Appears in the Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011

Dynamic Fine-Grain Scheduling of Pipeline Parallelism

Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, Christos Kozyrakis
Pervasive Parallelism Laboratory, Stanford University
{sanchezd, davidlo, rmyoo}@stanford.edu, yoel@cs.stanford.edu, christos@ee.stanford.edu

***Abstract*—Scheduling pipeline-parallel programs, defined as a graph of stages that communicate explicitly through queues, is challenging. When the application is regular and the underlying architecture can guarantee predictable execution times, several techniques exist to compute highly optimized static schedules. However, these schedules do not admit run-time load balancing, so variability introduced by the application or the underlying hardware causes load imbalance, hindering performance. On the other hand, existing schemes for dynamic fine-grain load balancing (such as task-stealing) do not work well on pipeline-parallel programs: they cannot guarantee memory footprint bounds, and do not adequately schedule complex graphs or graphs with ordered queues.**

We present a scheduler implementation for pipeline-parallel programs that performs fine-grain dynamic load balancing efficiently. Specifically, we implement the first real runtime for GRAMPS, a recently proposed programming model that focuses on supporting irregular pipeline and data-parallel applications (in contrast to classical stream programming models and schedulers, which require programs to be regular). Task-stealing with per-stage queues and queuing policies, coupled with a backpressure mechanism, allow us to maintain strict footprint bounds, and a buffer management scheme based on packet-stealing allows low-overhead and locality-aware dynamic allocation of queue data.

We evaluate our runtime on a multi-core SMP and find that it provides low-overhead scheduling of irregular workloads while maintaining locality. We also show that the GRAMPS scheduler outperforms several other commonly used scheduling approaches. Specifically, while a typical task-stealing scheduler performs on par with GRAMPS on simple graphs, it does significantly worse on complex ones; a canonical GPGPU scheduler cannot exploit pipeline parallelism and suffers from large memory footprints; and a typical static, streaming scheduler achieves somewhat better locality, but suffers significant load imbalance on a general-purpose multi-core due to fine-grain architecture variability (e.g., cache misses and SMT).

I. Introduction

Multi-core chips are now prevalent and core counts are increasing in accordance with Moore's Law. This trend has created a renewed interest in high-level parallel programming models such as Cilk [13], TBB [21], CUDA [32], OpenCL [25], and StreamIt [38]. These models provide constructs to express parallelism and synchronization in a manageable way, and their runtimes take care of *resource management* and *scheduling*.

While there are many important dimensions in evaluating a programming model —syntax, toolchain, ease of use, etc.— in this paper we focus on the *scheduling approaches* of different programming models. A scheduler should satisfy three desirable properties. First, it should keep the execution units well utilized, performing load balancing if needed. Second, it should guarantee bounds on the resources consumed. In particular, bounding memory footprint is especially important, as this enables allocating off-chip and on-chip memory resources, avoids thrashing and out-of-memory conditions, and reduces cache misses (in cache-based systems) or spills to main memory (in systems with explicitly managed scratchpads). Third, it should impose small scheduling overheads. The ability of the scheduler to realize these properties is constrained by the information available from the programming model. Consequently, schedulers are often tailored to a specific programming model.

We observe that most common scheduling approaches can be broadly grouped into three categories. First, *Task-Stealing* is popular in general-purpose multi-cores, and is used in Cilk, TBB, X10 [5], OpenMP [11], among others. It imposes small overheads, and some programming models, such as Cilk and X10, can bound memory footprint by tailoring its scheduling policies [1, 3]. However, it does not leverage program structure and does not work well when tasks have complex dependencies, so it has difficulties scheduling complex pipeline-parallel applications. Second, *Breadth-First* is used in GPGPU models such as CUDA and OpenCL, and focuses on extracting data parallelism, but cannot exploit task and pipeline parallelism and does not bound memory footprint. Third, *Static* is common in streaming architectures and stream programming models like StreamIt and StreamC/KernelC [10]. It relies on a priori knowledge of the application graph to statically generate an optimized schedule that uses bounded memory footprint [28]. Unfortunately, Static schedulers forgo run-time load balancing, and work poorly when the application is irregular or the architecture has dynamic variability, thus limiting their utility.

In contrast to these models, *GRAMPS* [36] is a programming model designed to support dynamic scheduling of pipeline and data parallelism. Similar to streaming models, GRAMPS applications are expressed as a graph of stages that communicate either explicitly through data queues or implicitly through memory buffers. However, GRAMPS introduces several enhancements that allow dynamic scheduling and applications with irregular parallelism. Compared to Task-Stealing models, knowing the application graph gives two main benefits. First, the graph contains all the producer-consumer relationships, enabling improved locality. Second, memory footprint is easily bounded by limiting the size of queues and memory buffers. However, prior work [36] was based on an idealized simulator

Approach	Supports Shader	Producer-Consumer	Hierarchical Work	Adaptive Schedule	Examples
Task-Stealing	No	No	No	Yes	Cilk, TBB, OpenMP
Breadth-First	Yes	No	Yes	No	CUDA, OpenCL
Static	Yes	Yes	Yes	No	StreamIt, Imagine
GRAMPS	**Yes**	**Yes**	**Yes**	**Yes**	GRAMPS

TABLE I: Comparison of different scheduling approaches.

with no scheduling overheads, making it an open question whether a practical GRAMPS runtime could be designed.

In this paper, we present the first real implementation of a GRAMPS runtime on multi-core machines. The scheduler introduces two novel techniques. First, *task-stealing with per-stage queues* and a *queue backpressure* mechanism enable dynamic load balancing while maintaining bounded footprint. Second, a *buffer management* technique based on *packet-stealing* enables dynamic allocation of data packets at low overhead, while maintaining good locality even in the face of frequent producer-consumer communication. To our knowledge, this is the first runtime that supports dynamic fine-grain scheduling of irregular streaming applications. While our runtime is specific to GRAMPS, the techniques used can be applied to other streaming programming models, such as StreamIt. We evaluate this runtime on a variety of benchmarks using a modern multi-core machine, and find that it efficiently schedules both simple and complex application graphs while preserving locality and bounded footprint.

Additionally, since the GRAMPS programming model provides a superset of the constructs of other models, the runtime can work with the other families of schedulers. We implement these schedulers and compare them using the same infrastructure, allowing us to focus on the differences between schedulers, not runtime implementations. We find that Task-Stealing is generally a good approach to schedule simple graphs, but becomes inefficient with complex graphs or ordered queues, and does not guarantee bounded footprint in general. Breadth-First scheduling is simple, but does not take advantage of pipeline parallelism and requires significantly more footprint than other approaches, putting more pressure on the memory subsystem. Finally, Static scheduling provides somewhat better locality than schedulers using dynamic load balancing due to carefully optimized, profile-based schedules. However, this benefit is negated by significant load imbalance, both from application irregularities and the dynamic nature of the underlying hardware. We show that our proposed GRAMPS scheduler achieves significant benefits over each of the other approaches.

