Flexible Filters in Stream Programs

REBECCA L. COLLINS and LUCA P. CARLONI, Columbia University

The stream-processing model is a natural fit for multicore systems because it exposes the inherent locality and concurrency of a program and highlights its separable tasks for efficient parallel implementations. We present *flexible filters*, a load-balancing optimization technique for stream programs. Flexible filters utilize the programmability of the cores in order to improve the data-processing throughput of individual bottleneck tasks by "borrowing" resources from neighbors in the stream. Our technique is distributed and scalable because all runtime load-balancing decisions are based on point-to-point handshake signals exchanged between neighboring cores. Load balancing with flexible filters increases the system-level processing throughput of stream applications, particularly those with large dynamic variations in the computational load of their tasks. We empirically evaluate flexible filters in a homogeneous multicore environment over a suite of five real-word stream programs.

Categories and Subject Descriptors: D.1.3 [**Programing Techniques**]: Concurrent Programing; D.3.2 [**Programing Languages**]: Language Classification—*Data-flow languages*

General Terms: Design, Performance

Additional Key Words and Phrases: Stream programming, dynamic load balancing

ACM Reference Format:
Collins, R. L. and Carloni, L. P. 2013. Flexible filters in stream programs. ACM Trans. Embedd. Comput. Syst. 13, 3, Article 45 (December 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2539036.2539041

1. INTRODUCTION

The properties of the stream-processing model naturally lend it to the challenge of programming multicore platforms; in particular, the strict organization of stream programs as a sequence of filters communicating through FIFO data pipes mitigates the complexity of scheduling tasks and intercore communication. Stream processing has been deployed in a wide range of applications, including high-performance embedded applications, signal processing, image compression, and continuous database queries [Buck et al. 2004; Chandrasekaran et al. 2003; Gummaraju and Rosenblum 2005; Kapasi et al. 2003; Kudlur and Mahlke 2008; McCool 2006; Shah et al. 2003; Thies et al. 2001].

The stream-processing paradigm decomposes an application into a sequence of data items (*tokens*) and a collection of tasks (referred to as *filters* or *kernels*) that operate upon the stream of tokens as they "flow" through. Filters communicate with each other explicitly by exchanging tokens through point-to-point communication channels. This model exposes the inherent locality and concurrency of the application and enables the realization of efficient implementations based on mapping the filters onto parallel processor architectures. Given a stream program and a target architecture, the

This research is partially supported by the National Science Foundation under awards 0644202 and 0811012.
Corresponding author's email: rebecca.l.c@gmail.com.

DOI: http://dx.doi.org/10.1145/2539036.2539041

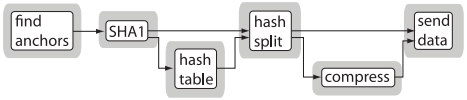


Fig. 1. Stream graph of the Dedup benchmark application.

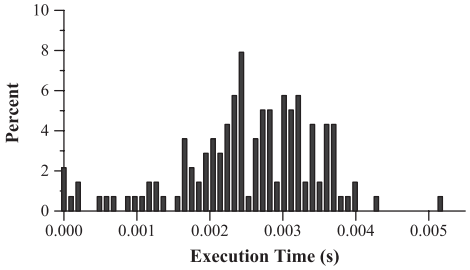


Fig. 2. Histogram of execution times for Dedup's compress filter.

filters of the stream program are mapped to the cores of the architecture, and the communication channels to the communication substructure of that architecture. The communication mapping includes mapping input and output buffers to the (possibly distributed) memory, and the communication events to underlying communication protocols, such as message passing. In this article, we work with a distributed memory architecture where each core has its own local memory, which is shared by the data and instructions associated with a task as well as communication buffers.

In general, it is a challenge to achieve an optimal mapping that maximizes the program performance given data dependencies among the filters and the available hardware resources (e.g., processing cores, memories, and interconnect). Moreover, the execution time of a software task is often variable, making mapping more difficult since the relative cost of filters with respect to each other is not constant. Consider the Dedup benchmark, a parallel compression application [Bienia et al. 2008], that can be implemented as a stream program with six main filters, as illustrated in Figure 1, which shows the corresponding *stream graph*. A data-dependent execution time characterizes the compress filter, illustrated by the histogram in Figure 2, which shows the distribution of execution times over a sample of input on that filter. The execution time to compress a data array varies by up to five milliseconds in the sample depending on the data content.

This article presents *flexible filters* as a technique for balancing stream programs on distributed-memory multicore platforms. Flexible filters combine static mapping of the stream program filters with dynamic load balancing of their execution. The goal is to increase the overall processing throughput of the stream program by reducing the impact of *bottleneck filters* running on particular cores. A filter can cause a bottleneck because either (a) its algorithmic characteristics make it disproportionately expensive to run on a given core with respect to the other filters running on neighboring cores or (b) at runtime, it may go through phases where it has to process a larger number of tokens per unit of time. When a filter becomes a bottleneck, its neighboring upstream or downstream filters, or both, may start suffering a loss of throughput, and ultimately, this affects the data processing throughput of the overall stream. If a slow computation creates a bottleneck by delaying the production of new tokens, the downstream filters may become idle due to the lack of inputs. Alternatively, a filter can also be a bottleneck if it cannot keep up with the data production of upstream filters. If this is the case, the input buffers of its processing core start filling up. This eventually leads to the emission of *backpressure* signals between the cores running the bottleneck filter and its upstream neighbors, thus forcing the upstream filters to become idle to avoid a loss of data from buffer overflows.

