Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs

Michael I. Gordon, William Thies, and Saman Amarasinghe

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
{mgordon, thies, saman}@mit.edu

Abstract

As multicore architectures enter the mainstream, there is a pressing demand for high-level programming models that can effectively map to them. Stream programming offers an attractive way to expose coarse-grained parallelism, as streaming applications (image, video, DSP, etc.) are naturally represented by independent filters that communicate over explicit data channels.

In this paper, we demonstrate an end-to-end stream compiler that attains robust multicore performance in the face of varying application characteristics. As benchmarks exhibit different amounts of task, data, and pipeline parallelism, we exploit all types of parallelism in a unified manner in order to achieve this generality. Our compiler, which maps from the StreamIt language to the 16-core Raw architecture, attains a 11.2x mean speedup over a single-core baseline, and a 1.84x speedup over our previous work.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications–Data-flow languages; D.3.3 [*Programming Languages*]: Language Constructs and Features–Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors–Compilers, Optimization

General Terms Design, Languages, Performance

Keywords coarse-grained dataflow, multicore, Raw, software pipelining, StreamIt, streams

1. Introduction

As centralized microprocessors are ceasing to scale effectively, multicore architectures are becoming the industry standard. For example, the IBM/Toshiba/Sony Cell processor has 9 cores [17], the Sun Niagara has 8 cores [21], the RMI XLR732 has 8 cores [1], the IBM/Microsoft Xbox 360 CPU has 3 cores [4], and most vendors are shipping dual-core chips. Cisco has described a next-generation network processor containing 192 Tensilica Xtensa cores [14]. This trend has pushed the performance burden to the compiler, as future application-level performance gains depend on effective parallelization across the cores. Unfortunately, traditional programming models such as C, C++ and FORTRAN are ill-suited to multicore architectures because they assume a single instruction stream and a monolithic memory. Extracting coarse-grained parallelism suitable for multicore execution amounts to a heroic compiler analysis that remains largely intractable.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

The stream programming paradigm offers a promising approach for exposing parallelism suitable for multicore architectures. Stream languages such as StreamIt [39], Brook [6], SPUR [42], Cg [27], Baker [9], and Spidle [10] are motivated not only by trends in computer architecture, but also by trends in the application space, as network, image, voice, and multimedia programs are becoming only more prevalent. In the StreamIt language, a program is represented as a set of autonomous actors that communicate through FIFO data channels (see Figure 1). During program execution, actors fire repeatedly in a periodic schedule. As each actor has a separate program counter and an independent address space, all dependences between actors are made explicit by the communication channels. Compilers can leverage this dependence information to orchestrate parallel execution.

Despite the abundance of parallelism in stream programs, it is nonetheless a challenging problem to obtain an efficient mapping to a multicore architecture. Often the gains from parallel execution can be overshadowed by the costs of communication and synchronization. In addition, not all parallelism has equal benefits, as there is sometimes a critical path that can only be reduced by running certain actors in parallel. Due to these concerns, it is critical to leverage the right combination of task, data, and pipeline parallelism while avoiding the hazards associated with each.

Task parallelism refers to pairs of actors that are on different parallel branches of the original stream graph, as written by the programmer. That is, the output of each actor never reaches the input of the other. In stream programs, task parallelism reflects logical parallelism in the underlying algorithm. It is easy to exploit by mapping each task to an independent processor and splitting or joining the data stream at the endpoints (see Figure 2b). The hazards associated with task parallelism are the communication and synchronization associated with the splits and joins. Also, as the granularity of task parallelism depends on the application (and the programmer), it is not sufficient as the only source of parallelism.

Data parallelism refers to any actor that has no dependences between one execution and the next. Such "stateless" actors[1] offer unlimited data parallelism, as different instances of the actor can be spread across any number of computation units (see Figure 2c). However, while data parallelism is well-suited to vector machines, on coarse-grained multicore architectures it can introduce excessive communication overhead. Previous data-parallel streaming architectures have focused on designing a special memory hierarchy to support this communication [18]. However, data parallelism has the hazard of increasing buffering and latency, and the limitation of being unable to parallelize actors with state.

Pipeline parallelism applies to chains of producers and consumers that are directly connected in the stream graph. In our previ-

[1] A stateless actor may still have read-only state.

ous work [15], we exploited pipeline parallelism by mapping clusters of producers and consumers to different cores and using an on-chip network for direct communication between actors (see Figure 2d). Compared to data parallelism, this approach offers reduced latency, reduced buffering, and good locality. It does not introduce any extraneous communication, and it provides the ability to execute any pair of stateful actors in parallel. However, this form of pipelining introduces extra synchronization, as producers and consumers must stay tightly coupled in their execution. In addition, effective load balancing is critical, as the throughput of the stream graph is equal to the minimum throughput across all of the processors.

In this paper, we describe a robust compiler system that leverages the right combination of task, data, and pipeline parallelism to achieve good multicore performance across a wide range of input programs. Because no single type of parallelism is a perfect fit for all situations, a unified approach is needed to obtain consistent results. Using the StreamIt language as our input and targeting the 16-core Raw architecture, our compiler demonstrates a mean speedup of 11.2x over a single-core baseline; 7 out of 12 benchmarks speedup by over 12x. This also represents a 1.84x improvement over our previous work [15].

As part of this effort, we have developed two new compiler techniques that are generally applicable to any coarse-grained multicore architecture. The first technique leverages data parallelism, but avoids the communication overhead by first increasing the granularity of the stream graph. Using a program analysis, we fuse actors in the graph as much as possible so long as the result is stateless. Each fused actor has a significantly higher computation to communication ratio, and thus incurs significantly reduced communication overhead in being duplicated across cores. To further reduce the communication costs, the technique also leverages task parallelism; for example, two balanced task-parallel actors need only be split across half of the cores in order to obtain high utilization. On Raw, coarse-grained data parallelism achieves a mean speedup of 9.9x over a single core and 4.4x over a task-parallel baseline.

The second technique leverages pipeline parallelism. However, to avoid the pitfall of synchronization, it employs software pipelining techniques to execute actors from different iterations in parallel. While software pipelining is traditionally applied at the instruction level, we leverage powerful properties of the stream programming model to apply the same technique at a coarse level of granularity. This effectively removes all dependences between actors scheduled in a steady-state iteration of the stream graph, greatly increasing the scheduling freedom. Like hardware-based pipelining, software pipelining allows stateful actors to execute in parallel. However, it avoids the synchronization overhead because processors are reading and writing into a buffer rather than directly communicating with another processor. On Raw, coarse-grained software pipelining achieves a 7.7x speedup over a single core and a 3.4x speedup over a task-parallel baseline.

Combining the techniques yields the most general results, as data parallelism offers good load balancing for stateless actors while software pipelining enables stateful actors to execute in parallel. Any task parallelism in the application is also naturally utilized, or judiciously collapsed during granularity adjustment. This integrated treatment of coarse-grained parallelism leads to an overall speedup of 11.2x over a single core and 5.0x over a task-parallel baseline.

2. The StreamIt Language

StreamIt is an architecture-independent programming language for high-performance streaming applications [39, 2]. As described previously, it represents programs as a set of independent actors (referred to as *filters* in StreamIt) that use explicit data channels for all communication. Each filter contains a `work` function that executes a single step of the filter. From within `work`, filters can *push* items onto the output channel, *pop* items from the input channel, or *peek* at an input item without removing it from the channel. While peeking requires special care in parts of our analysis, it is critical for exposing data parallelism in sliding-window filters (e.g., FIR filters), as they would otherwise need internal state.

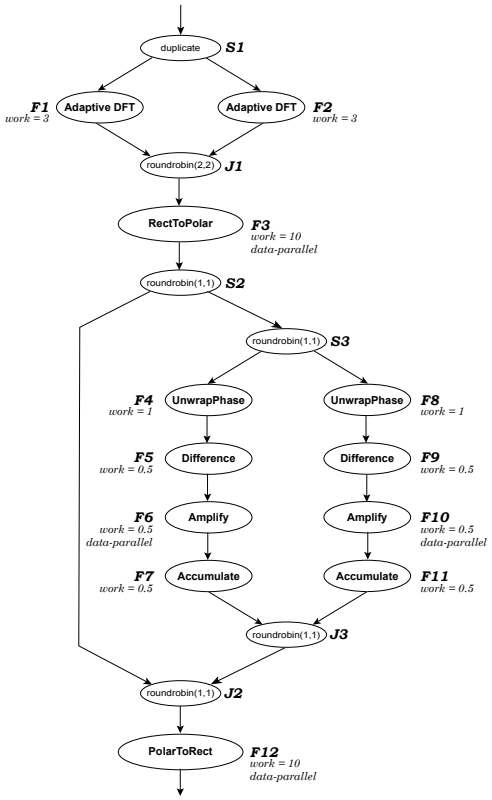

Figure 1. Stream graph for a simplified subset of our Vocoder benchmark. Following a set of sliding DFTs, the signal is converted to polar coordinates. Node `S2` sends the magnitude component to the left and the phase component to the right. In this simplified example, no magnitude adjustment is needed.