II. Overview of Scheduling Approaches

In this section, we give the necessary background and definitions for different scheduling approaches: Task-Stealing, Breadth-First, Static, and GRAMPS. Rather than comparing specific scheduler implementations, our objective is to distill the key scheduling policies of each and to compare them.

A. Scheduler Features

We use four main criteria to compare scheduling approaches:

- **Support for shaders**: The scheduler supports a built-in construct for *data-parallel* work, which is automatically parallelized by the scheduler across independent lightweight instances.
- **Support for producer-consumer**: The scheduler is aware of data produced as intermediate results (i.e., created and consumed during execution) and attempts to exploit this during scheduling.
- **Hierarchical work**: The scheduler supports work being expressed and grouped at different granularities rather than all being expressed at the finest granularity.
- **Adaptive schedule**: The scheduler has freedom to choose what work to execute at run-time, and can choose from all available work to execute.

Using the above criteria, we discuss the four scheduling approaches considered. Table I summarizes the differences between scheduling approaches.

B. Previous Scheduling Approaches

Task-Stealing: A Task-Stealing scheduler sees an application as a set of explicitly divided, concurrent and independent tasks. These tasks are scheduled on worker threads, where each worker thread has a queue of ready tasks to which it enqueues and dequeues tasks. When a worker runs out of tasks, it tries to steal tasks from other workers.

Task-Stealing has been shown to impose low overheads and scale better than alternative task pool organizations [18]. Therefore, it is used by a variety of parallel programming models, such as Cilk [13], X10 [5], TBB [21], OpenMP [11], and Galois [27].

Task-Stealing is rooted in programming models that exploit fine-grain parallelism, and often focus on low-overhead task creation and execution [2]. As a result, they tend to lack features that add overhead, such as task priorities. All tasks appear to be equivalent to the scheduler, preventing it from exploiting producer-consumer relationships. Task-Stealing has several algorithmic options that provide some control over scheduling, such as the order in which tasks are enqueued, dequeued and stolen (e.g. FIFO or LIFO) or the choice of queues to steal from (e.g. randomized or nearest neighbor victims). Several programming models, such as Cilk and X10, focus on fork-join task parallelism. In these cases, LIFO enqueues and dequeues with FIFO steals from random victims is the most used policy, as it achieves near-optimal performance and guarantees that footprint grows at most linearly with the number of threads [3]. However, several studies have shown that there is no single best scheduling policy in the general case [11, 16, 18]. In fact, Galois, which targets irregular data-parallel applications that

are often sensitive to the scheduling policy, exposes a varied set of policies for task grouping and ordering, and enables the programmer to control the scheduling policy [27, 30].

In this work we leverage Task-Stealing as an efficient load balancing mechanism, combining it with additional techniques to schedule pipeline-parallel applications efficiently.

Breadth-First: In Breadth-First scheduling, the application is specified as a sequence of *data-parallel stages* or *kernels*. A stage is written in an implicitly parallel style that defines the work to be performed per input element. Stages are executed one at a time, and the scheduler automatically instances and manages a collection of shaders that execute the stage, with an implicit barrier between stages.

This model is conceptually very simple, but has weaknesses in extracting parallelism and constraining data footprint. If an application has limited parallelism per stage but many independent stages, the system will be under-utilized. Furthermore, even if a stage produces results at the same rate as the next stage that will consume them, the explicit barrier leaves no alternative but to spend memory space and bandwidth to spill the entire intermediate output of the first stage and to read it back during the second stage.

GPGPU programming models (e.g., CUDA [32] and OpenCL [25]) rely on a GPU's high bandwidth and large execution context count to implement Breadth-First schedulers. However, such assumptions could be problematic for a general-purpose multi-core machine.

Static: In this scheduling approach, an application is expressed as a graph of *stages*, which communicate explicitly via data *streams*. The scheduler uses static analysis, profiling, and/or user annotations to derive the execution time of each stage and the communication requirements across stages. Using this knowledge, it schedules stages across execution contexts in a pattern optimized for low inter-core communication and small memory footprints [15, 22, 26, 33]. Scheduling is done offline, typically by the compiler, eliminating run-time scheduling overheads.

Static schedulers take advantage of producer-consumer locality by scheduling producers and consumers in the same or adjacent cores. They work well when all stages are regular, but cannot adapt to irregular or data-dependent applications.

This scheduling approach is representative of StreamIt [38] and streaming architectures [9, 24], where it is assumed that a program has full control of the machine. However, it can suffer load imbalance in general-purpose multi-cores where resources (e.g., cores or memory bandwidth available to the application) can vary at run-time, as we will see in Section V.

C. GRAMPS

We now discuss the core concepts of the GRAMPS programming model that are relevant to scheduling. However, GRAMPS is expressive enough to describe a wide variety of computations. A full description of all constructs supported by the GRAMPS programming model and its detailed API can be found in [35].

GRAMPS applications are structured as graphs of application-defined *stages* with producer-consumer communication between stages through *data queues*. Application graphs may be pipelines, but cycles are also allowed. Figure 1 shows an example application graph.

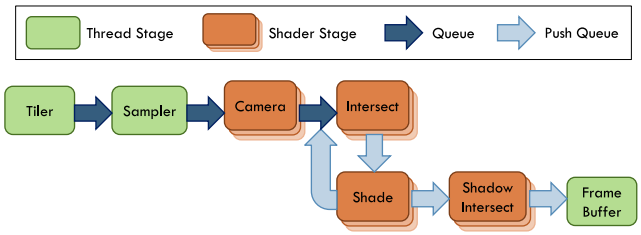

Fig. 1: A GRAMPS application: the raytracing graph from [36].

The GRAMPS programming model defines two types of stages: *Shaders* and *Threads* (there are also Fixed-function stages, but they are effectively Thread stages implemented in hardware and not relevant to this paper). Shader stages are stateless and automatically instanced by the scheduler. They are an efficient mechanism to express data-parallel regions of an application. Thread stages are stateful, and must be manually instanced by the application. Thread stages are typically used to implement task-parallel, serial, and other regions of an application characterized by (1) large per-element working sets or (2) operations dependent on multiple elements at once (e.g., reductions or re-sorting of data).

Stages operate upon data queues in units of *packets*, which expose bundles of grouped work over which queue operations and runtime decisions can be amortized. The application specifies the capacity of each queue in terms of packets and whether GRAMPS must maintain its contents in strict FIFO order. Applications can also use *buffers* to communicate between stages. Buffers are statically sized random-access memory regions that are well suited to store input datasets and final results.

There are three basic operations on queues: `reserve`, `commit`, and `push`. `reserve` and `commit` claim space in a queue and notify the runtime when the stage is done with it (either input was consumed or output was produced). Thread stages explicitly use `reserve` and `commit`. For Shader stages, GRAMPS implicitly `reserve`s packets before running a shader and `commit`s them when it finishes. `push` provides support for shaders with variable output. Shaders can `push` elements to a queue instead of whole packets. These elements are buffered and coalesced into full packets by the runtime, which then enqueues them. For example, in Figure 1 the Shadow Intersect stage operates on 32-ray input packets using SIMD operations, but the Shade stage produces a variable number of output rays. Push queues allow full 32-ray packets to be formed, maintaining the efficiency of SIMD operations.