Figure 3 illustrates our proposed design flow to guide the application of flexible filters. Bottleneck filters in a stream application are identified through profiling of

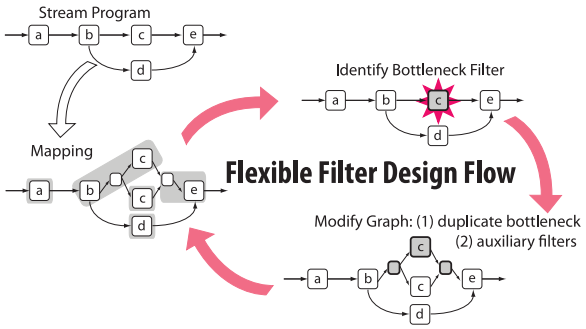


Fig. 3. The design flow to guide the application of flexible filters.

the application with the target multicore platform or through modeling. Based on this profiling, the graph is modified to include redundant copies of the flexible filter as well as auxiliary code which leverages the backpressure mechanism to dynamically activate the execution of the additional copies of the bottleneck filters when necessary, while preserving the correct ordering of the tokens in the data stream. Finally, the resulting stream graph is mapped to the target multicore architecture. This is a cyclical process, since the first program profile depends on a mapping of the application, and the modification to the original stream graph may give rise to new bottlenecks.

The main contribution of this article is a comprehensive presentation of flexible filters as an effective load-balancing technique for stream programs. The idea of flexible filters was first presented at EMSOFT 2009 [Collins and Carloni 2009], and its application to one particular high-performance embedded application was discussed in an extended abstract at HPEC 2010 [Collins and Carloni 2010]. Additional contributions with respect to these publications include an extension of the library of auxiliary functions to cover bottleneck filters with multiple input and output channels and a comprehensive experimental evaluation of flexible filters with a suite of five real-world stream programs.

The rest of this article is organized as follows: in Section 2, we describe how a stream graph is modified once a bottleneck filter has been identified; in Section 3, we discuss the implementation of the auxiliary functions, *flex_split* and *flex_merge*, which manage dynamic flow redirection; in Section 4, we describe a compositional performance model for stream programs which aids in profiling the trade-offs of different filter mappings and the expected throughput improvement gained through flexibility; in Section 5, we present experimental results to evaluate the performance gains that can be obtained with flexible filters. Our experiments show that flexible filters achieve speedup over a wide variety of application domains and in cases where the execution time of filters varies during runtime. Finally, in Section 6, we discuss the related work.

2. FLEXIBLE FILTERS

The presence of a bottleneck filter may limit performance by preventing utilization of the full capabilities of an architecture. This section first defines throughput as a performance metric and then presents a small example to illustrate how performance can be lost because of a bottleneck filter and how the incorporation of flexibility into a program corrects for this loss.

In order to implement a stream program on a multicore architecture, each of its filters must be mapped to at least one core. A core may host several filters and rely on a scheduler to time-multiplex the core's resources among the filters. The performance of a given implementation can be measured by its *maximum sustainable throughput (MST)*,

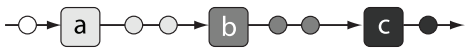


Fig. 4. Example of the structure of a stream program.

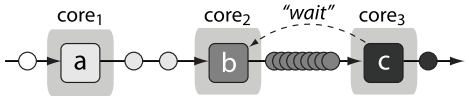


Fig. 5. Pipeline mapping.

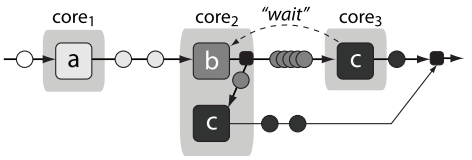


Fig. 6. Flexible-filter mapping.

that is, the maximum rate at which data tokens can be processed under the assumption that the environment is always willing to produce new tokens and never requires the system to stall through a backpressure signal. In an ideal multicore architecture, (1) the overhead of intercore communication and intracore context switching is negligible, and (2) each core has unlimited local memory. An ideal mapping of filters would result in a runtime execution where no core ever stalls and the MST scales linearly with the number of cores.

Consider the simple example of a generic stream program whose structure is shown in Figure 4: it consists of three filters a, b, and c with data tokens traveling between them on communication channels (a, b) and (b, c). If the filters have *execution times*[1] $L_a = 2$, $L_b = 2$, and $L_c = 3$, respectively, then the ideal MST (i.e., assuming that no core is idle and performance scales linearly) is $\frac{\#\ cores}{L_a+L_b+L_c} = \frac{3}{7} = 0.429$.

Figure 5 illustrates a simple pipeline mapping: each filter is mapped to a separate core. Using the same execution times as before, this mapping delivers an MST equal to $\frac{1}{3} = 0.333$, lower than the ideal 0.429 MST, because filter c can process a new data token only every three time steps, thus limiting the performance of the program. Once the buffers between core_2 and core_3 (where b and c are located, respectively) fill up, core_3 requests core_2 to stall occasionally through the emission of a backpressure signal (and backpressure continues to propagate upstream).

However, suppose that core_2 can also execute filter c. Then, instead of stalling, core_2 can "work ahead" on the data tokens in its buffers. Now the rate at which data tokens are processed by filter c increases, the load on core_3 decreases, and as a result, the system runs faster. Thus, load balancing based on flexible filters duplicates bottleneck filters and maps the duplicate copies together with upstream filters. For instance, as shown in Figure 6, adding flexibility to the stream program from Figure 5 makes it possible to alleviate the bottleneck caused by filter c on core_3. The new mapping duplicates filter c on core_2 so that core_2 can share the load of filter c.

Besides increasing the code footprint in core_2 with respect to the pipeline mapping, flexible filters also add some complexity to the program, because now the data stream is split and merged around core_3. The two auxiliary filters, *flex_split* and *flex_merge* accomplish the split and merge steps. These filters, which are represented as small black boxes in Figure 6, can be added to the stream program without changing any of the original filters. Flexible filters provide a notion of semantic preservation whereby the final output of the program preserves the ordering of tokens, and lossless channels guarantee that no token is dropped so that the resulting output data stream is unaltered when some filters are made flexible in the execution.

