoupled execution model, can easily bypass this stage when not required, therefore saving latency and data movement energy.

Another restriction of HLS pipelines is that they require scalar FIFO queues for communication between stages. Imposing such a condition on dedup would lead to losing parallelism in the \( S2 \) stage, which can execute out-of-order. However, TAPAS can exploit this parallelism by using a wide (superscalar) task queue.

Finally, the loop termination condition is dynamically determined, which is difficult for HLS to handle, as in the case of SAXPY. This problem is further exacerbated for dedup because the termination value is not known even dynamically, until the last iteration is reached. Therefore, blocks cannot be configured with the correct iteration count, even dynamically. TAPAS easily handles this case using dynamic allocation.

**Fibonacci and Mergesort - Recursion**
The general approach to handling recursive code and the Fibonacci algorithm in particular was described in Section 3.2.5. Here we’ll describe a parallel implementation of Mergesort. Figure 4.9(a) lists commented code for the recursive mergesort() function. Two recursive mergesort() calls are spawned followed by a sync. The sync will ensure that the deferred call to the function mergesort_merge() will only run after the preceding two recursive calls have completed. Figure 4.9(b) shows a high level view of the hardware architecture generated for this code. Note how the mergesort() task controller can manage multiple worker tiles of the mergesort TXU. In turn, each TXU tile is able to spawn new recursive calls to its own task controller. With multiple tiles spawning multiple tasks in parallel the task controller could potentially
become overloaded. For this reason, the Task Controller was designed to support processing new task every clock cycle to reduce its chance of becoming a bottleneck.

```c
void mergesort(unsigned B[], unsigned iBegin, unsigned iEnd, unsigned A[]) {
  continue_t p;
  p.A = B; // Note: swap of A<->B is intentional
  p.B = A;
  p.iBegin = iBegin;
  p.iEnd = iEnd;

  if (iEnd - iBegin < 2) // if run size == 1
    return; // consider it sorted

  // split the run longer than 1 item into halves
  unsigned iMiddle = (iEnd + iBegin) / 2; // iMiddle = mid point
  p.iMiddle = iMiddle;

  // recursively sort both runs from array A[] into B[]
  cilk_spawn mergesort(A, iBegin, iMiddle, B); // sort the left run
  cilk_spawn mergesort(A, iMiddle, iEnd, B); // sort the right run
  cilk_sync;

  // merge the resulting runs from array B[] into A[]
  mergesort_merge(&p);
}
```

(a) Mergesort using Cilk

(b) Microarchitecture

Figure 4.9: Mergesort

4.3.2 Results

Even though the overhead required to support dynamic parallelism and complex tasks is higher than a more restrictive statically scheduled approach, it is still low enough to exploit the parallelism inherent in fine-grained tasks. To demonstrate this, we compare how the performance scales when we parallelize seven benchmarks by adding additional worker tiles. The performance change is compared against a baseline of our pipelined implementation with only a single task tile. We contrast the accelerator results against CPU results in Figure 4.10. Part (a) demonstrates that CPU performance generally deteriorates
as more workers are added. This is as expected because the benchmarks are very small and the added overhead of managing multiple workers exceeds any possible benefit. One exception is in the recursive Fibonacci benchmark where improvement is seen up to 4 workers. Part (b) shows that the accelerator has increasing performance from the baseline across all examples although two of the benchmarks (dedup and image scale) quickly become bandwidth limited.

The Matrix Addition example scales well initially up to four tiles of the outer loop. The inner loop eventually hits the memory bandwidth while issuing memory operations limiting further performance improvement. Saxpy and image scale benchmarks also quickly saturate the DRAM cache bandwidth as their inner loops are dominated by memory reads and writes. The Stencil benchmark is more computationally intense and consequently scales well even up to 8 workers.

The dedup benchmark is a bit different as it is composed of multiple pipelined tasks that are already running in parallel in the baseline case. Adding additional tiles of a given stage will improve performance if that particular stage is significantly slower than the other stages in the pipeline. In the case of our simple dedup example, the third stage (S2) that performs the compression can become a bottleneck on input data streams that have many duplicates. The performance increased by spawning to two separate (S2) tiles but no further improvement is seen for four or eight tiles.