StreamIt provides three hierarchical primitives for composing filters into larger stream graphs. A *pipeline* connects streams sequentially; for example, there is a four-element pipeline beginning with `UnwrapPhase` in the Vocoder example (Figure 1). A *splitjoin* specifies independent, task-parallel streams that diverge from a common *splitter* and merge into a common *joiner*. For example, the `AdapativeDFT` filters in Figure 1 form a two-element splitjoin (each filter is configured with different parameters). The final hierarchical primitive in StreamIt is the *feedbackloop*, which provides a way to create cycles in the graph. In practice, feedbackloops are rare and we do not consider them in this paper.

In this paper, we require that the push, pop, and peek rates of each filter are known at compile time. This enables the compiler to calculate a steady-state for the stream graph: a repetition of each filter that does not change the number of items buffered on any data channel [26, 19]. In combination with a simple program analysis that estimates the number of operations performed on each invocation of a given work function, the steady-state repetitions offer an estimate of the work performed by a given filter as a fraction of the overall program execution. This estimate is important for our software pipelining technique.

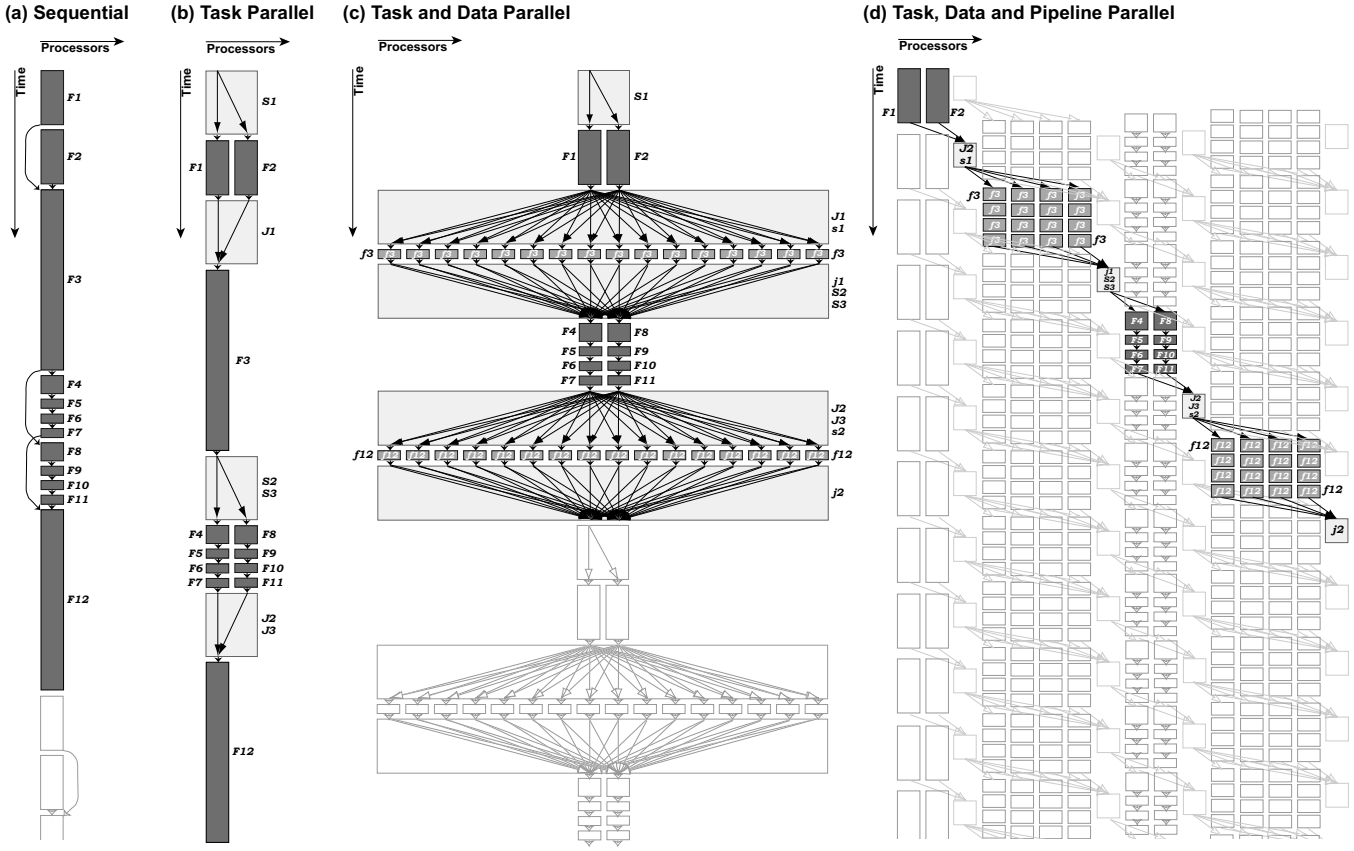

Figure 2. Parallel execution models for stream programs. Each block corresponds to a filter in the Vocoder example (Figure 1). The height of the block reflects the amount of work contained in the filter.

3. Coarse-Grained Data Parallelism

There is typically widespread data parallelism in a stream graph, as stateless filters can be applied in parallel to different parts of the data stream. For example, Figure 3a depicts our Filterbank benchmark, in which all of the filters are stateless. We detect stateless filters using a simple program analysis that tests whether there are any filter fields (state variables) that are written during one iteration of the work function and read during another. To leverage the implicit data parallelism of stateless filters, we convert the filters into an explicit splitjoin of many filters, such that each filter can be mapped to a separate processor. This process, which is called *filter fission*, causes the steady-state work of the original filter to be evenly split across the components of the splitjoin (though the `work` function is largely unchanged, each fission product executes less frequently than the original filter). Fission is described in more detail elsewhere [15].

On a multicore architecture, it can be expensive to distribute data to and from the parallel products of filter fission. As a starting point, our technique only fisses filters in which the estimated computation to communication ratio is above a given threshold. (In our experiments, we use a threshold of 10 compute instructions to 1 item of communication.) To further mitigate the communication cost, we introduce two new techniques: coarsening the granularity, and complementing task parallelism.

3.1 Coarsening the Granularity

Stateless filters are often connected together in a pipeline (see Figure 3a). Using naive filter fission, each filter is converted to a splitjoin that scatters and gathers data to and from the data-parallel components. This corresponds to fine-grained data parallelism at the loop level. However, as this introduces excessive communication, we instead aim to fiss the entire pipeline into many parallel pipelines, such that each data-parallel unit maintains local communication. We implement this functionality by first fusing the pipeline into a single filter, and then fissing the filter into a data-parallel splitjoin. We apply fusion first because we assume that each data-parallel product will execute on a single core; fusion enables powerful inter-node optimizations such as scalar replacement [35] and algebraic simplification [3, 24].

Some pipelines in the application cannot be fully fused without introducing internal state, thereby eliminating the data parallelism. For example, in Figure 3a, the `LowPassFilters` and `HighPassFilters` perform peeking (a sliding window computation) and always require a number of data items to be present on the input channel. If either filter is fused with filters above it, the data items will become state of the fused filter, thereby prohibiting data parallelism. In the general case, the number of persistent items buffered between filters depends on the initialization schedule [20].