[1]The execution time of a filter is the time necessary to execute it on a given core as a stand-alone task. In a heterogeneous multicore architecture, the same filter would have different execution times when executed on different programmable cores. However, this example considers only homogeneous architectures.

2.1. Stateless Filters

Since the data splitting of *flex_split* and *flex_merge* around a filter f occurs agnostically with respect to f's operation, to be eligible for flexibility, a filter must be stateless, that is, given an input token x, a stateless filter will produce the same output token regardless of what tokens came before x. Stateless, or functional, programming applies to many classes of applications, including financial applications, telecommunications, and digital signal processing [Däcker 2000; Graef et al. 2006; Minsky and Weeks 2008].

Besides statelessness, an additional requirement for a filter f to be eligible for flexibility is that each firing of f consumes a constant number of input tokens. In our implementation of *flex_split* and *flex_merge*, we also assumed that each firing produces a constant number of output tokens; this, however, is not necessarily a requirement. An extended implementation could support dynamic output by bookkeeping to keep track of placeholders between sets of output tokens. Notice that a program may consist of both flexible filter and regular filters: no restrictions apply to filters which are not made flexible, since their output tokens are not rearranged.

Although stateful filters cannot themselves be flexible, the flexible filter approach can still sometimes improve the performance of a stream program that includes stateful filters. Only the bottleneck filter needs to be made flexible. Neighboring non-bottleneck filters do not prevent the application of flexibility to the rest of the program. For example, the Dedup benchmark described in Section 1 uses flexibility to speed up the compression filter, notwithstanding the presence of a stateful hash map filter earlier in the stream.

In some cases, it is possible to break a stateful bottleneck filter up so that the most computationally expensive part is stateless, even if the original bottleneck filter was stateful. Refactoring filters to remove their state is an application-dependent task which is not addressed by the flexible filter methodology but is left to application designers.

2.2. Static Load Balancing vs. Dynamic Load Balancing

The flexible filter solution combines a static mapping of stream tasks with dynamic flow management so that the flow may be redirected at runtime around bottlenecks, as allowed by flexibility in the static mapping. Note that in the example from Figure 6, a static load-balancing split and join could achieve the same speedup as flexible filters if each core always had the same execution time. Previous works have shown that static splits and joins of the data flow can be used to balance the workload of cores and improve performance [Gordon et al. 2006]. The decision of where to insert splits and joins and to what extent a job should be split is left to the compiler. Hence, it is a static optimization choice. On top of static load balancing, some systems analyze various possible load scenarios and enable several choices of static mappings from which the runtime system can dynamically choose in order to adapt to different scenarios [Chen et al. 2005; Stuijk et al. 2011].

Flexible filter load balancing is able to handle an environment where constant execution times are not known, which includes both the case where bottlenecks are data dependent as well as when the availability of resources in the system changes dynamically due to contention with other applications. Moreover, one of the key advantages of flexible filters is that they provide dynamic load balancing without any form of a centralized control or runtime mapping readjustment.

Flexible filters differ from previous load-balancing approaches because backpressure alone drives load balancing, and data dependencies across the filters in the stream program guide the task reassignment to idle cores rather than random reassignment [Chen et al. 2005; Fellheimer 2006; Hormati et al. 2009; Shah et al. 2003; Xing et al. 2005;

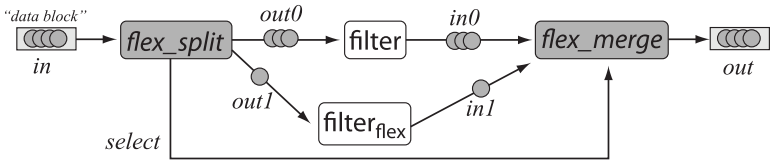


Fig. 7. Relationship of *flex_split* and *flex_merge*.

Zhang et al. 2008]. The approach is entirely distributed and sends no extra messages among cores beyond backpressure messages, which are already often present to prevent the communication buffers from overflowing. Since the runtime load drives load balancing, flexible filters can be used not only to optimize the implementation of programs whose filters have constantly unbalanced computational loads, but also to adjust temporary imbalances due to spikes of activity, for example, detecting "bargains" during the processing of real-time streaming stock-ticker data [Gedik et al. 2008].

3. IMPLEMENTATION OF FLEX_SPLIT AND FLEX_MERGE

After the programmer identifies potential bottleneck filters through profiling or other program analysis, the original stream program is transformed into a flexible stream program by duplicating these filters and by adding pairs of *flex_split* and *flex_merge* auxiliary filters around the flexible duplicates. Figure 7 shows the connections among the filters. *Flex_split* and *flex_merge* can be provided by an application-independent library because they do not depend on application-specific details. Furthermore, *flex_split* and *flex_merge* do not require modification of the original stream filters.

Implementations of *flex_split* and *flex_merge* work with *data blocks*. Each data block may consist of many data tokens, and the blocks, like tokens, form a stream and follow an ordering that depends on their place in the bigger stream. One difference between data tokens and data blocks with respect to scheduling the flow of data is that it is possible to break a data block up into several pieces and process them in parallel, while a data token is indivisible. Data blocks are a realistic abstraction, since real-life implementations frequently buffer data in order to amortize the overhead of communication.

A data token corresponds to the minimum data needed to fire the execution of a filter. In practice, the minimum data needed varies from one filter to the next. For example, one filter may require only a single integer for an input token, while another may require an array of integers.

A data block is the input unit for *flex_split*. The divisibility of data blocks is one factor that enables load balancing with flexible filters, but data blocks can only contain a finite number of data tokens and cannot be divided into arbitrarily-sized fractions. Lower granularity (fewer tokens per block) can limit the benefits of flexibility in the data stream because it puts more constraints on the possible data flow.