Queue sets provide a mechanism to enable parallel consumption with synchronization: packets are consumed in sequence within each *subqueue*, but different subqueues may be processed in parallel. Consider a renderer updating its final output image: with unconstrained parallelism, instances cannot safely modify pixel values without synchronization. If the image is divided into disjoint tiles and updates are grouped by tile, then

tiles can be updated in parallel. In Figure 1, by replacing the input queue to the frame buffer stage with a queue set, the stage can be replaced with an instanced Thread stage (with one instance per subqueue) to exploit parallelism.

A GRAMPS scheduler should dynamically multiplex Thread and Shader stage instances onto available hardware contexts. The application graph can be leveraged to (1) reduce footprint by giving higher priority to downstream stages, so that the execution is geared towards pulling the data out of the pipeline, (2) bound footprint strictly by enforcing queue sizes, and (3) exploit producer-consumer locality by co-scheduling producers and consumers as much as possible.

At a high level, GRAMPS and streaming programming models have similar goals: both attempt to minimize footprint, exploit producer-consumer locality, and load-balance effectively across stages. However, GRAMPS achieves these goals via dynamic scheduling at run-time, while Static scheduling performs it offline at compile-time. GRAMPS also dynamically emulates filter fusion and fission [15] by co-locating producers and consumers on the same execution context and by time-multiplexing stages. Most importantly, Static schedulers rely on regular stage execution times and input/output rates to derive the long-running *steady state* of an application, which they can then schedule [28]. In contrast, GRAMPS does not require applications to have a steady state, allowing dynamic or irregular communication and execution patterns. For instance, a thread stage can issue an impossibly large `reserve`, which will be satisfied only when all the upstream stages have finished, thus effectively forming a barrier.

III. GRAMPS Runtime Implementation

Task-based schedulers, such as Task-Stealing, can perform load balancing efficiently because they represent work in compact tasks, so the cost of moving a task between cores is significantly smaller than the time it takes to execute it. However, these schedulers do not include support for data queues. On the other hand, in Static streaming schedulers worker threads simply run through a pre-built schedule, and have no explicit notion of a task. Work is implicitly encoded in the availability of data in each worker's fixed-size input buffers. Since *scheduling and buffer management are so fundamentally bound*, fine-grain load balancing on streaming runtimes is unachievable.

To achieve fine-grain load balancing and bounded footprint, the GRAMPS runtime *decouples scheduling and buffer management*. The runtime is organized around two entities:

- A *scheduler* that tracks runnable tasks and decides what to run on each thread context.
- A *buffer manager* that allocates and releases packets. It is essentially a specialized memory allocator for packets.

A. Scheduler Design

The GRAMPS scheduler is task-based: at initialization, the scheduler creates a number of *worker threads* using PThread facilities. Each of these threads has *task queues* with priorities, to which it enqueues newly produced tasks and from which it dequeues tasks to be executed. As in regular task-stealing, when a worker runs out of tasks, it tries to obtain more by stealing tasks from other threads. Worker threads leverage the application graph to determine the order in which to execute and steal tasks. All these operations are performed in a scalable but globally coordinated fashion.

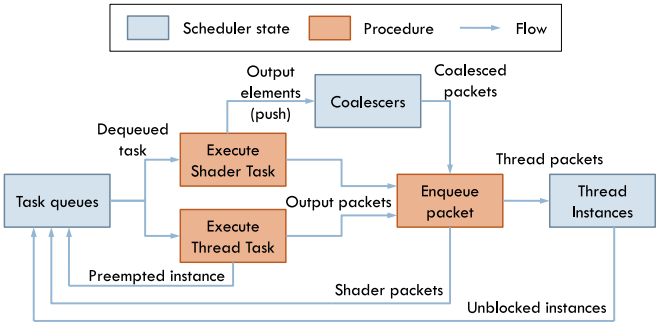

Fig. 2: Scheduler organization and task/packet flows.

Figure 2 shows an overview of the scheduler organization. We begin by describing how different kinds of stages are represented and executed in the runtime. We then describe the scheduling algorithms in detail.

Shader Stages: Shader stages are stateless and data-parallel, and a Shader can always be run given a packet from its input queue. Therefore, every time an input packet for a Shader stage is produced, a new Shader task with a pointer to that packet is generated and enqueued in the task queue. Shaders cannot block, so they are executed non-preemptively, avoiding context storage and switching overheads.

Shaders have two types of output queues: *packet queues*, to which they produce full packets, and *push queues*, to which they enqueue element by element. Executing a Shader with no push queues is straightforward: first, the output packets (or packet) are allocated by the buffer manager. The Shader task is then executed, its input packet is released to the buffer manager, and the generated packets are made available, possibly generating additional tasks. Shaders with push queues are treated slightly differently: each worker thread has a *coalescer*, which aggregates the elements enqueued by several instances of the Shader into full packets, and makes the resulting packets available to the scheduler.

The GRAMPS runtime and API are designed to avoid data copying: application code has direct access to input packets, and writes to output packets directly, even with push queues.

Thread Stages: Thread stages are stateful, long-lived, and may have queue ordering requirements. In particular, Thread stages operate on input and output queues explicitly, `reserv`ing and `commit`ting packets from them. Thus, they need to be executed preemptively: they can either block when they try to `reserve` more packets than are available in one of their input queues, or the scheduler can decide to preempt them at any `reserve` or `commit` operation.

To facilitate low-cost preemption, each Thread stage instance is essentially implemented as an *user-level thread*. We encapsulate all its state (stack and context) and all its input queues into

an *instance* object. Input queues store packets in FIFO order.

Each task for a Thread stage represents a *runnable* instance, and is simply a pointer to the instance object. At initialization, an instance is always runnable. A Thread stage instance that is preempted by the runtime before it blocks is still runnable, so after preemption we re-enqueue the task. To amortize preemption overheads, the runtime allows an instance to produce a fixed number of output packets (currently 32) before it preempts it. However, when an instance blocks, it is not runnable, so no task is generated. Instead, when enough packets are enqueued to the input queue that the instance blocked on, the runtime unblocks the instance and produces a task for it.

Note that by organizing scheduler state around tasks, coalescers, and instance objects, *only data queues that feed Thread stages actually exist as queues*. Queues that feed Shader stages are a useful abstraction for the programming model, but they do not exist in the implementation: each packet in these logical queues is physically stored as a task in the task queues. This is important, because many applications perform most of the work in Shaders, so this organization bypasses practically all the overheads of having data queues.

Per-Stage Task Queues: As mentioned in Section II, GRAMPS gives higher priority to stages further down the pipeline. Executing higher priority stages drains the pipeline, reducing memory footprint.

To implement per-stage priorities, each worker thread maintains a set of task queues, one per stage. For regular dequeues, a task is dequeued from higher priority stages first. When stealing tasks, however, the GRAMPS scheduler steals from lower priority stages first, which produce more work per invocation: these stages are at a lower depth in the application graph, so they will generate additional tasks enqueued to their consumer stages, which is desirable when threads are running out of tasks. For example, in Figure 1, a Camera task will generate more work than a Shadow Intersect task. To keep task queue operation costs low on graphs with many stages, worker threads store pointers to the highest and lowest queues that have tasks. The range of queues with tasks is small (typically 1 to 3), so task dequeues are fast.