Thus, our algorithm for coarsening the granularity of data-parallel regions operates by fusing pipeline segments as much as possible so long as the result of each fusion is stateless. For every pipeline in the application, the algorithm identifies the largest sub-segments that contain neither stateful filters nor buffered items and fuses the sub-segments into single filters. It is important to note that pipelines may contain splitjoins in addition to filters, and thus stateless splitjoins may be fused during this process. While such fusion temporarily removes task parallelism, this parallelism will

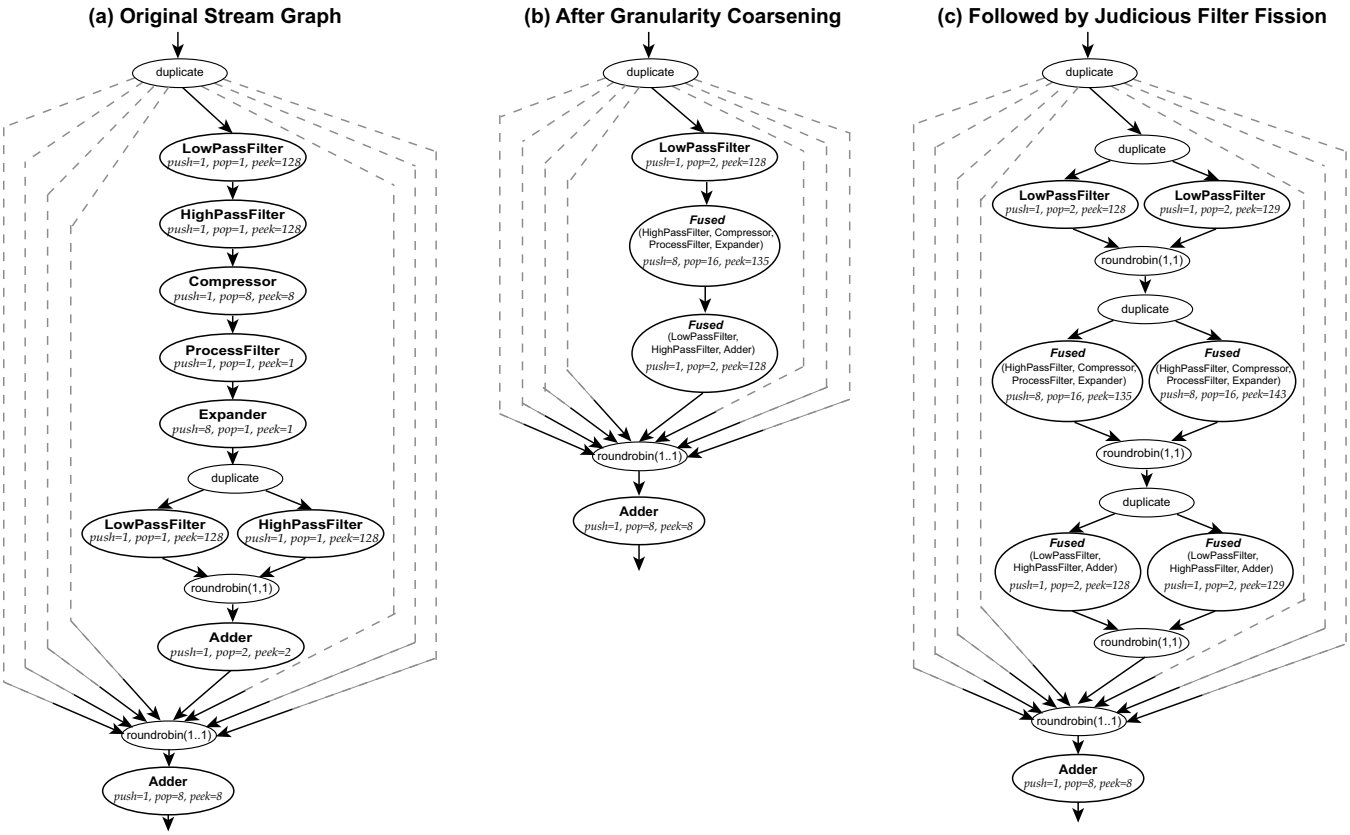

Figure 3. Exploiting coarse-grained data parallelism in the FilterBank benchmark. Only one pipeline of the toplevel splitjoin is shown; the other parallel streams are identical and are transformed in the same way.

be restored in the form of data parallelism once the resulting filter is fissed. The output of the algorithm on the FilterBank benchmark is illustrated in Figure 3b.

3.2 Complementing Task Parallelism

Even if every filter in an application is data-parallel, it may not be desirable to fiss each filter across all of the cores. Doing so would eliminate all task parallelism from the execution schedule, as only one filter from the original application could execute at a given time. An alternate approach is to preserve the task parallelism in the original application, and only introduce enough data parallelism to fill any idle processors. This serves to reduce the synchronization imposed by filter fission, as filters are fissed to a smaller extent and will span a more local area of the chip. Also, the task-parallel filters are a natural part of the algorithm and avoid any computational overhead imposed by filter fission (e.g., fission of peeking filters introduces a decimation stage on each fission product).

In order to balance task and data parallelism, we employ a "judicious fission" heuristic that estimates the amount of work that is task-parallel to a given filter and fisses the filter accordingly. Depicted in Algorithm 1, this algorithm works by ascending through the hierarchy of the stream graph. Whenever it reaches a splitjoin, it calculates the ratio of work done by the stream containing the filter of interest to the work done by the entire splitjoin (per steady state execution). Rather than summing the work within a stream, it considers the average work per filter in each stream so as to mitigate the effects of imbalanced pipelines. After estimating a filter's work as a fraction of those running in parallel, the algorithm attempts to fiss the filter the minimum number of times needed to ensure that none of the fission products contains more than $1/N$ of the total task-parallel work (where N is the total number of cores). Note that if several filters are being fissed, the work estimation is calculated ahead of time and is not updated during the course of fission.

Algorithm 1 Heuristic algorithm for fissing a filter as little as possible while filling all cores with task or data-parallel work.

```
▷ F is the filter to fiss; N is the total number of cores
JUDICIOUSFISSION(filter F, int N)
  ▷ Estimate work done by F as fraction of
  ▷ everyone running task-parallel to F
  fraction = 1.0
  Stream parent ← GETPARENT(F)
  Stream child ← F
  while parent ≠ null do
    if parent is splitjoin then
      total-work ←
          ∑_{c∈CHILDREN(parent)} AVGWORKPERFILTER(c)
      my-work ← AVGWORKPERFILTER(child)
      fraction ← fraction * my-work/total-work
    end if
    child ← parent
    parent ← GETPARENT(parent)
  end while
  ▷ Fiss F according to its weight in task-parallel unit
  Fiss F into CEIL(fraction * N) filters
```

Figure 3c illustrates the outcome of performing judicious fission on the coarsened-granularity stream graph from Figure 3b. Because

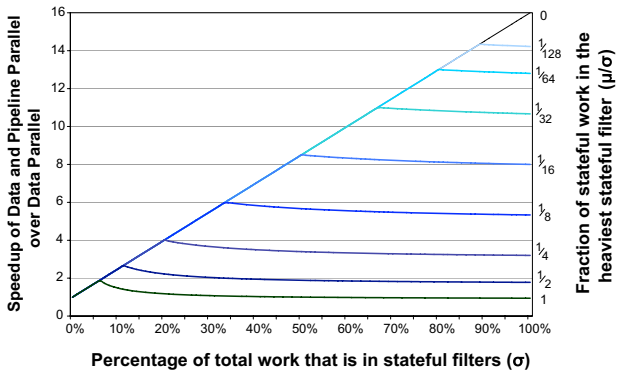

Figure 4. Potential speedups of pure pipeline parallelism over pure data parallelism for varying amounts of stateful work in the application. Each line represents a different amount of work in the heaviest stateful filter. The graph assumes 16 cores and does not consider task parallelism or communication costs.

there is 8-way task parallelism between all of the pipelines, the filters in each pipeline are fissed a maximum of 2 ways so as not to overwhelm the communication resources. As described in the Section 6, the combination of granularity coarsening and judicious fission offers a 6.8x mean speedup over a naive fission policy.

4. Coarse-Grained Software Pipelining

4.1 Benefits of Pipeline Parallelism

Pipeline parallelism is an important mechanism for parallelizing filters that have dependences from one iteration to another. Such "stateful" filters are not data parallel and do not benefit from the techniques described in the previous section. While many streaming applications have abundant data parallelism, even a small number of stateful filters can greatly limit the performance of a purely data-parallel approach on a large multicore architecture.

The potential benefits of pipeline parallelism are straightforward to quantify. Consider that the sequential execution of an application requires unit time, and let σ denote the fraction of work (sequential execution time) that is spent within stateful filters. Also let μ denote the maximum work performed by any individual stateful filter. Given N processing cores, we model the execution time achieved by two scheduling techniques: 1) data parallelism, and 2) data parallelism plus pipeline parallelism. In this exercise, we assume that execution time is purely a function of load balancing; we do not model the costs of communication, synchronization, locality, etc. We also do not model the impact of task parallelism.

1. Using data parallelism, $1-\sigma$ parts of the work are data-parallel and can be spread across all N cores, yielding a parallel execution time of $(1-\sigma)/N$. The stateful work must be run as a separate stage on a single core, adding σ units to the overall execution. The total execution time is $\sigma + (1-\sigma)/N$.
2. Using data and pipeline parallelism, any set of filters can execute in parallel during the steady state. (That is, each stateful filter can now execute in parallel with others; the stateful filter itself is not parallelized.) The stateful filters can be assigned to the processors, minimizing the maximum amount of work allocated to any processor. Even a greedy assignment (filling up one processor at a time) guarantees that no processor exceeds the lower-bound work balance of σ/N by more than μ, the heaviest stateful filter. Thus, the stateful work can always complete in $\sigma/N + \mu$ time. Remaining data parallelism can be freely distributed across processors. If it fills each processor to $\sigma/N + \mu$ or beyond, then there is perfect utilization and execution completes in $1/N$ time; otherwise, the state is the bottleneck. Thus the general execution time is $\max(\sigma/N + \mu, 1/N)$.