Flex_split (pseudocode shown in Algorithm 1) dynamically reuses the backpressure information on the current capacity of the downlink input buffers to manage load balancing by dividing the data stream between $out0$ and $out1$. Specifically, it checks how much space is available on the buffering queue for the primary copy of the flexible filter, f, and divides the data stream by sending as much data to f's primary copy as it can (stream $out0$) and then sending any leftover data to the flexible copy (stream $out1$). *Flex_split* also produces a *select* bitstream that contains information on how to reconstruct the correct ordering of the stream. *Flex_merge* (pseudocode in Algorithm 2) takes the input streams $in0$ and $in1$ from both of f's copies along with the *select* bitstream, which comes directly from *flex_split*. The *select* bitstream indicates which of

ALGORITHM 1: flex_split (Input: stream in; Output: streams $out0, out1, select$)

```
pop data block b from in
n0 ← min(avail(out0), |b|)
n1 ← |b| − n0
for i = 0 to n0 − 1 do
  push 0 to select
end for
for i = n0 to |b| − 1 do
  push 1 to select
end for
push n0 tokens from b to out0
push n1 tokens from b to out1
```

ALGORITHM 2: flex_merge (Input: streams $in0, in1, select$; Output: stream out)

```
pop i from select
if i is 0 then
  pop token t from in0
else
  pop token t from in1
end if
push t to out;
```

f's copies has the next data token, thus allowing *flex_merge* to reassemble the stream into its original order.

Backpressure plays a key role in the implementation of flexible filters. Before a core can send data downstream, it must ensure the availability of adequate buffering space for the data in the receiving core. A typical handshake protocol guarantees that buffers do not overflow and proceeds through a sequence of phases: it starts with the sending core placing a request to send data; then, the receiving core sends back an acknowledgement with information on how much data it can receive (backpressure); and finally the sending core sends the data. In practice, the various phases can be overlapped to further improve performance by adding sufficient memory space.

If a flexible filter is inherently slower than its upstream neighbor, then the imbalance will cause the input buffering queue of its primary copy to be full often, and *flex_split* will redirect the data flow to f's secondary copy at regular intervals. Instead, if f experiences only occasional spikes of activity that cause it to slow down, or if f's upstream neighbor occasionally creates extra data tokens on its output, then the flow of data will usually behave as if there is no redundant flexible filter present, and *flex_split* will intervene sporadically when a spike arises.

Finally, notice that instead of having a distinct bit for every token, a compressed format may reduce the *select* bitstream to counts of how many of the next tokens go to $out0$ and then how many go to $out1$. In practice, if the data tokens are vectors or other large data structures, using a distinct *select* bit for each token does not take up a significant portion of memory.

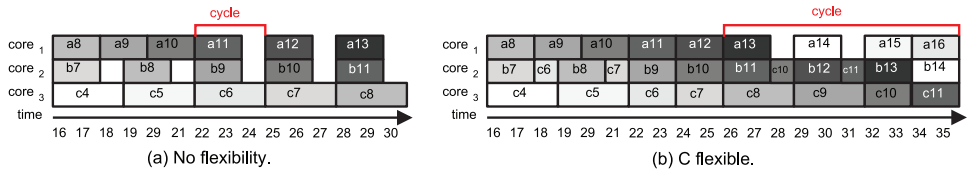
3.1. Example

We now walk through the execution of a stream program at runtime when flexibility is invoked to balance the load. Table I shows the timeline for the example shown in Figure 5, using the same example execution times used in Section 2 (with $L_a = 2$, $L_b = 2$, and $L_c = 3$). The table shows both the current step being executed on each core and the contents of the core's local buffering memory.

Table I. Baseline Pipeline Mapping Timeline

Time Steps		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
$core_1$	Step	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$	$a_{2,0}$	$a_{2,1}$	$a_{3,0}$	$a_{3,1}$	$a_{4,0}$	$a_{4,1}$
	Block(s)	0	0	1	1	2	2	3	3	4	4
$core_2$	Step			$b_{0,0}$	$b_{0,1}$	$b_{1,0}$	$b_{1,1}$	$b_{2,0}$	$b_{2,1}$	$b_{3,0}$	$b_{3,1}$
	Block(s)			0	0	1	1	2	2	3	3
$core_3$	Step					$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$
	Block(s)					0	0	0,1	1	1,2	1,2
Time Steps		t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}
$core_1$	Step	$a_{5,0}$	$a_{5,1}$	$a_{6,0}$	$a_{6,1}$	$a_{7,0}$	$a_{7,1}$	$a_{8,0}$	$a_{8,1}$	$a_{9,0}$	$a_{9,1}$
	Block(s)	5	5	6	6	7	7	8	8	8,9	9
$core_2$	Step	$b_{4,0}$	$b_{4,1}$	$b_{5,0}$	$b_{5,1}$	$b_{6,0}$	$b_{6,1}$	$b_{7,0}$	$b_{7,1}$	-	$b_{8,0}$
	Block(s)	4	4	4,5	5	5,6	5,6	6,7	6,7	6,7	7,8
$core_3$	Step	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	$c_{3,0}$	$c_{3,1}$	$c_{3,2}$	$c_{4,0}$	$c_{4,1}$	$c_{4,2}$	$c_{5,0}$
	Block(s)	2,3	2,3	2,3	3,4	3,4	3,4	4,5	4,5	4,5	5,6