We implement each task queue as a Chase-Lev deque [6], with LIFO local enqueues/dequeues and FIFO steals. Local operations do not require atomic instructions, and steals require a single atomic operation. Stealing uses the non-blocking ABP protocol [2] and is locality-aware, as threads try to steal tasks from neighboring cores first.

Footprint and Backpressure: In task-based parallel programming models, it is generally desirable to guarantee that footprint is bounded regardless of the number of worker threads. Programming models that exploit fork-join parallelism, like Cilk and X10, can guarantee that footprint grows at most linearly with the number of threads by controlling the queuing and stealing policies [1, 3]. However, most pipeline-parallel and streaming programming models cannot limit footprint as easily. In particular, GRAMPS applications can experience unbounded memory footprint in three cases:

1) Bottlenecks on Thread consumers: If producers produce packets faster than a downstream Thread stage can consume them, an unbounded number of packets can be generated.
2) Thread preemption policy: In order to amortize preemption overheads, GRAMPS does not immediately preempt Thread stages. However, without immediate preemption, footprints can grow superlinearly with the number of worker threads due to stealing [3].
3) Cycles: A graph cycle that generates more output than input will consume unbounded memory regardless of the scheduling strategy.

To solve issues (1) and (2), the runtime enforces bounded queue sizes by applying *backpressure*. The runtime tracks the utilization of each data queue in a scalable fashion, and when a queue becomes full, the stages that output to that queue are marked as non-executable. This guarantees that both data queue and task queue footprints are bounded.

In general, guaranteeing bounded footprint and deadlock-freedom with cycles is not a trivial task, and Static scheduling algorithms have significant problems handling cycles [26, 33]. To address this issue, we do not enforce backpressure on *backward* queues (i.e., queues that feed a lower-priority stage from a higher-priority stage). If the programmer introduces cycles with uncontrolled loops, we argue that this is an incorrect GRAMPS application, similar to reasoning that a programmer will not introduce infinite recursion in a Cilk program. If a cycle is well behaved, it is trivial to see that footprint is bounded due to stage priorities, even with stealing.

Overall, we find that backpressure strictly limits the worst-case footprint of applications while adding minimal overheads. We will evaluate the effectiveness of this approach in Section V.

Ordered Data Queues: Except for push queues, GRAMPS queues can be FIFO-ordered, which guarantees that the consuming stage of a queue will receive packets in the same order as if the producing stage was run serially. FIFO ordering on queues between Thread stages is maintained by default, but queues with Shader inputs or outputs are not automatically ordered since Shader tasks can execute out of order. Guaranteeing queue ordering allows GRAMPS to implement ordering-dependent streaming applications, but a naïve implementation could cause significant overheads.

We leverage the fact that guaranteeing ordering across two Thread stages is sufficient to guarantee overall ordering. Specifically, to maintain ordering on a chain of stages with Thread stage endpoints and Shaders in the middle, the runtime allocates and enqueues the output packet of the leading Thread stage and the corresponding input packet of the last stage atomically. The pointer to that last packet is then propagated through the intermediate Shader packets. While Shaders can execute out of order, at the end of the chain the packet is filled in and made available to the Thread consumer in order. Essentially, the last stage's input queue acts as a reorder buffer.

This approach minimizes queue manipulation overheads, but it may increase footprint since packets are pre-reserved and, more importantly, the last queue is subject to head-of-line

blocking. To address this issue, task queues of intermediate Shader stages follow FIFO ordering instead of the usual LIFO.

B. Buffer Manager Design

To decouple scheduling and buffer management, we must engineer an efficient way for the scheduler to dynamically allocate new packets and release used ones, while preserving locality. This is the function of the *buffer manager*.

The simplest possible buffer manager, if the system supports dynamic memory allocation, is to allocate packets using malloc and release them using free. We call this a *dynamic memory buffer manager*. However, this approach has high overheads: as more threads stress the memory allocator, synchronization and bookkeeping overheads can quickly dominate execution time.

A better buffer management strategy without dynamic memory overheads is to use per-queue, statically sized memory pools, and allocate packets from the corresponding queue. Since queue space is bounded in GRAMPS, we can preallocate enough buffer space per queue, and avoid malloc/free calls and overhead completely. We refer to this approach as *per-queue buffer manager*. However, this approach may hurt locality, as different threads share the same buffer space. To maximize locality, we would like a packet to be reused by the same core as much as possible. Nevertheless, it would be inefficient to just partition each per-queue pool among worker threads, since the buffer space demands of a worker are not known in advance, and often change throughout execution. Additionally, accessing the shared queue pools can incur synchronization overheads.

Instead, we propose a specialized *packet-stealing buffer manager* that *maximizes locality* while maintaining *low overheads*. In this scheme, queues get their packets from a set of pools, where each pool contains all the packets of the same size. Initially, within each pool, packets are evenly divided across worker threads; each allocation tries to dequeue a packet from the corresponding partition. However, if a worker thread finds its partition empty, it resorts to stealing to acquire additional packets. When done using a packet, a worker thread enqueues the packet back to its partition. To amortize stealing overheads, threads steal multiple packets at once.

Since backpressure limits queue size, stealing is guaranteed to succeed, except when cycles are involved. For cycles, the worker thread tries one round of stealing, and if it fails, it allocates and adds new packets to the pool using malloc. In practice, this does not happen in applications with correctly sized loop queues, and is just a deadlock avoidance safeguard.

Packet-stealing keeps overheads low, and more importantly, enables high reuse. For example, in applications with linear pipelines, the LIFO policy will cause each worker thread to use only two packets to traverse the pipeline. As a result, packet stealing only happens as frequently as stealing in the task queues takes place, which is rare for balanced applications.

IV. Other scheduling approaches

As mentioned in Section I, we augment our GRAMPS implementation to serve as a testbed for comparing other scheduling approaches. Specifically, we have defined a modular scheduler interface that enables using different schedulers.

We preserve the core GRAMPS abstractions and APIs —data queues, application graphs, etc.— for all schedulers, even for those used in programming models that customarily lack built-in support for them. This isolates the changes derived from scheduling policies from any distortion caused by changing the programming model. Additionally, it eliminates application implementation variation, as the same version of each application is used in all four modes.

The rest of this section describes the specific designs we chose to represent their respective scheduling approaches. While several variations are possible for a given scheduler type, we strive to capture the key philosophy behind each scheduler, while leaving out implementation-specific design choices.

A. Task-Stealing Scheduler

Our Task-Stealing scheduler mimics the widely used Cilk 5 scheduler [13]. It differs from the GRAMPS scheduler in three key aspects. First, each worker thread has a single LIFO Chase-Lev task queue [6]; i.e., there are no per-stage queues. Steals are done from the tail of the queue. Second, data queues are unbounded (without per-stage task queues, we cannot enforce backpressure). Third, each thread stage is preempted as soon as it commits a single output packet. This emulates Cilk's *work-first* policy, which switches to a child task as soon as it is created. This policy enables Cilk to guarantee footprint bounds [13]. Although this does not imply bounded buffer space for GRAMPS, the work-first approach works well in limiting footprint except when there is contention on thread consumers, as we will see in Section V.