Using these modeled runtimes, Figure 4 illustrates the potential speedup of adding pipeline parallelism to a data-parallel execution model for various values of μ/σ on a 16-core architecture. In the best case, μ/σ approaches 0 and the speedup is $(\sigma + (1-\sigma)/N)/\max(\sigma/N, 1/N) = 1+\sigma*(N-1)$. For example, if there are 16 cores and even as little as $1/15$th of the work is stateful, then pipeline parallelism offers potential gains of 2x. For these parameters, the worst-case gain ($\mu = \sigma$) is 1.8x. The best and worst cases diverge further for larger values of σ.

4.2 Exploiting Pipeline Parallelism

At any given time, pipeline-parallel actors are executing different iterations from the original stream program. However, the distance between active iterations must be bounded, as otherwise the amount of buffering required would grow toward infinity. To leverage pipeline parallelism, one needs to provide mechanisms for both decoupling the schedule of each actor, and for bounding the buffer sizes. This can be done in either hardware or software.

In coarse-grained *hardware pipelining*, groups of filters are assigned to independent processors that proceed at their own rate (see Figure 5a). As the processors have decoupled program counters, filters early in the pipeline can advance to a later iteration of the program. Buffer size is limited either by blocking FIFO communication, or by other synchronization primitives (e.g., a shared-memory data structure). However, hardware pipelining entails a performance tradeoff:

- If each processor executes its filters in a single repeating pattern, then it is only beneficial to map a contiguous[2] set of filters to a given processor. Since filters on the processor will always be at the same iteration of the steady state, any filter missing from the contiguous group and executing at a remote location would only increase the latency of the processor's schedule. The requirement of contiguity can greatly constrain the partitioning options and thereby worsen the load balancing.
- To avoid the constraints of a contiguous mapping, processors could execute filters in a dynamic, data-driven manner. Each processor monitors several filters and fires any who has data available. This allows filters to advance to different iterations of the original stream graph even if they are assigned to the same processing node. However, because filters are executing out-of-order, the communication pattern is no longer static and a more complex flow-control mechanism (e.g., using credits) may be needed. There is also some overhead due to the dynamic dispatching step.

Coarse-grained *software pipelining* offers an alternative that does not have the drawbacks of either of the above approaches (see Figure 5b). Software pipelining provides decoupling by executing two distinct schedules: a loop prologue and a steady-state loop. The prologue serves to advance each filter to a different iteration of the stream graph, even if those filters are mapped to the same core. Because there are no dependences between filters within an iteration of the steady-state loop, any set of filters (contiguous or non-contiguous) can be assigned to a core. This offers a new degree of freedom to the partitioner, thereby enhancing the load balancing. Also, software pipelining avoids the overhead of the demand-driven model by executing filters in a fixed and repeatable pattern on each

[2] In an acyclic stream graph, a set of filters is *contiguous* if, in traversing a directed path between any two filters in the set, the only filters encountered are also within the set.

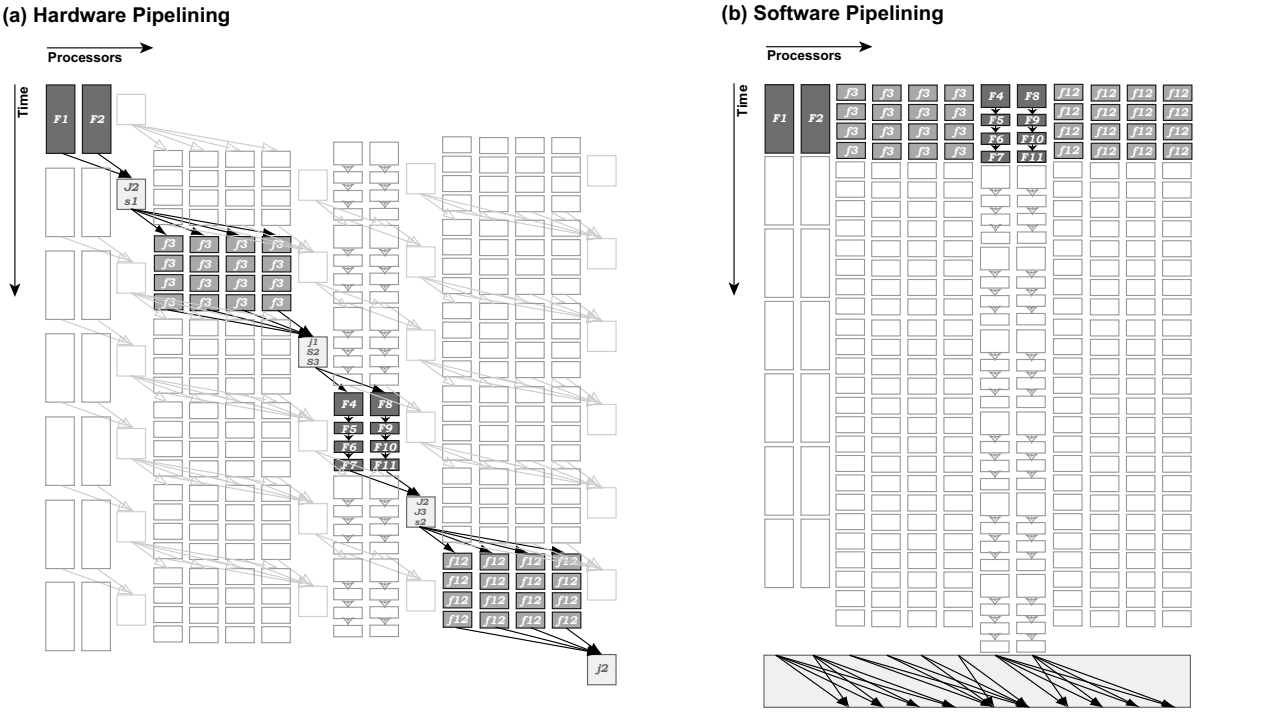

Figure 5. Comparison of hardware pipelining and software pipelining for the Vocoder example (see Figure 1). For clarity, the same assignment of filters to processors is used in both cases, though software pipelining admits a more flexible set of assignments than hardware pipelining. In software pipelining, filters read and write directly into buffers and communication is done at steady-state boundaries. The prologue schedule for software pipelining is not shown.

core. Buffering can be bounded by the on-chip communication networks, without needing to resort to software-based flow control.

4.3 Software Pipelining Implementation

Our software pipelining algorithm maps filters to cores in a similar manner that traditional algorithms map instructions to ALUs. This transformation is enabled by an important property of the StreamIt programming model, namely that the entire stream graph is wrapped with an implicit outer loop. As the granularity of software pipelining increases from instructions to filters, one needs to consider the implications for managing buffers and scheduling communication. We also describe our algorithm for mapping filters to cores, and compare the process to conventional software pipelining.

We construct the loop prologue so as to buffer at least one steady state of data items between each pair of dependent filters. This allows each filter to execute completely independently during each subsequent iteration of the stream graph, as they are reading and writing to buffers rather than communicating directly. The buffers could be stored in a variety of places, such as the local memory of the core, a hardware FIFO, a shared on-chip cache, or an off-chip DRAM. On Raw, off-chip DRAM offers higher throughput than a core's local memory, so we decided to store the buffers there. However, we envision that on-chip storage would be the better choice for most commodity multicores.

As filters are reading and writing into buffers in distributed memory banks, there needs to be a separate communication stage to shuffle data between buffers. Some of this communication is a direct transfer of data, while others performs scatter or gather operations corresponding to the splitjoins in StreamIt. The communication stage could be implemented by DMA engines, on-chip networks, vector permutations, or other mechanisms. On Raw, we leverage the programmable static network to perform all communication in a single stage, which is situated between iterations of the steady state. On architectures with DMA engines, it would be possible to parallelize the communication stage with the computation stage by double-buffering the I/O of each filter.

In assigning filters to cores, the goal is to optimize the load balancing across cores while minimizing the synchronization needed during the communication stage. We address these criteria in two passes, first optimizing load balancing and then optimizing the layout. As the load-balancing problem is NP-complete (by reduction from SUBSET-SUM [29]), we use a greedy partitioning heuristic that assigns each filter to one of N processors. The algorithm considers filters in order of decreasing work, assigning each one to the processor that has the least amount of work so far. As described in Section 4.1, this heuristic ensures that the bottleneck processor does not exceed the optimum by more than the amount of work in the heaviest filter.