Time Steps		t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}
$core_1$	Step	$a_{10,0}$	$a_{10,1}$	$a_{11,0}$	$a_{11,1}$	-	$a_{12,0}$
	Block(s)	9,10	9,10	10,11	10,11	10,11	11,12
$core_2$	Step	$b_{8,1}$	-	$b_{9,0}$	$b_{9,1}$	-	$b_{10,0}$
	Block(s)	7,8	7,8	8,9	8,9	8,9	9,10
$core_3$	Step	$c_{5,1}$	$c_{5,2}$	$c_{6,0}$	$c_{6,1}$	$c_{6,2}$	$c_{7,0}$
	Block(s)	5,6	5,6	6,7	6,7	6,7	7,8

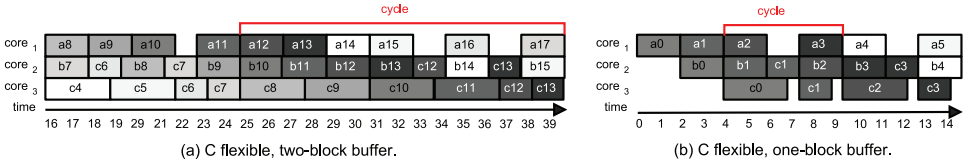


(a) No flexibility. (b) C flexible.

Fig. 8. Flexible filter timelines.

In Table I, each filter completes processing a block i in the time of that filter's execution time. For example, filter a, whose execution time is two, computes block i in two timesteps, denoted $a_{i,0}$ and $a_{i,1}$, respectively. Since filter c has an execution time of three, it must compute blocks in three timesteps ($c_{i,0}$, $c_{i,1}$, and $c_{i,2}$). Even though the filters' latencies are not equal, the buffer capacity allows the faster filters to work ahead initially. However, at time step t_{18}, $core_2$ must stall. At this timestep, $core_2$'s memory contains Blocks 6 and 7, and even though $core_1$ is ready to pass Block 8, $core_3$ holds Blocks 4 and 5 and will not be ready to take the next block from $core_2$ until it is done processing Block 4. Therefore, $core_2$ must wait until $core_3$ is ready to accept the next block before it can make space in its memory for Block 8. The state of the system is the same at time steps t_{22} and t_{25} in terms of the state of each core with respect to the blocks in that core's memory. In fact, the system begins to cycle through a pattern of states; in this case, the pattern from t_{22} to t_{24}. During one cycle, this implementation completes one block every three cycles, confirming that the MST is $\frac{1}{3}$, as calculated in Section 2. Note that if the filter latencies are unbalanced, stalling will occur no matter how much buffering space is available on the cores: additional memory simply extends the time that it takes to initially fill up the buffers.

Figure 8 summarizes two timelines in a more abbreviated format that does not include the current memory state. The timelines start at t_{16} using the same state of t_{16} in Table I and continue until a cyclic pattern emerges. Figure 8(a) depicts the same



(a) C flexible, two-block buffer. (b) C flexible, one-block buffer.

Fig. 9. Timeline when filter c has a granularity of two tokens per block.

timeline as in Table I. Figure 8(b) shows the timeline for a flexible-filter mapping, where filter c is made flexible and is mapped to core_2 and core_3 (same as Figure 6). The cyclic pattern for this mapping begins at time step t_{26} and continues until t_{35}. In the example, when no filters are flexible, the MST is degraded by 22% compared to the ideal throughput. When only filter c is flexible, the MST is increased to $\frac{4}{10} = 0.400$ (only 7% degradation). If b were also made flexible, the MST would reach its ideal limit of 0.429. Note that we are not simply duplicating a filter to achieve data parallelism (e.g., as in [Gordon et al. 2006]); instead, data parallelism is used to balance load dynamically as an alternative to stalling one of the processing cores in response to backpressure. The cost of flexibility is that the code of the flexible filter must be present on more than one core, and the flexible copy of the filter also shares communication buffers with the other filters.

Granularity of Firing Constraints and Buffer Size. In the previous examples, we assume that it is always possible to break one of c's data blocks up into thirds. Suppose, however, that the local data memory of each core only holds a block of two tokens for c. Since data tokens are the minimum amount of data that a filter can fire on, it is now only possible to break one of c's data blocks up into two pieces. Figure 9 repeats the mapping from Figure 8(b) to show the timeline when c has this constraint. There are two cases shown. In Figure 9(a), we assume buffers of size two, just like in the previous examples, while in Figure 9(b), we assume that the buffer has capacity for one data block only. At t_7 in Figure 9(b), core_3 must wait for Block 1 until core_2 is ready to send it. Similarly, core_1 must also wait to send Block 2 to core_2. When buffers have enough capacity for two blocks, the MST is $\frac{6}{15} = 0.4$, which is the same as the MST when we did not have the additional granularity constraint. However, when the buffers only hold one block, the MST is degraded to $\frac{2}{5.5} = 0.364$. This example shows that the local buffering memory plays a critical role in insulating performance from granularity constraints.

3.2. Practical Implementation Concerns

3.2.1. Multichannel Flexible Filters. The preceding discussion of *flex_split* and *flex_merge* assumes that the flexible filter has exactly one input and one output channel. Filters with multiple input and output channels may also be made flexible, but *flex_split* and *flex_merge* are inserted differently into the graph, and in the case of a filter with more than one input channel, *flex_split* requires modification.

For a filter with several output channels but only one input channel, no modification to *flex_split* or *flex_merge* is necessary: the flexible stream graph simply inserts a separate *flex_merge* for every output channel and copies the *select* bits of *flex_split* to each copy of *flex_merge*, as illustrated in Figure 10. Because each copy of the flexible filter produces data tokens to its output channels in the same order, the same *select* bitstream is correct for every *flex_merge*.