B. Breadth-First Scheduler

The Breadth-First scheduler executes one stage at a time in breadth-first order, so a stage is run only when all its producers have finished. It is the simplest of our schedulers, and represents a typical scheduler for GPGPU programming model (e.g., CUDA). All worker threads run the current stage until they are out of work, then advance in lock-step to the next stage. As usual, load balancing is implemented on each stage with Chase-Lev dequeues. Some of our applications have cycles in their graphs so the scheduler will reset to the top of the graph a finite number of times if necessary. As with Task-Stealing, Breadth-First has no backpressure.

C. Static Scheduler

The Static scheduler represents schedulers for stream programming models [10, 38]. To generate a static schedule, the application is first profiled running under the GRAMPS scheduler with the desired number of worker threads. We then run METIS [23] to compute a graph partitioning that minimizes the communication to computation ratio, while balancing the computational load in each partition. We then feed the partitioning to the Static scheduler, which executes the application again following a minimum-latency schedule [22], assigning each partition to a hardware context.

Workload	Origin	Graph Complexity	Shader Work	Pipeline Parallel	Regularity
raytracer	GRAMPS	Medium	99%	Yes	Irregular
hist-r/c	MapReduce	Small	97%	No	Irregular
lr-r/c	MapReduce	Small	99%	No	Regular
pca	MapReduce	Small	99%	No	Regular
mergesort	Cilk	Medium	99%	Yes	Irregular
fm	StreamIt	Large	43%	Yes	Regular
tde	StreamIt	Huge	8%	Yes	Regular
fft2	StreamIt	Medium	70%	Yes	Regular
serpent	StreamIt	Medium	86%	Yes	Regular
srad	CUDA	Small	99%	No	Regular
rg	CUDA	Small	99%	No	Regular

TABLE II: Application characteristics. Pipeline parallelism denotes the existence of producer-consumer parallelism at the graph level.

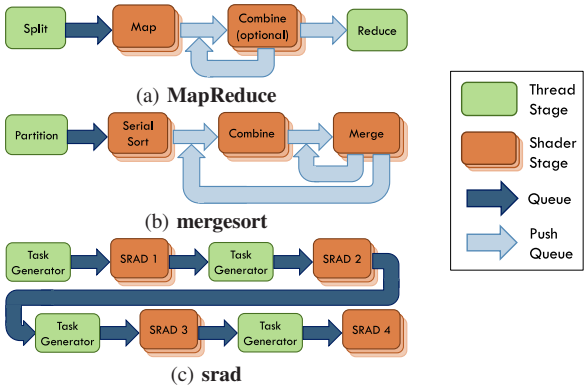

Fig. 3: Example application graphs.

The Static scheduler does no load balancing; instead, worker threads have producer-consumer queues that they use to send and receive work [15, 26, 33]. Once a thread runs out of work, it simply waits until it receives more work or the phase terminates. To handle barriers in programs with multiple phases, the Static scheduler produces one schedule per phase.

Overall, the Static scheduler trades off dynamic load balancing for better locality and lower communication.

V. Evaluation

A. Methodology

We perform all experiments on a 2-socket system with hexa-core 2.93 GHz Intel Xeon X5670 (Westmere) processors. With 2-way Simultaneous Multi-Threading (SMT), the system features a total of 12 cores and 24 hardware threads. The system has 256 KB per-core L2 caches, 12 MB per-processor L3 caches, and 48 GB of DDR3 1333 MHz memory (about 21 GB/s peak memory bandwidth). The processors communicate through a 6.4 GT/s QPI interconnect. This machine runs 64-bit GNU/Linux 2.6.35, with GCC 4.4.5. We ran all of our experiments several times and report average results; experiments were run until the 95% confidence intervals on each average became negligible ($< 1\%$).

Applications: In order to broadly exercise the schedulers, we have expanded the original set of GRAMPS applications [36] with a variety of examples from other programming models. Table II summarizes their qualitative characteristics, and Figure 3 shows a few representative graphs. The applications are:

- **raytracer** is the packetized ray tracer from [36]. We run it with no reflection bounces (ray-0), in which case it is a pipeline, and with one bounce (ray-1), in which case its graph has a cycle (as in Figure 1).
- **histogram, lr, pca** are MapReduce applications from the Phoenix suite [41]. We run **histogram** and **lr** in two forms: reduce-only (r) and with a combine stage (c).
- **mergesort** is a parallel mergesort implementation using Cilk-like spawn-sync parallelism. Its graph, shown in Figure 3, contains two nested loops.
- **fm, tde, fft2, serpent** are streaming benchmarks from the StreamIt suite [38]. All have significant pipeline parallelism and use ordered queues. **fm** and **tde** have very large graphs, with a small amount of data parallelism, while **fft2** and **serpent** have smaller graphs and more data parallelism.
- **srad, rg** are data-parallel CUDA applications. **srad** (Speckle Reducing Anisotropic Diffusion, an image-processing benchmark) was ported from the Rodinia suite [7], and **rg** (Recursive Gaussian) comes from the CUDA SDK [31].

B. GRAMPS Scheduler Performance

Figure 4 shows the speedups achieved by the GRAMPS scheduler from 1 to 24 threads. The knee that consistently occurs at 12 threads is where all the physical cores are used and the runtime starts using the second hardware thread per core (SMT). Overall, all applications scale well. With more cores, **srad** and **rg** scale sublinearly because they become increasingly bound by cache and memory bandwidth. They are designed for GPUs, which have significantly more bandwidth.

Figure 5 gives further insight into these results. It shows the execution time breakdown of each application when using all 24 hardware threads. In this section, we focus on the results with the GRAMPS scheduler (the leftmost bar for each app). Each bar is split into four categories, showing the fraction of time spent in application code, scheduler code, buffer manager, and stalled (which in the GRAMPS scheduler means stealing, with no work to execute). This breakdown is obtained with low-overhead profiling code that uses the CPU timestamp counter, which adds $\leq 2\%$ to the execution time.

Overall, results show that GRAMPS is effective at finding and dynamically distributing parallelism: applications spend minimal time without work to do. Furthermore, runtime overheads are small: the majority of workloads spend less than 2% of the time in the scheduler and buffer manager. Even tracking the tens of stages in **fm** takes only 13% of elapsed time in scheduling overheads. The worst-case buffer manager overhead happens in queue-intensive **fft2**, at 15%.

Finally, Table III shows the average and maximum footprints of the GRAMPS scheduler. Footprints are reasonable, and always below the maximum queue sizes, due to backpressure providing strict footprint bounds.