To minimize synchronization, we wrap the partitioning algorithm with a selective fusion pass. This pass repeatedly fuses the two adjacent filters in the graph that have the smallest combined work. After each fusion step, the partitioning algorithm is re-executed; if the bottleneck partition increases by more than a given threshold (10%), then the fusion is reversed and the process terminates. This process increases the computation to communication ratio of the stream graph, while also leveraging the inter-node fusion optimizations mentioned previously. It improves performance by up to 2x on the Radar benchmark, with a geometric mean of 15% across all benchmarks.

Overall, coarse-grained software pipelining on a multicore architecture avoids many of the complications and exposes many new optimization opportunities versus traditional software pipelining. In traditional software pipelining, the limited size of the register file is always an adversary (register pressure), but there is ample memory available for buffering stream data. Another recurring issue tra-

ditionally is the length of the prologue to the software pipelined loop, but this is less problematic in the streaming domain because the steady-state executes longer. Lastly, and most importantly, we can exploit these properties to fully pipeline the graph, removing all dependences and thus removing all the constraints for our scheduling algorithm.

5. Implementation and Methodology

5.1 The Raw Architecture

We target the Raw microprocessor [37, 40], which addresses the wire delay problem by providing direct instruction set architecture (ISA) analogs to three underlying physical resources of the processor: gates, wires and pins. The architecture exposes the gate resources as a scalable 2-D array of identical, programmable cores, that are connected to their immediate neighbors by four on-chip networks. Values routed through the networks off of the side of the array appear on the pins, and values placed on the pins by external devices (wide-word A/Ds, DRAMS, etc.) appear on the networks. Each of the cores contains a compute processor, some memory and two types of routers—one static, one dynamic—that control the flow of data over the networks as well as into the compute processor. The compute processor interfaces to the network through a bypassed, register-mapped interface [37] that allows instructions to use the networks and the register files interchangeably.

Because we generate bulk DRAM transfers, we do not want these optimizable accesses to become the bottleneck of the hardware configuration. So, we employ a simulation of a CL2 PC 3500 DDR DRAM, which provides enough bandwidth to saturate both directions of a Raw port [38]. Additionally, each chipset contains a streaming memory controller that supports a number of simple streaming memory requests. In our configuration, 16 such DRAMs are attached to the 16 logical ports of the chip. The chipset receives request messages over the dynamic network for bulk transfers to and from the DRAMs. The transfers themselves can use either the static network or the general dynamic network (the desired network is encoded in the request).

The results in this paper were generated using btl, a cycle-accurate simulator that models arrays of Raw cores identical to those in the .15 micron 16-core Raw prototype ASIC chip, with a target clock rate of 450 MHz. The core employs as compute processor an 8-stage, single issue, in-order MIPS-style pipeline that has a 32 KB data cache, 32 KB of instruction memory, and 64 KB of static router memory. The simulator includes a 2-way set associative hardware instruction caching mechanism (not present in the hardware) that is serviced over the dynamic network, with resource contention modeled accordingly.

5.2 StreamIt Compiler Infrastructure

The techniques presented in this paper are evaluated in the context of the StreamIt compiler infrastructure. The system includes a high-level stream IR with a host of graph transformations including graph canonicalization, synchronization removal, refactoring, fusion, and fission [15]. Also included are domain specific optimizations for linear filters (e.g., FIR, FFT, and DCT) [24], state-space analysis [3], and cache optimizations [35]. We leverage StreamIt's spatially-aware Raw backend for this work.

Previously, we described hardware and software pipelining as two distinct techniques for exploiting pipeline parallelism. The StreamIt compiler has full support for each. It also implements a hybrid approach where hardware pipelined units are scheduled in a software pipelined loop. While we were excited by the possibilities of the hybrid approach, it does not provide a benefit on Raw due the tight coupling between processors and the limited FIFO buffering of the network. Although these are not fundamental limits of the architecture, they enforce fine-grained orchestration of communication and computation that is a mismatch for our coarse-grained execution model.

In the compiler, we elected to buffer all streaming data off-chip. Given the Raw configuration we are simulating and for the regular bulk memory traffic we generate, it is more expensive to stream data onto the network from a core's local data cache than to stream the data from the streaming memory controllers. A load hit in the data cache incurs a 3-cycle latency. So although the networks are register-mapped, two instructions must be performed to hide the latency of the load, implying a maximum bandwidth of 1/2 word per cycle, while each streaming memory controller has a load bandwidth of 1 word per cycle for unit-stride memory accesses. When targeting an architecture with more modest off-chip memory bandwidth, the stream buffers could reside completely in on-chip memory. For example, the total buffer allocation for each of our benchmarks in Section 6 would fit in Cell's 512KB L2 cache.

Streaming computation requires that the target architecture provides an efficient mechanism for implementing split and join (scatter and gather) operations. The StreamIt compiler programs the switch processors to form a network to perform the splitting (including duplication) and the joining of data streams. Since Raw's DRAM ports are banked, we must read all the data participating in the reorganization from off-chip memory and write it back to off-chip memory. The disadvantages to this scheme are that all the data required for joining must be available before the join can commence and the compute processors of the cores involved are idle during the reorganization. Architectures that include decoupled DMA engines (e.g., Cell) can overlap splitting/joining communication with useful computation.

5.2.1 Baseline Scheduler

To facilitate evaluation of coarse-grained software pipelining, we implemented a separate scheduling path for a non-software-pipelined schedule that executes the steady-state respecting the dataflow dependencies of the stream graph. This scheduling path ignores the presence of the encompassing outer loop in the stream graph and is employed for the task and task + data parallel configurations of the evaluation section.

This scheduling problem is equivalent to static scheduling of a coarse-grained dataflow DAG to a multiprocessor which has been a well-studied problem over the last 40 years (see [23] for a good review). We leverage simulated annealing as randomized solutions to this static scheduling problem are superior to other heuristics [22].

Briefly, we generate an initial layout by assigning the filters of the stream graph to processors in dataflow order, assigning a random processing core to each filter. The simulated annealing perturbation function randomly selects a new core for a filter, inserting the filter at the correct slot in the core's schedule of filters based on the dataflow dependencies of the graph. The cost function of the annealer uses our static work estimation to calculate the maximum critical path length (measured in cycles) from a source to a sink in the graph. After the annealer is finished, we use the configuration that achieved the minimum critical path length over the course of the search.

5.2.2 Instruction-Level Optimizations

For the results detailed in the next section we ran a host of optimizations including function inlining, constant propagation, constant folding, array scalarization, and loop unrolling (with a factor of 4). These optimizations are especially important for a fused filter, as we can possibly unroll enough to scalarize constituent splitter and joiner buffers, eliminating the shuffling operations. We do not enable StreamIt's aggressive cache optimizations [35] or linear

Benchmark	Description	Filters	Peeking Filters	SplitJoins	Comp / Comm	Task Parallel Critical Path	σ	μ
BitonicSort	Bitonic Sort	28	0	16	8	25%	0%	0%
ChannelVocoder	Channel Voice Coder	54	34	1	22380	9%	0%	0%
DCT	16x16 IEEE Reference DCT	36	0	2	191	23%	0%	0%
DES	DES Encryption	33	0	8	15	89%	0%	0%
FFT	256 Element FFT	17	0	0	63	100%	0%	0%
Filterbank	Filter Bank for Multirate Signal Processing	68	32	9	2962	9%	0%	0%
FMRadio	FM Radio with Equalizer	29	14	7	673	17%	0%	0%
Serpent	Serpent Encryption	61	0	25	36	52%	0%	0%
TDE	Time Delay Equalization for GMTI	28	0	0	335	100%	0%	0%
MPEG2Decoder	MPEG-2 Block and Motion Vector Decoding	26	0	5	102	59%	1%	1%
Vocoder	Bitrate Reduction Vocoder	96	17	8	180	85%	16%	1%
Radar	Radar Array Front-End	54	0	2	1341	11%	98%	4%

Figure 6. Benchmark descriptions and characteristics.

optimizations [24], as they conflate too many factors into the experiments. Finally, we produce a mix of C and assembly code that is compiled with GCC 3.4 at optimization level 3.

6. Experimental Evaluation

In this section we present an evaluation of our full compiler system and compare to previous techniques for compiling stream programs to multicore architectures. This section will include an elaboration of the following contributions and conclusions:

- Our compiler achieves consistent and excellent parallelization of our benchmark suite to a 16-core architecture, with a mean speedup of 11.2x over sequential, single-core performance.
- Our technique for exploiting coarse-grained data parallelism achieves a mean performance gain of 9.9x over a sequential, single-core baseline. This also represents a 6.8x speedup over a fine-grained data parallel approach.
- Coarse-grained software pipelining is a an effective technique for extracting parallelism beyond task and data parallelism, with an additional mean speedup of 1.45x for our benchmarks with stateful computation and 1.13x across all of our benchmarks. On its own, our software pipelining technique affords a 7.7x performance gain over a sequential single-core baseline.
- Our compiler, employing the combination of the techniques presented in this paper, improves upon our previous work that exploited a combination of task and hardware pipeline parallelism, with a mean speedup of 1.84x.