Adding flexibility around several input channels poses a greater challenge. Duplicating *flex_split* in the same way that *flex_merge* is duplicated for the multiple-output case does not result in a correct implementation, because *flex_split* splits the data

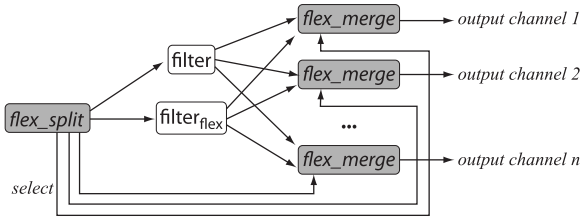


Fig. 10. Block diagram of a flexible filter with n output channels.

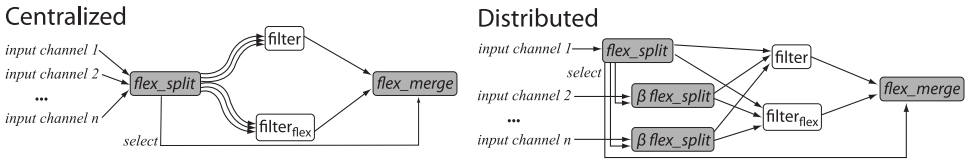


Fig. 11. Two alternative implementations of a flexible filter with n input channels.

stream and builds the *select* bitstream based on how much queue space is available downstream. If multiple copies of *flex_split* check for queue space separately at slightly different times, they may get different answers, and thus the data tokens in the input streams would be mismatched. Figure 11 illustrates two possible solutions. One option is to create one centralized *flex_split* that monitors all of the input queues for the copies of the flexible filter (pseudocode in Algorithm 3) and then splits the data stream in a way that is consistent across all input streams. The downside of this approach is that it may create a bottleneck in processing.

ALGORITHM 3: Centralized flex_split (Input: streams in_0, $in_1, \ldots, in_n$; Output: stream $out0_0$, $out0_1, \ldots, out0_n$, $out1_0$, $out1_1, \ldots, out1_n$, *select*)

```
pop data blocks b_0, b_1, ... 3 b_n from in_0, in_1, ..., in_n {Assumes that this stream filter will block
until there is adequate data on each input channel to supply one data block of size |b|.}
n0 ← min(avail(out0_i) over all i, |b|)
n1 ← |b| − n0
for i = 0 to n0 − 1 do
  push 0 to select
  for j = 0 to n do
    push b_j[i] to out0_j {where b_j[i] denotes the i^th token from b_j}
  end for
end for
for i = n0 to |b| − 1 do
  push 1 to select
  for j = 0 to n do
    push b_j[i] to out1_j
  end for
end for
```

Another approach is to introduce a second version of the *flex_split* implementation, denoted *βflex_split* (pseudocode in Algorithm 4). The original *flex_split* is used for the first channel, and then instead of building new *select* bitstreams, *βflex_split* filters reuse the original *flex_split*'s *select* stream and wait for sufficient space on their output queues before proceeding. This approach avoids forcing all of the input channels

ALGORITHM 4: βflex_split (Input: streams in, $select$; Output: stream $out0$, $out1$)

```
pop i from select
pop token t from in
if i is 0 then
  push token t to out0
else
  push token t to out1
end if
```

through a bottleneck but may result in extra stalling by the new *βflex_split* filters. Conceptually, *βflex_split* mirrors *flex_merge*, since they both follow the same *select* bitstream to guarantee the correct ordering of tokens.

3.2.2. Implementation of Backpressure. Streaming programming languages typically abstract away the backpressure mechanism that is implemented at the lower level of the intercore communication stack [Barnes et al. 1993; Thies et al. 2001]. Hence, programmers need not worry about the current state of the buffers between stream functions and can focus on the computational aspects of the algorithm and data manipulation through higher-level functions, such as *push* and *pop*. At the same time, the underlying message-passing API functions that support the handshake communication protocol and backpressure mechanism between communicating cores and that are often specific to the target architecture may also be made available to allow performance optimizations. Our implementation of *flex_split* and *flex_merge* rely on such functions. In particular, the *flex_split* implementation given in Algorithm 1 uses the *avail()* function that returns how much buffering space is available in the next core's buffer.[2] If the programmer does not use *avail()* to check the buffering availability of its output channels at runtime, then the filter will automatically stall whenever there is not sufficient space for the data to be sent on any of its output channels. Instead, using *avail()* to check the available space on a channel allows the programmer to dynamically send only the right amount of data to that channel and then proceed to the next instruction without stalling the filter. For instance, to avoid stalling when there is not enough space to send the entire block to f's primary copy, *flex_split* sends exactly the amount of data equal to $avail(out0)$ to $out0$. Then, the rest of the data is sent to f's secondary copy without calling *avail()* on this channel but relying instead on the underlying backpressure mechanism to regulate the stream $out1$. In our experience, relying on the implicit backpressure of the channel instead of explicitly checking *avail()* on $out1$ tends to produce better results, possibly because the leftover portion of the output stream can move forward faster to the filter's secondary copy in the presence of a temporary input buffering shortage.

4. MODEL FOR FLEXIBLE FILTERS

When exploring the search space for a good mapping, the performance for new mappings can be evaluated through either experimentation or simulation. Empirically implementing and testing a stream application on a target multicore platform can be very time consuming. The implementation is challenging for several reasons, including the usage of synchronization primitives and also low-level memory management such as direct memory access operations and data alignment. Moreover, most multicore platforms still lack sophisticated debugging tools, which further increases the time to design and test an application. For these reasons, a performance model may be preferable

[2]Though we refer specifically to the *avail()* function of the Gedae language, any stream language that exposes backpressure will provide a similar function.

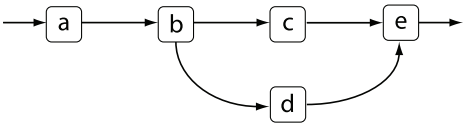


Fig. 12. Example of a stream program.