C. Comparison of Scheduler Alternatives

We now evaluate the differences among the scheduling alternatives discussed in Section II. Figure 4, Figure 5, and

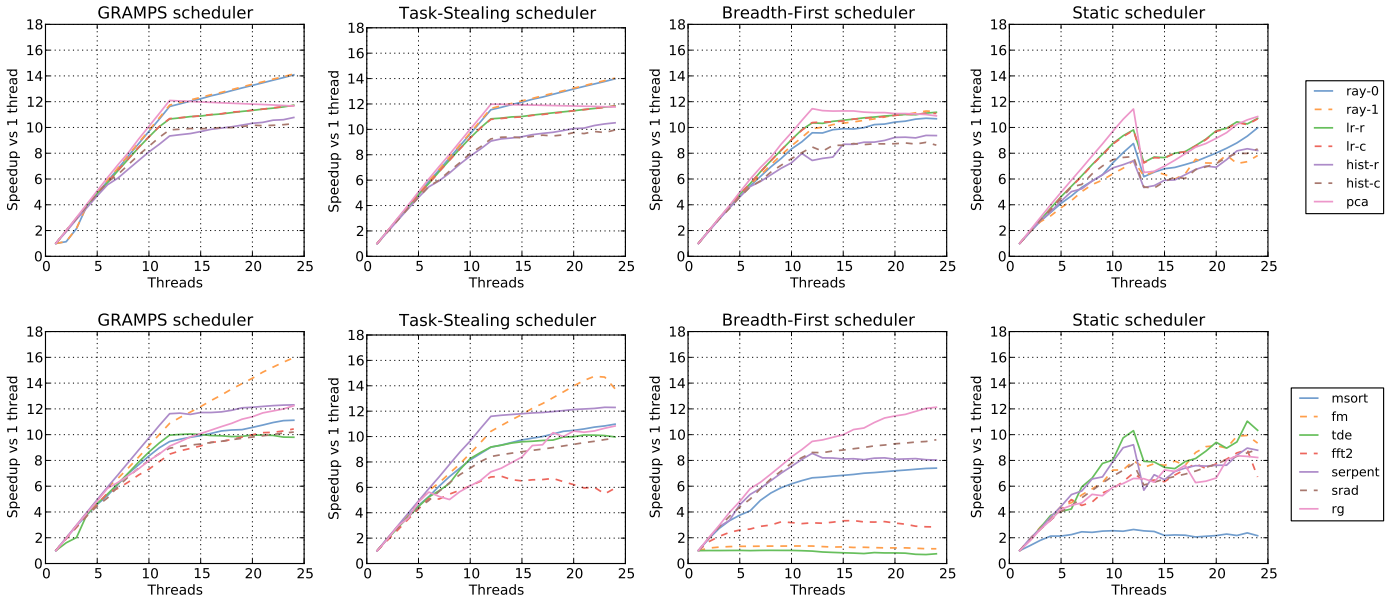

Fig. 4: Scalability of GRAMPS, Task-Stealing, Breadth-First, and Static schedulers from 1 to 24 threads (12 cores).

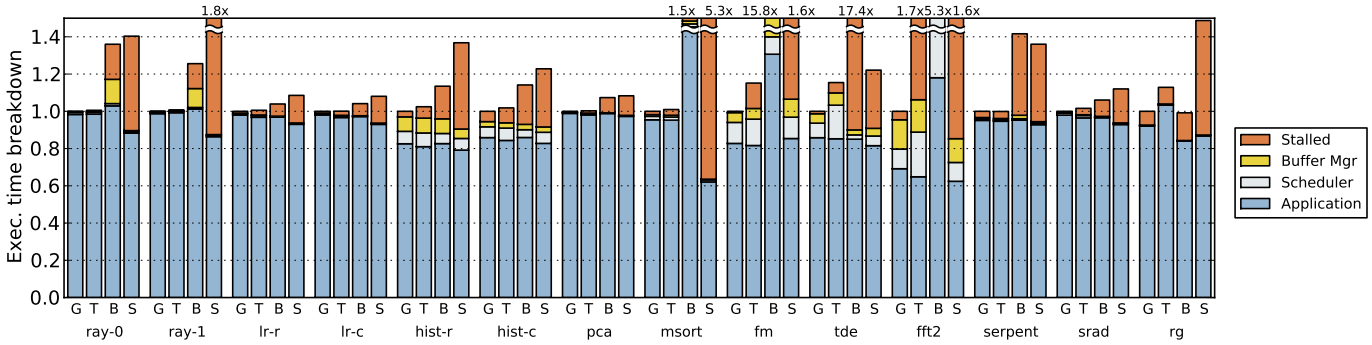

Fig. 5: Runtime breakdown of GRAMPS (G), Task-Stealing (T), Breadth-First (B), and Static schedulers (S) at 24 threads. Each bar shows the execution time breakdown. Results are normalized to the runtime with the GRAMPS scheduler.

Table III show the scalability, execution time breakdowns, and footprints for the different schedulers, respectively.

Task-Stealing: For applications with simple graphs, the Task-Stealing scheduler achieves performance and footprint results similar to the GRAMPS scheduler. However, it *struggles on applications with complex graphs*: **fm** and **tde**, and applications with ordering, **fft2**.

fm and **tde** have the most complex graphs, with 121 and 412 stages respectively (Table II), and have abundant pipeline parallelism but little data parallelism, and ordered queues. In both **fm** and **tde**, Task-Stealing is unable to keep the system fully utilized, as evidenced by the larger Stalled application times, due to the simple LIFO policy and lack of backpressure, which also cause larger footprints and overheads. In **fft2**, scalability is limited by the last stage in the pipeline, a Thread consumer. Since this workload has ordering requirements, the LIFO task ordering causes significant head-of-line blocking; packets are released in bursts, which causes this stage to bottleneck sooner. This bottleneck shows as the large Stalled part of the execution breakdown in **fft2** under Task-Stealing. In contrast, with graph knowledge, GRAMPS uses FIFO task queuing on ordered stages (Section III). Hence packets arrive almost ordered, and the receiving stage does not bottleneck.

Breadth-First: The Breadth-First scheduler cannot take advantage of pipeline parallelism. Consequently, it *matches the GRAMPS scheduler only on those applications without pipeline parallelism*, **srad** and **rg**. In other applications, the one-stage-at-a-time approach significantly affects performance and footprint.

Performance is most affected in **fm** and **tde**, which are highly pipeline-parallel but not data-parallel (Table II). Looking at the execution breakdown for those two applications, we see that Breadth-First scheduling leaves the system starved for work for up to 95% of the time. Compared to GRAMPS, the slowdowns are as high as 15.8x and 17.4x, respectively.