In the evaluation, speedup of configuration A over B is calculated as the throughput for an average steady-state of A divided by B; the initialization and prologue schedules, if present, are not included. Figure 7 presents a table of throughput speedups normalized to single-core for most configurations. Previous work has shown that sequential StreamIt executing on a single Raw core outperformed hand-written C implementations executing on a single core over a benchmark suite similar to ours [38]. Furthermore, we have increased the performance of sequential compilation since this previous evaluation.

6.1 Benchmark Suite

We evaluate our techniques using the benchmark suite given in Figure 6. The benchmark suite consists of 12 StreamIt applications. MPEG2Decoder implements the block decoding and the motion vector decoding of an MPEG-2 decoder, containing approximately one-third of the computation of the entire MPEG-2 decoder. The DCT benchmark implements a 16x16 IEEE reference DCT while the MPEG2Decoder benchmark includes a fast 8x8 DCT as a component. For additional information on MPEG2Decoder, Vocoder, and Radar, please refer to [11], [34], and [25], respectively.

In the table, the measurements given in each column are obtained from the stream graph as conceived by the programmer, before it is transformed by our techniques. The "Filters" columns gives the total number of filters in the stream (including file input filters and file output filters that are not mapped to cores). The number of filters that perform peeking is important because peeking filters cannot be fused with upstream neighbors without introducing internal state. "Comp / Comm" gives the static estimate of the computation to communication ratio of each benchmark for one steady-state execution. This is calculated by totaling the computation estimates across all filters and dividing by the number of dynamic `push` or `pop` statements executed in the steady-state (all items pushed and popped are 32 bits). Notice that although the computation to communication ratio is much larger than one across our benchmarks, we will demonstrate that inter-core synchronization is an important factor to consider. "Task Parallel Critical Path" calculates, using static work estimates, the work that is on the critical path for a task parallel model, assuming infinite processors, as a percentage of the total work. Smaller percentages indicate the presence of more task parallelism.

"σ" is defined in Section 4.1 as the fraction of total work that is stateful. "μ" is the maximum fraction of total work performed by any individual stateful filter. Referring to Figure 6, we see that three of our benchmarks include stateful computation. Radar repeatedly operates on long columns of an array requiring special behavior at the boundaries; thus, the state tracks the position in the column and does some internal buffering. Vocoder performs an adaptive DFT that uses a stateful decay to ensure stability; it also needs to retain the previous output across one iteration within a phase transformation. MPEGDecoder has negligible state in retaining predicted motion vectors across one iteration of work.

6.2 Task Parallelism

To motivate the necessity of our parallelism extraction techniques let us first consider the task parallel execution model. This model closely approximates a thread model of execution where the only form of coarse-grained parallelism exploited is fork/join parallelism. In our implementation, the sole form of parallelism exploited in this model is the parallelism across the children of a splitjoin. The first bar of Figure 8 gives the speedup for the each of our benchmarks running in the task parallel model executing on 16-core Raw, normalized to sequential StreamIt executing on a single core of Raw. For the remainder of the paper, unless otherwise noted, we target all 16 cores of Raw. The mean performance speedup for task parallelism is 2.27x over sequential performance. We can see that for most of our benchmarks, little parallelism is exploited; notable exceptions are Radar, ChannelVocoder, and FilterBank. Each contains wide splitjoins of load-balanced children. In the case of BitonicSort, the task parallelism is expressed at too fine a granularity for the communication system. Given that we are

Benchmark	Throughput Normalized to Single Core StreamIt					Task + Data + Soft Pipe	
	Task	Hardware Pipelining	Task + Data	Task + Soft Pipe	Task + Data + Soft Pipe	Compute Utilization	MFLOPS @ 450 MHz
BitonicSort	0.3	4.5	8.4	3.6	9.8	54.9%	N/A
ChannelVocoder	9.1	4.4	12.0	10.2	12.4	59.7%	940
DCT	3.9	4.1	14.4	5.7	14.6	67.6%	1316
DES	1.2	4.2	13.9	6.8	13.9	86.8%	N/A
FFT	1.0	8.4	7.9	7.7	7.9	45.7%	538
Filterbank	11.0	7.5	14.2	14.8	14.8	86.6%	1350
FMRadio	3.2	4.0	8.2	7.3	8.6	87.6%	1169
Serpent	2.6	16.4	15.7	14.0	15.7	56.6%	N/A
TDE	1.0	14.0	8.8	9.5	9.6	45.5%	352
MPEG2Decoder	1.9	3.4	12.8	5.1	12.1	63.9%	N/A
Vocoder	1.0	3.5	3.0	3.0	5.1	31.8%	192
Radar	8.5	10.9	9.2	19.6	17.7	97.7%	2006
Geometric Mean	**2.3**	**6.1**	**9.9**	**7.7**	**11.2**		

Figure 7. Throughput speedup comparison and Task + Data + Software Pipelining performance results.

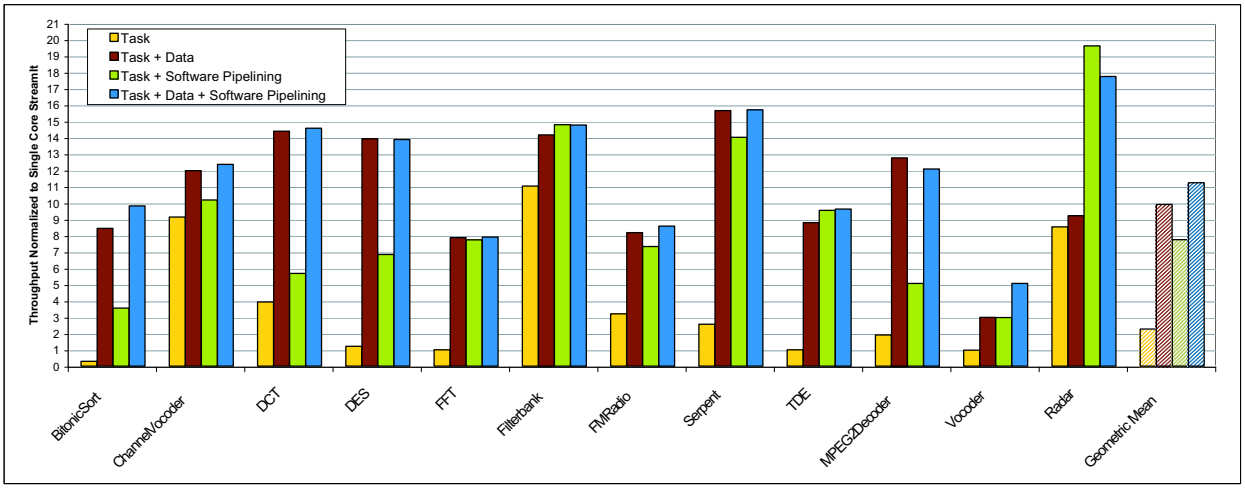

Figure 8. Task, Task + Data, Task + Software Pipelining, and Task + Data + Software Pipelining normalized to single core.

targeting a 16-core processor, a mean speedup of 2.27x is inadequate.

6.3 Coarse-Grained Data Parallelism

The StreamIt programming model facilitates relatively simple analysis to determine opportunities for data parallelism. But the granularity of the transformations must account for the additional synchronization incurred by data-parallelizing a filter. If we attempt to exploit data parallelism at a fine granularity, by simply replicating each stateless filter across the cores of the architecture we run the risk of overwhelming the communication substrate of the target architecture. To study this, we implemented a simple algorithm for exposing data parallelism: replicate each filter by the number of cores, mapping each fission product to its own core. We call this fine-grained data parallelism. In Figure 9, we show this technique normalized to single-core performance. Fine-grained data parallelism achieves a mean speedup of only 1.40x over sequential StreamIt. Note that FilterBank is not included in Figure 9 because the size of the fine-grained data parallel stream graph stressed our infrastructure. For four of our benchmarks, fine-grained duplication on 16 cores has lower throughput than single core.