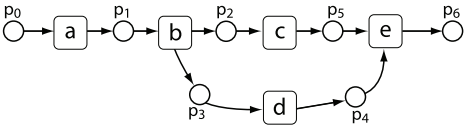


Fig. 13. The stream program of Figure 12 represented as a Petri net.

over actual experiments to facilitate rapid design-space exploration of different filter mappings. In this section, we describe the performance model we used for testing the flexible filter design methodology.

The model is based on the *Petri nets* model of computation [Petri 1962]. Petri nets are directed bipartite graphs with two kinds of nodes, *transitions* and *places*. Transitions may only be connected to places, and vice versa. A place is a container for tokens and starts with an initial *marking*, denoting how many tokens it initially contains. A transition is a node that connects places to each other and is able to *fire*, consuming tokens from each of the places on its incoming arcs and producing tokens on each of its outgoing arcs. A transition is only enabled to fire when all of the places on its incoming arcs have a sufficient number of tokens. Each arc is assigned a weight corresponding to the number of tokens that are produced or consumed during one firing event. The rate of production/consumption of tokens by the actors of a Petri net can be used to model the data-processing throughput of the components of a computing system. In addition to these constraints, we make the following assumptions.

—Firing semantics are discretized over time steps, and all transitions enabled at a given time step will fire.
—When nondeterminism is present in the graph (i.e., more than one transition could consume tokens from the same place, but the firing of one transition would preclude that of others), the transitions follow a round-robin priority schedule.
—Communication latency between cores is uniform across the system and is a function of the following message size.

$$latency = \alpha + \beta * message\ size, \tag{1}$$

where profiling of the target architecture determines the α and β factors.
—We assign the same memory capacity μ to each core c_i.

We now walk through the modeling of an example stream program, compositionally building the implementation constraints with modular constructs, including the mapping of tasks to cores, data buffering, communication latency, and flexibility. Figure 12 illustrates a simple example of a stream program consisting of five computational tasks which are composed with pipelined intertask communication. Each filter corresponds to a Petri net transition (transitions are shown as rectangles, while places are large empty circles, and tokens are small filled-in circles within the places). The execution time, e, of a transition corresponds to the measured or estimated execution time of the filter on the target hardware, and when a transition fires, it will not produce tokens until e time steps have passed. Since we evaluate throughput by simulation of the graph, irregular firing patterns are easily simulated by varying the execution time of a transition. Pipelined communication paths and dependencies are illustrated as directed edges in the figures, and places with an initial marking of zero are inserted between transitions. Figure 13 illustrates the full set of places and arcs in the Petri net representation of this example. In the case of feedback loops within the stream program, it is necessary to insert tokens so that the Petri net can make progress. A minimum of one token must be present in the initial marking of every cycle in the

original graph, but more may be inserted depending on the initialization of filters in the real program.

4.0.3. Mutual Exclusion. To capture the fact that each core only works on one filter at a time, mutual exclusion (mutex) constructs are added for all filters co-mapped to the same core. A mutex construct consists of a place initialized with one mutex token, such that a filter must consume this token in order to fire and will not return the token until it is done executing. Figure 14 adds a mutex construct between filters a and b. Our simulator enforces a round-robin priority scheme when more than one filter is simultaneously enabled but requires the same mutex token.

4.0.4. Data Buffering in Pipeline Communication. When two neighboring (pipelined) filters are co-mapped on the same core, data may be passed between these filters via in-place buffering (i.e., in the core's local memory) and does not incur additional storage cost. Consequently, there is never backpressure due to buffering between co-mapped neighbors. However, when neighboring filters are not co-mapped, a data buffer of finite size is maintained between them. We model backpressure, which is related to the available buffer space, by adding a backwards arc and place between tasks. The place is initialized with q tokens for a buffer of q size. The figures of this section follow the convention of representing backpressure edges as dashed lines. In Figure 15, four backpressure edges are added among filters b, c, d, and e. In this example, all back edges have the same number q of tokens, though uniform buffer sizing is not mandatory.

4.0.5. Communication Latency. The model incorporates the cost of communication through additional transitions, shown as darkened rectangles in Figure 16. These transitions add additional latency based on the size of data being passed and on the execution time of the tasks that follow them. The cost of pipeline communication may be hidden when the data movement is overlapped with computation. This is known as *double buffering*, a popular technique to optimize the execution of stream programs on multicore architectures [Chen et al. 2007]. However, if the execution time of a filter is relatively low compared to the latency of communication, the communication overhead may not be hidden. Notice that Figure 16 imposes mutual exclusion constraints on the communication transitions. The latency of these transitions corresponds to the communication overhead which is not hidden through double buffering. Multiple incoming communication streams do typically overlap, such as the two incoming communication transitions of filter e. In this case, it is not necessary to add a mutex loop to the second communication transition. Notice that the backpressure arcs bypass the communication overhead. This reflects the difference in latency between sending a block of data and a control message. Depending on the hardware platform, the properties of communication latency and how it changes with data size may vary; example communication latencies for the Cell processor are discussed in Section 5.4.2.

4.0.6. Flexibility. Flexible filters are modeled with the *flex_split* and *flex_merge* structures shown in Figure 17. (For clarity, places with an initial marking of zero and mutex loops to communication transitions are omitted from the rest of the figures.) The construction of *flex_split* has two transitions (shown shaded gray in Figure 17), one with an execution time of zero and one with a small positive execution time, ϵ. The difference enforces a permanent priority of the original copy of c over c_{flex}. Our implementation of *flex_merge* buffers incoming data from c and c_{flex} separately (see Section 3). The performance model abstracts this into a shared backpressure place where both tasks may consume tokens from the buffer. The overhead in latency and communication cost of *flex_split* and *flex_merge* is added to the *flex_merge* transitions (labeled v in Figure 17). Figure 18 depicts the overall model representation of the program from Figure 12, including mutual exclusion, buffering, communication overhead, and flexibility.