In terms of footprint, the large amount of intermediate results generated by each stage put high pressure on the memory system and the buffer manager. Footprint differences are most pronounced in **raytracer** and the StreamIt applications

App	GRAMPS		Task-Stealing		Breadth-First		Static	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
ray-0	215.9 KB	287.8 KB	191.7 KB	276.5 KB	23096.0 KB	51648.5 KB	322.7 KB	722.7 KB
ray-1	361.1 KB	447.4 KB	327.5 KB	424.6 KB	31257.1 KB	78706.5 KB	1133.5 KB	2271.0 KB
lr-r	22.9 KB	40.7 KB	22.9 KB	40.9 KB	24.0 KB	44.0 KB	23.7 KB	43.8 KB
lr-c	8.1 KB	8.8 KB	7.9 KB	8.2 KB	15.2 KB	30.2 KB	9.8 KB	12.8 KB
hist-r	4921.6 KB	9658.1 KB	4925.0 KB	9654.1 KB	4695.3 KB	9736.6 KB	4903.2 KB	9653.3 KB
hist-c	1860.3 KB	3096.8 KB	1864.5 KB	3096.3 KB	1503.9 KB	3092.7 KB	1854.7 KB	3098.9 KB
pca	0.7 KB	1.3 KB	0.4 KB	0.4 KB	89.2 KB	178.8 KB	4.6 KB	8.0 KB
msort	2.0 KB	7.0 KB	1.4 KB	4.6 KB	6.1 KB	10.4 KB	5.3 KB	18.0 KB
fm	1672.3 KB	4276.8 KB	2245.2 KB	3691.1 KB	165145.5 KB	563173.0 KB	29646.3 KB	57391.3 KB
tde	3989.0 KB	6662.2 KB	18398.7 KB	36231.2 KB	89843.5 KB	179282.0 KB	18378.9 KB	31071.5 KB
fft2	261.3 KB	376.0 KB	149.8 KB	211.0 KB	75624.4 KB	80096.0 KB	1183.7 KB	1395.0 KB
serpent	79.1 KB	88.0 KB	68.7 KB	73.2 KB	1031.4 KB	1048.0 KB	735.8 KB	1003.2 KB
srad	0.9 KB	1.6 KB	0.4 KB	0.5 KB	40.0 KB	80.0 KB	2.6 KB	8.2 KB
rg	0.7 KB	1.4 KB	0.4 KB	0.5 KB	1.9 KB	5.0 KB	0.6 KB	1.6 KB

TABLE III: Average and maximum footprints of different schedulers.

(Table III). For example, the **raytracer**'s worst-case footprint is 447 KB with GRAMPS, but 78.7 MB with Breadth-First. This turns into a larger buffer manager overhead, which takes 12% of the execution time (Figure 5). Larger footprint reduces the effectiveness of caches, hurting locality, as seen from the slightly higher application time.

Static: The Static scheduler trades off load balancing for near-optimal static work division and minimized producer-consumer communication. Although this is likely a good trade-off in embedded/streaming systems with fully static applications, we see that *it is a poor choice when either the system or the application is dynamic*.

Focusing on the scalability graph (Figure 4), we see that from 1 to 12 threads the static scheduler achieves reasonable speedups for highly regular applications: **pca**, **lr**, and **tde** get close to linear scaling. However, more irregular applications experience milder speedups, e.g., up to 7x for the **raytracer**. The worst-performing application is **mergesort**, which has highly irregular packet rates, and static partitioning fails to generate an efficient schedule.

As we move from 12 to 13 threads, performance drops in all applications. At this point, SMT starts being used in some cores, so threads run at different speeds. While the other schedulers easily handle this by performing load balancing, this fine-grain variability significantly hinders the Static scheduler. Interestingly, the execution time breakdown (Figure 5) shows that the static scheduler actually achieves lower application times, due to optimized locality. However, these improvements are more than negated by load imbalance, which increases the time spent waiting for work.

In summary, we see that GRAMPS and Task-Stealing achieve the best performance overall. However, Task-Stealing's simple LIFO queuing does not work well for complex application graphs or graphs with ordering, and cannot bound footprint due to lack of backpressure. While Breadth-First scheduling takes advantage of data parallelism, it cannot extract pipeline parallelism. Breadth-First scheduling also cannot take advantage of producer-consumer communication typical of pipeline-parallel

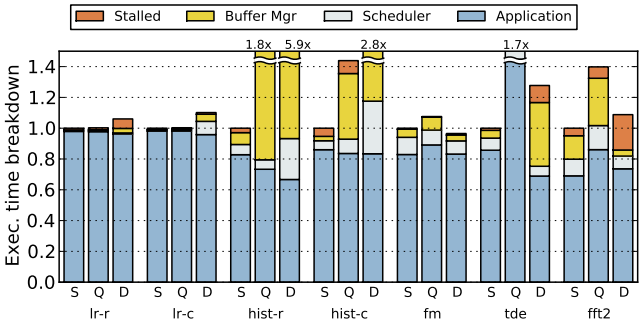

Fig. 6: Runtime breakdown with GRAMPS scheduler on queue-heavy applications, using the three alternative buffer managers: packet-stealing (S), per-queue (Q), and dynamic memory (D).

programs, causing memory footprint to be extremely large. Static scheduling is able to effectively schedule for locality, but cannot handle irregular applications or run-time variability in the underlying hardware. More importantly, we observe that GRAMPS' dynamic scheduling overheads are *mostly negligible* and locality is not significantly worse than what is achieved by locality-optimized Static schedules. This shows that, contrary to conventional wisdom, dynamic schedulers can efficiently schedule complex pipeline-parallel applications.

D. Comparison of Buffer Management Strategies

All the results previously shown have used the proposed packet-stealing buffer manager. We now evaluate the importance of this choice. Figure 6 compares the performance of the different buffer management approaches discussed in Section III, focusing on applications where the choice of buffer manager had an impact.

We observe that the packet-stealing approach achieves small overheads and good locality. In contrast, the dynamic buffer manager often has significant slowdowns, as frequent calls to malloc/free bottleneck at the memory allocator. We used tcmalloc as the memory allocator [14], which was the highest-performing allocator of those we tried (ptmalloc2, hoard, dlmal-

loc, and nedmalloc). Compared to the stealing buffer manager, the worst-case slowdown, 5.9x, happens in **histogram**, which is especially footprint-intensive.

The per-queue buffer manager shows lower overheads than the dynamic scheme. However, overheads can still be large (up to 80% in **histogram**), and more importantly, having per-queue slabs can significantly affect locality, as seen by the higher application times in several benchmarks. This is most obvious in **tde**, where the large number of global per-queue pools (due to the large number of stages) causes application time to increase by 50%. In contrast, the stealing allocator uses a reduced number of thread-local LIFO pools, achieving much better locality.

Overall, we conclude that buffer management is an important issue in GRAMPS applications, and the stealing buffer manager is a good match to GRAMPS in systems with cache hierarchies, achieving low overheads while maintaining locality.

VI. Related Work

We now discuss additional related work not covered in Section II.

Although many variants of Task-Stealing schedulers have been proposed, one of the most relevant aspects is whether the scheduler follows a work-first policy (moving depth-first as quickly as possible) or a help-first policy (producing several child tasks at once). Prior work has shown there are large differences between these approaches [16], and has proposed an adaptive work-first/help-first scheduler [17]. In GRAMPS, limiting the number of output packets produced by a Thread before it is preempted controls this policy. Our Task-Stealing scheduler follows the work-first policy, but we also tried the help-first policy, which did not generally improve performance, as the benefits of somewhat reduced preemption overheads were countered by higher memory footprints. The GRAMPS scheduler follows the help-first policy, preempting threads after several output packets, and leverages backpressure to keep footprint limited.