This motivates the need for a more intelligent approach for exploiting data parallelism in streaming applications when targeting multicore architectures. The second bar of Figure 8 gives the speedup of coarse-grained data parallelism over single-core StreamIt. The mean speedup across our suite is 9.9x over single core and 4.36x over our task parallel baseline. BitonicSort, whose original granularity was too fine, now achieves a 8.4x speedup over a single core. 6 of our 12 applications are stateless and non-peeking (BitonicSort, DCT, DES, FFT, Serpent, and TDE) and thus fuse to one filter that is fissed 16 ways. For these benchmarks the mean speedup is 11.1x over the single core. For DCT, the algorithm data-parallelizes the bottleneck of the application (a single filter that performs more than 6x the work of each of the other filters). Coarse-grained data parallelism achieves a 14.6x speedup over single-core, while fine-grained achieves only 4.0x because it fisses at too fine a granularity, ignoring synchronization. Coarsening and then parallelizing reduces the synchronization costs of data parallelizing. For Radar and Vocoder, data parallelism is paralyzed by the preponderance of stateful computation.

6.4 Coarse-Grained Software Pipelining

Our technique for coarse-grained software pipelining is effective for exploiting coarse-grained pipelined parallelism (though it under-performs when compared to coarse-grained data parallelism). More importantly, combining software pipelining with our data parallelism techniques provides a cumulative performance gain, especially for applications with stateful computation.

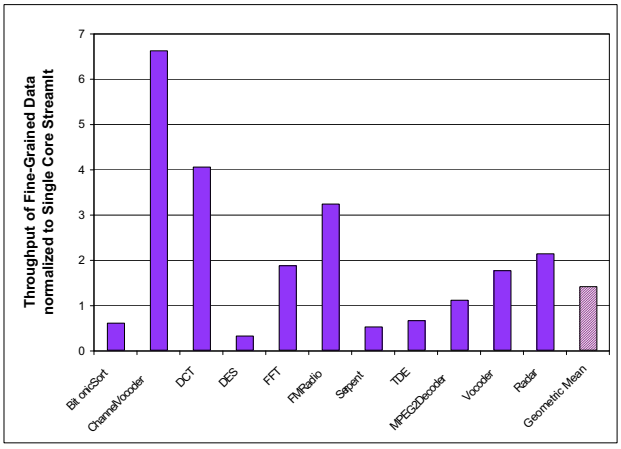

Figure 9. Fine-Grained Data Parallelism normalized to single core.

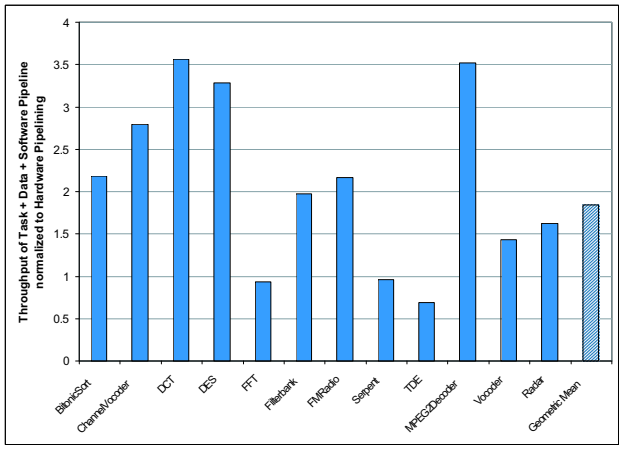

Figure 10. Task + Data + Software Pipelining normalized to Hardware Pipelining.

The third bar of Figure 8 gives the speedup for software pipelining over single core. On average, software pipelining has a speedup of 7.7x over single core (compare to 9.9x for data parallelism) and a speedup of 3.4x over task parallelism. Software pipelining performs well when it can effectively load-balance the packing of the dependence-free steady-state. In the case of Radar, TDE, FilterBank, and FFT, software pipelining achieves comparable or better performance compared to data parallelism (see Figure 8). For these applications, the workload is not dominated by a single filter and the resultant schedules are statically load-balanced across cores. For the Radar application, software pipelining achieves a 2.3x speedup over data parallelism and task parallelism because there is little coarse-grained data parallelism to exploit and it can more effectively schedule the dependence-free steady-state.

However, when compared to data parallelism, software pipelining is hampered by its inability to reduce the bottleneck filter when the bottleneck filter contains stateless work (e.g., DCT, MPEGDecoder). Also, our data parallelism techniques tend to coarsen the stream graph more than the selective fusion stage of software pipelining, removing more synchronization. For example, in DES, selective fusion makes a greedy decision that it cannot remove communication affecting the critical path workload. Software pipelining performs poorly for this application when compared to data parallelism, 6.9x versus 13.9x over single core, although it calculates a load-balanced mapping. Another consideration when comparing software pipelining to data parallelism is that the software pipelining techniques rely more heavily on the accuracy of the static work estimation strategy, although it is difficult to quantify this effect.

6.5 Combining the Techniques

When we software pipeline the data-parallelized stream graph, we achieve a 13% mean speedup over data parallelism alone. The cumulative effect is most prominent when the application in question contains stateful computation; for such benchmarks, there is a 45% mean speedup over data parallelism. For example, the combined technique achieves a 69% speedup over each individual technique for Vocoder. For ChannelVocoder, FilterBank, and FM, software pipelining further coarsens the stream graph without affecting the critical path work (as estimated statically) and performs splitting and joining in parallel. Each reduces the synchronization encountered on the critical path.

The combined technique depresses the performance of MPEG by 6% because the selective fusion component of the software pipeliner fuses one step too far. In most circumstances, fusion will help to reduce inter-core synchronization by using the local memory of the core for buffering. Consequently, the algorithm does not model the communication costs of each fusion step. In the case of MPEG, it fuses too far and adds synchronization. The combined technique also hurts Radar as compared to only software pipelining because we fiss too aggressively and create synchronization across the critical path.

In Figure 7, we report the compute utilization and the MFLOPS performance (N/A for integer benchmarks) for each benchmark employing the combination of our techniques, task plus data plus software pipeline parallelism. Note that for our target architecture, the maximum number of MFLOPS achievable is 7200. The compute utilization is calculated as the number of instructions issued on each computer processor divided by the total number possible for a steady-state. The utilization accurately models pipeline hazards and stalls of Raw's single issue, in-order processing cores. We achieve generally excellent compute utilization; in 7 cases the utilization is 60% or greater.

6.6 Comparison to our Previous Work: Hardware Pipelining

In Figure 10 we show our combined technique normalized to our previous work for compiling streaming applications to multicore architectures. This baseline configuration is a maturation of the ideas presented in [15][3] and implements a task plus hardware pipeline parallel execution model relying solely on on-chip buffering and the on-chip static network for communication and synchronization. In this hardware pipelining model, we require that the number of filters in the stream graph be less than or equal to the number of processing cores of the target architecture. To achieve this, we repeatedly apply fusion and fission transformations as directed by a dynamic programming algorithm.

Our new techniques achieve a mean speedup of 1.84x over hardware pipelining. For most of our benchmarks, the combined techniques presented in this paper offer improved data parallelism, improved scheduling flexibility, and reduced synchronization compared to our previous work. This comparison demonstrates that combining our techniques is important for generalization to stateful benchmarks. For Radar, data parallelism loses to hardware pipelining by 19%, while the combined technique enjoys a 38% speedup. For Vocoder, data parallelism is 18% slower, while the combined technique is 30% faster.

Hardware pipelining performs well in 3 out of 12 benchmarks (FFT, TDE, and Serpent). This is because these applications con-

[3] Please note that due to a bug in our tools, the MFLOPS numbers reported in the proceedings version of [15] were inaccurate.

tain long pipelines that can be load-balanced. For example, the stream graph for Serpent is a pipeline of identical splitjoins that is fused down to a balanced pipeline. Hardware pipelining incurs less synchronization than usual in this case because the I/O rates of the filters are matched; consequently, its compute utilization is higher than our combined technique (64% versus 57%). The combined approach fuses Serpent to a single filter and then fisses it 16 ways, converting the pipeline parallelism into data parallelism. In this case, data-parallel communication is more expensive than the hardware pipelined communication.

7. Related Work

In addition to StreamIt, there are a number of stream-oriented languages drawing from domains such as functional, dataflow, CSP and synchronous programming [36]. The Brook language is architecture-independent and focuses on data parallelism [6]. Stream kernels are required to be stateless, though there is special support for reducing streams to a single value. StreamC/KernelC is lower level than Brook; kernels written in KernelC are stitched together in StreamC and mapped to the data-parallel Imagine processor [18]. SPUR adopts a similar decomposition between "microcode" stream kernels and skeleton programs to expose data parallelism [42]. Cg exploits pipeline parallelism and data parallelism, though the programmer must write algorithms to exactly match the two pipeline stages of a graphics processor [27]. Compared to these languages, StreamIt places more emphasis on exposing task and pipeline parallelism (all the languages expose data parallelism). By adopting the synchronous dataflow model of execution [26], StreamIt focuses on well-structured programs that can be aggressively optimized. The implicit infinite loop around programs is also a key StreamIt characteristic that enables the transformations in this paper. Spidle [10] is also a recent stream language that was influenced by StreamIt.