Figure 4.10: Normalized Performance
In Table 4.3 we compare power consumption and resource utilization of the generated accelerator for each application. The accelerator power measurement is generated using the Intel FPGA Quartus PowerPlay Analyzer tool. It is a measure of total power (static and dynamic) based on signal activity levels derived from simulation. Mergesort is the largest design but still uses less than half of the available chip resources.

<table>
<thead>
<tr>
<th>Design</th>
<th>Worker Tiles</th>
<th>Freq (MHz)</th>
<th>ALMs</th>
<th>Registers</th>
<th>M10K RAM</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 Addition</td>
<td>3</td>
<td>223</td>
<td>4702</td>
<td>7025</td>
<td>3</td>
<td>0.677</td>
</tr>
<tr>
<td>Image Scale</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>

Table 4.3: Benchmark Results on Cyclone V

We compared the performance/watt of the most efficient accelerator implementations against the most efficient Cilk configuration on a CPU using the same Intel i7 processor as before. The power measure for the CPU is package power reported by the likwid-powermeter tool. The package power is the total power (static and dynamic) which is most comparable to the FPGA measurements. Neither power measurement includes external memory power. The performance is measured in benchmark executions per second. The results are plotted in Figure 4.11. As can be seen, the accelerator designs exceed the energy efficiency of the CPU for all benchmarks and in one case by up to $27 \times$.

![Figure 4.11: Performance/watt Gain of Accelerator](image-url)
Chapter 5

Conclusion

5.1 Conclusions

TAPAS’s primary goal is to provide an intuitive HLS toolchain for software programmers to generate parallel accelerators capable of supporting dynamic task parallelism. The examples we discussed should illustrate the capability of TAPAS to generate accelerators for complex concurrency patterns. 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 runtime. We also demonstrate that the resulting accelerators can exceed the power efficiency of equivalent software-only solutions by up to 27 times.

5.2 Current limitations

While TAPAS adds important new capabilities to HLS tool flows, it still shares many of the same common restrictions found in other HLS tools. For example, few if any tools support for dynamic memory allocation. At the time of writing, TAPAS has the following additional limitations of note:

1. Insufficient stack or task buffer memory allocation could result in stalls or deadlock. This is especially true the case of recursion.

2. Mutual recursion (e.g. foo() calls bar(), bar() calls foo()) is untested and unsupported.

3. Single recursion is supported, but not yet fully automated. Manual intervention is required to ensure live-ins to the deferred code is passed via the stack.

4. No floating point support

5. Memory blocks are instantiated manually

6. Task units and RAMs are connected manually

7. A single Tapir synce IR instruction can dynamically wait for all detached tasks including and any descendent tasks to complete. While this is supported by the Chisel library components, current automated tools will only connect a sync node to tasks spawned from its current context.

8. TAPAS currently doesn’t support Cilk Reducers. Reducers address the problem of computing a value by incrementally updating a variable in parallel code. They allow a variable to be safely used by multiple strands running in parallel.
5.3 Future Work

In addition to rectifying the previously listed limitations, the following areas could be avenues of future exploration.

**Synchronization Primitives:** Tapir makes minimal assumptions about memory concurrency and neither Tapir nor Cilk itself provides any explicit synchronization primitives. They rely on the runtime system to provide such ordering. It is an open question as to what primitives may be of value on an SoC system, and how to implement a consistent interface across multiple types distributed of memories, including complex memory systems shared with a CPU (e.g. accessed over AXI bus).

**Vector Operations:** The asynchronous handshaking interface used by TAPAS between dataflow nodes allows tolerance of variable latency operations and improves flexibility. However, it can also reduce a design’s performance due to additional overhead. This is especially evident in tight loops where delays in the loops increment logic can put an upper bound on its maximum iteration rate. By supporting vector and reduction nodes directly in the Chisel Library, TAPAS could leverage LLVM’s automatic vectorization passes which can also detect some reduction variables. This could significantly improve performance, in particular with filter type operations.

**Library Functions:** The Task Controller blocks (Figure 3.4) in TAPAS are currently only used to support spawning parallel tasks. However, they could also be used to support the sharing of commonly used functions between tasks. For example, floating point division is resource intensive for hardware but also pipelined and so can support multiple concurrent calculations. By using a Task Controller block to manage a complex shared node like a division, multiple different tasks could share the same divider increasing its utilization.
Bibliography