Some Task-Stealing models implement limited support for pipeline parallelism; TBB [21] supports simple 1:1 pipelines, and Navarro et al. [29] use this feature to study pipeline applications on CMPs, and provide an analytical model for pipeline parallelism. As we have shown, Task-Stealing alone does not work well for complex graphs.

Others have also observed that Breadth-First schedulers are poorly suited to pipeline-parallel applications. Horn et al. [20] find that a raytracing pipeline can benefit by bypassing the programming model and writing a single uber-kernel with dynamic branches. Tzeng et al. [39] also go against the programming model by using uber-kernels, and implement several load balancing strategies on GPUs to parallelize irregular pipelines. They find that task-stealing provides the least contention and highest performance. The techniques presented in this paper could be used to extend GRAMPS to GPUs.

Although most work in streaming scheduling is static, prior work has introduced some degree of coarse-grain dynamism. To handle irregular workloads, Chen et al. propose to pre-generate multiple schedules for possible input datasets and steady states, and switch schedules periodically [8]. Flextream [19] proposes online adaptation of offline-generated schedules for coarse-grain load balancing. These techniques could fix some of the maladies shown by Static scheduling on GRAMPS (e.g., SMT effects). However, applications with fine-grain irregularities, like **raytracer** or **mergesort**, would not benefit from this. Feedback-directed pipelining [37] proposes a coarse-grain hill-climbing partitioning strategy to maximize parallelism and power efficiency on pipelined loop-parallel code. While the power-saving techniques proposed could be used by GRAMPS, its coarse-grained nature faces similar limitations.

Finally, this work has focused on parallel programming models commonly used in either general-purpose multi-cores, GPUs and streaming architectures. We have not covered programming models that target clusters, MPPs or datacenter-scale deployments. These usually expose a multi-level memory organization to the programmer, who often has to manage memory and locality explicitly. Examples include MPI [34], PGAS-based models such as UPC [4] or Titanium [40], and more recent efforts like Sequoia [12].

VII. Conclusions

We have presented a scheduler for pipeline-parallel programs that performs fine-grain dynamic load balancing efficiently. Specifically, we implement the first real runtime for GRAMPS, a recently proposed programming model that focuses on supporting irregular pipeline-parallel applications. Our evaluation shows that the GRAMPS runtime achieves good scalability and low overheads on a 12-core, 24-thread machine, and that our scheduling and buffer management policies efficiently schedule simple and complex application graphs while preserving locality and bounded footprint.

Our scheduler comparison indicates that both Breadth-First and Static scheduling approaches are not broadly suitable on general purpose processors, and that GRAMPS and Task-Stealing behave similarly for simple application graphs. However, as graphs become more complex, GRAMPS shows an advantage in memory footprint and execution time because it is able to exploit knowledge of the application graph, which is unavailable to Task-Stealing.

Acknowledgements

We sincerely thank Asaf Cidon, Christina Delimitrou, and the anonymous reviewers for their useful feedback on the earlier versions of this manuscript. We also thank Jongsoo Park for providing his static scheduling framework, and Jithun Nair for porting some of the StreamIt workloads to GRAMPS. This work was supported in part by the Stanford Pervasive Parallelism Laboratory and the Gigascale Systems Research Center (FCRP/GSRC). Daniel Sanchez is supported by a Hewlett-Packard Stanford School of Engineering Fellowship, and Richard M. Yoo is supported by the David and Janet Chyan Stanford Graduate Fellowship.

REFERENCES

[1] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of X10 computations with bounded resources," in *Proc. of the 19th annual ACM Symp. on Parallel Algorithms and Architectures*, 2007.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proc. of the 10th annual ACM Symp. on Parallel Algorithms and Architectures*, 1998.

[3] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proc. of the 35th annual symp. on Foundations of Computer Science*, 1994.

[4] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Center for Computing Sciences, IDA, Tech. Rep., 1999.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proc. of the 20th annual ACM SIGPLAN conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[6] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proc. of the 17th annual ACM Symp. on Parallel Algorithms and Architectures*, 2005.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2009.

[8] J. Chen, M. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand, "A reconfigurable architecture for load-balanced rendering," in *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics Hardware*, 2005.

[9] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with streams," in *Proc. of the 17th annual intl. conf. on Supercomputing*, 2003.

[10] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proc. of the 15th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2006.

[11] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *4th Intl. Workshop in OpenMP*, 2008.

[12] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally *et al.*, "Sequoia: Programming the Memory Hierarchy," in *Proc. of the 2006 ACM/IEEE conf. on Supercomputing*, 2006.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 1998.

[14] S. Ghemawat and P. Menage, "TCMalloc : Thread-Caching Malloc http://goog-perftools.sourceforge.net/doc/tcmalloc.html."

[15] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. of the 12th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[16] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for terminally strict parallel programs," in *Proc. of the 23rd IEEE Intl. Parallel and Distributed Processing Symp.*, 2009.

[17] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *Proc. of the 15th ACM SIGPLAN symp. on Principles and Practice of Parallel Programming*, 2010.

[18] R. Hoffmann, M. Korch, and T. Rauber, "Performance evaluation of task pools based on hardware synchronization," in *Proc. of the 2004 ACM/IEEE conf. on Supercomputing*, 2004.

[19] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *Proc. of the 18th intl. conf. on Parallel Architectures and Compilation Techniques*, 2009.

[20] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-D Tree GPU Raytracing," in *Proc. Symp. on Interactive 3D Graphics and Games*, 2007.

[21] Intel, "TBB http://www.threadingbuildingblocks.org."

[22] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased scheduling of stream programs," in *Proc. of the ACM SIGPLAN conf. on Language, compiler, and tool for embedded systems*, 2003.

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, 1998.

[24] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: media processing with streams," *Micro, IEEE*, vol. 21, no. 2, 2001.

[25] Khronos Group, "OpenCL 1.0 specification," 2009.

[26] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 2008.

[27] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Scheduling strategies for optimistic parallel execution of irregular programs," in *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

[28] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, 1987.

[29] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *Proc. of the 18th intl. conf. on Parallel Architectures and Compilation Techniques*, 2009.

[30] D. Nguyen and K. Pingali, "Synthesizing concurrent schedulers for irregular algorithms," in *Proc. of the 16th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[31] NVIDIA, "CUDA SDK Code Samples http://developer.nvidia.com/object/cuda_sdk_samples.html."

[32] NVIDIA , "CUDA 3.0 reference manual," 2010.

[33] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[34] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference*. MIT Press, 1998.

[35] J. Sugerman, "Programming many-core systems with GRAMPS," *Ph.D. Thesis, Stanford University*, 2010.

[36] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, 2009.

[37] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010.

[38] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. of the 10th Intl. Conf. on Compiler Construction*, 2001.

[39] S. Tzeng, A. Patney, and J. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proc. of the conf. on High Performance Graphics*, 2010.

[40] K. Yelick *et al.*, "Titanium: A high-performance Java dialect," in *Proc. of the Workshop on Java for High-Performance Network Computing*, 1998.

[41] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2009.