Liao et al. map Brook to multicore processors by leveraging the affine partitioning model [41]. While affine partitioning is a powerful technique for parameterized loop-based programs, in StreamIt we simplify the problem by fully resolving the program structure at compile time. This allows us to schedule a single steady state using flexible, non-affine techniques (e.g., simulated annealing) and to repeat the found schedule for an indefinite period at runtime. Gummaraju and Rosenblum map stream programs to a general-purpose hyperthreaded processor [16]. Such techniques could be integrated with our spatial partitioning to optimize per-core performance. Gu et al. expose data and pipeline parallelism in a Java-like language and use a compiler analysis to efficiently extract coarse-grained filter boundaries [12]. Ottoni et al. also extract decoupled threads from sequential code, using hardware-based software pipelining to distribute the resulting threads across cores [30]. By embedding pipeline-parallel filters in the programming model, we focus on the mapping step.

Previous work in scheduling computation graphs to parallel targets has focused on partitioning and scheduling techniques that exploit task and pipeline parallelism [33, 32, 28, 23, 13]. Application of loop-conscious transformations to coarse-grained dataflow graphs has been investigated. Unrolling (or "unfolding" in this domain) is employed for synchronous dataflow (SDF) graphs to reduce the initiation interval but they do not evaluate mappings to actual architectures [7, 31]. Software pipelining techniques have been applied to SDF graphs onto various embedded and DSP targets [5, 8], but has required programmer knowledge of both the application and the architecture. To our knowledge, none of these systems automatically exploit the combination of task, data, and pipeline parallelism. Furthermore, these systems do not provide a robust end-to-end path for application parallelization from a high-level, portable programming language.

8. Conclusions

As multicore architectures become ubiquitous, it will be critical to develop a high-level programming model that can automatically exploit the coarse-grained parallelism of the underlying machine without requiring heroic efforts on the part of the programmer. Stream programming represents a promising approach to this problem, as high-level descriptions of streaming applications naturally expose task, data, and pipeline parallelism.

In this paper, we develop general techniques for automatically bridging the gap between the original granularity of the program and the underlying granularity of the architecture. To bolster the benefits of data parallelism on a multicore architecture, we build coarse-grained data-parallel units that are duplicated as few times as needed. And to leverage the benefits of pipeline parallelism, we employ software pipelining techniques—traditionally applied at the instruction level—to coarse-grained filters in the program.

A detailed evaluation in the context of the StreamIt language and the 16-core Raw microprocessor offers very favorable results that are also quite consistent across diverse applications. Coarse-grained data parallelism offers a 4.4x speedup over a task-parallel baseline and a 9.9x speedup over a sequential code. Without our granularity coarsening pass, these reduce to 0.7x and 1.4x, respectively. Coarse-grained software pipelining improves the generality of the compiler, as it is able to parallelize stateful filters with dependences from one iteration to the next. Our two techniques are complementary and offer a combined speedup of 11.2x over the baseline (and 1.84x over our previous work).

Though data parallelism is responsible for greater speedups on a 16-core chip, pipeline parallelism may become more important as multicore architectures scale. Data parallelism requires global communication, and keeps resources sitting idle when it encounters stateful filters (or feedback loops). According to our analysis in Section 4.1, leveraging pipeline parallelism on a 64-core chip when only 10% of the filters have state could offer up to a 6.4x speedup (improvement in load balancing). Exposing pipeline parallelism in combination with data parallelism for the stateful benchmarks in our suite provided a 1.45x speedup over data parallelism alone.

As our techniques rely on specific features of the StreamIt programming model, the results suggest that these features are a good match for multicore architectures. Of particular importance are the following two language features:

1. Exposing producer-consumer relationships between filters. This enables us to coarsen the computation to communication ratio via filter fusion, and also enables pipeline parallelism.
2. Exposing the outer loop around the entire stream graph. This is central to the formulation of software pipelining; it also enables data parallelism, as the products of filter fission may span multiple steady-state iterations.

While our implementation targets Raw, the techniques developed should be applicable to other multicore architectures. As Raw has a relatively high communication bandwidth, coarsening the granularity of data parallelism may benefit commodity multicores even more. In porting this transformation to a new architecture, one may need to adjust the threshold computation-to-communication ratio that justifies filter fission. As for coarse-grained software pipelining, the scheduling freedom afforded should benefit many multicore systems. One should consider the most efficient location for intermediate buffers (local memory, shared memory, FIFOs, etc.) as well as the best mechanism for shuffling data (DMA, on-chip network, etc.). The basic algorithms for coarsening granularity, judicious fission, partitioning, and selective fusion are largely architecture-independent.

Acknowledgments

We would like thank the members of the StreamIt team, both past and present, and especially Jasper Lin, Rodric Rabbah, and Allyn Dimock for their contributions to this work. We are very grateful to Michael Taylor, Jonathan Eastep, and Samuel Larsen for their help with the Raw infrastructure, and to Ronny Krashinsky for his comments on this paper. This work is supported in part by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890, and NSF awards CNS-0305453 and EIA-0071841.

References

[1] Raza Microelectronics, Inc. `http://www.razamicroelectronics.com/products/xlr.htm`.

[2] StreamIt Language Specifi cation. `http://cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf`.

[3] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing Stream Programs Using Linear State Space Analysis. In *CASES*, San Francisco, CA, Sept. 2005.

[4] J. Andrews and N. Baker. Xbox 360 System Architecture. *IEEE Micro*, 26(2), 2006.

[5] S. Bakshi and D. D. Gajski. Partitioning and pipelining for performance-constrained hardware/software systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(4):419–432, 1999.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[7] L.-F. Chao and E. H.-M. Sha. Scheduling Data-Flow Graphs via Retiming and Unfolding. *IEEE Trans. on Parallel and Distributed Systems*, 08(12), 1997.

[8] K. S. Chatha and R. Vemuri. Hardware-Software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3), 2002.

[9] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. In *PLDI*, New York, NY, USA, 2005.

[10] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL Approach to Specifying Streaming Applications. In *2nd Int. Conf. on Generative Prog. and Component Engineering*, 2003.

[11] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe. MPEG-2 Decoding in a Stream Programming Language. In *IPDPS*, Rhodes Island, Greece, April 2006.

[12] W. Du, R. Ferreira, and G. Agrawal. Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism. In *Supercomputing*, 2005.

[13] E. and D. Messerschmitt. Pipeline interleaved programmable DSP's: Synchronous data flow programming. *IEEE Trans. on Signal Processing*, 35(9), 1987.

[14] W. Eatherton. The Push of Network Processing to the Top of the Pyramid. Keynote Address, Symposium on Architectures for Networking and Communications Systems, 2005.

[15] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

[16] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *MICRO*, 2005.

[17] H. P. Hofstee. Power Effi cient Processor Architecture and The Cell Processor. *HPCA*, 00:258–262, 2005.

[18] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 2003.

[19] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *LCTES*, San Diego, CA, June 2003.

[20] M. A. Karczmarek. Constrained and Phased Scheduling of Synchronous Data Flow Graphs for the StreamIt Language. Master's thesis, MIT, 2002.

[21] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[22] Y.-K. Kwok and I. Ahmad. FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999.

[23] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[24] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *PLDI*, San Diego, CA, June 2003.

[25] J. Lebak. Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, Lincoln Laboratory, Mass. Inst. of Technology, 2001.

[26] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[27] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH*, 2003.

[28] D. May, R. Shepherd, and C. Keane. Communicating Process Architecture: Transputers and Occam. *Future Parallel Computers: An Advanced Course, Pisa, Lecture Notes in Computer Science*, 272, June 1987.

[29] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[30] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*, 2005.

[31] K. Parhi and D. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs Via Optimum Unfolding. *IEEE Transactions on Computers*, 40(2), 1991.

[32] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A Hierarchical Multiprocessor Scheduling Framework for Synchronous Dataflow Graphs. Technical Report UCB/ERL M95/36, May 1995.

[33] J. L. Pino and E. A. Lee. Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors. *Proc. of the IEEE Conference on Acoustics, Speech, and Signal Processing*, 1995.

[34] S. Seneff. Speech transformation system (spectrum and/or excitation) without pitch extraction. Master's thesis, MIT, 1980.

[35] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache Aware Optimization of Stream Programs. In *LCTES*, Chicago, 2005.

[36] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[37] M. B. Taylor et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro vol 22, Issue 2, 2002.

[38] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA*, Munich, Germany, June 2004.

[39] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, France, 2002.

[40] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, et al. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9), 1997.

[41] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *CGO*, 2006.

[42] D. Zhang, Z.-Z. Li, H. Song, and L. Liu. A Programming Model for an Embedded Media Processing Architecture. In *SAMOS*, 2005.