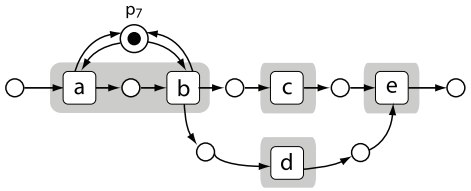


Fig. 14. Modeling mutual exclusion.

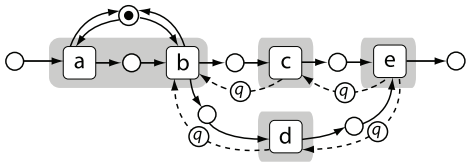


Fig. 15. Modeling backpressure.

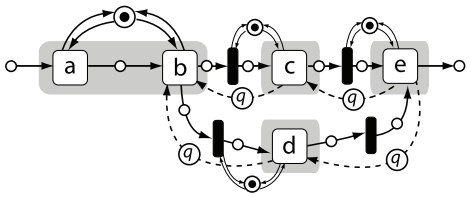


Fig. 16. Modeling communication overhead.

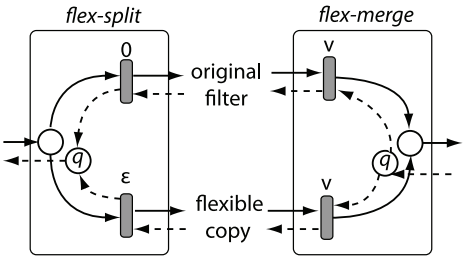


Fig. 17. Modeling *flex_split* and *flex_merge*.

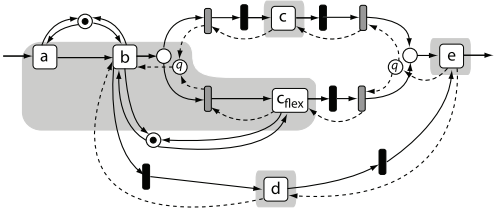


Fig. 18. Overall Petri net performance model representation of a stream program.

5. EXPERIMENTS

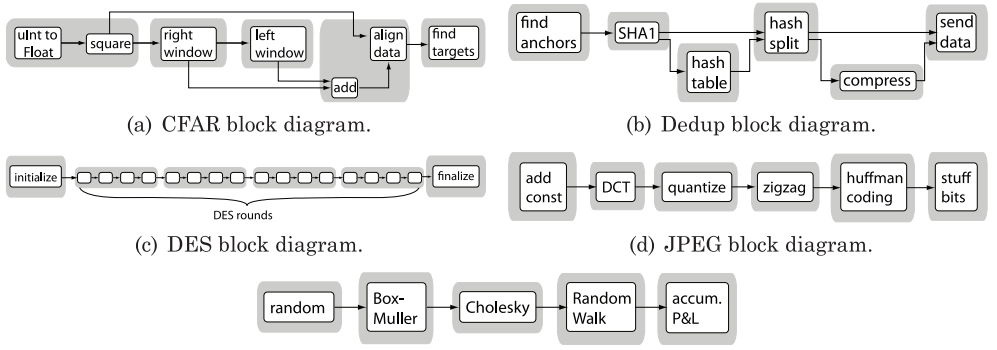
In this section, we evaluate flexible filters over several benchmarks on the Cell BE platform and then compare the application throughput observed in the experiments to the throughput derived by the flexible filter model presented in Section 4.

We performed all of our experiments on a Sony PlayStation3 (PS3) which hosts one Cell BE processor [Nanda et al. 2007]. The Cell architecture is a heterogeneous multicore system-on-chip originally designed for high-performance embedded applications [Kahle et al. 2005; Pham et al. 2005]. It features one PowerPC processing core called the PPU, eight synergistic SIMD processing units called SPUs, and the Element Interconnect Bus (EIB), an on-chip communication network capable of sustaining 205 GB/s of data transfers. Each SPU core has a 256KB local memory that is shared between code and data. The Cell processor is a good architecture for testing the performance of flexible filters since it exposes the trade-off between program code and data buffering when we make filters flexible. Because the PS3 architecture enables only six of the eight SPUs, this is the maximum number of cores used in our experiments.

To program the Cell, we took advantage of Gedae, a dataflow language that also provides an abstraction of the communication layer for our implementation by handling low-level details like direct-memory access (DMA) alignment and double buffering [Barnes et al. 1993; Lundgren et al. 2005]. Gedea provides a fully dynamic model for stream programming where stream filters can have either fixed or dynamic rates of data I/O. Gedae's API contains functions to implement communication channels, including an *avail()* function that gives information on how much space is available in a channel's input and output buffers.

Table II. Suite of Benchmark Applications to Test Flexible Filters

Benchmark Name	Field	Description
Constant False Alarm Rate Detection (CFAR)	Signal Processing	From the HPEC benchmark suite, CFAR identifies targets in a stream of incoming data given a noisy background, using an adjustable threshold value that changes based on the background noise so that the false alarm rate is constant [Haney et al. 2005].
Dedup compression	Information Theory	From the PARSEC benchmark suite, Dedup is a pipeline compression algorithm that breaks a file up into blocks according to the Rabin fingerprinting method and then compresses on a block-by-block basis, checking the hash of each block beforehand so duplicate blocks are only compressed once [Bienia et al. 2008].
DES encryption	Security	A block cipher algorithm.
JPEG encoder	Image Processing	Implements the baseline grayscale JPEG encoder [Gonzalez and Woods 2001].
Value-at-Risk (VAR)	Finance	Calculates the value-at-risk for a portfolio of stocks (or other assets) averaged over a number of random walks over a discrete number of time steps, assuming that the stocks change at each time step according to a random correlated set of moves .



(a) CFAR block diagram.

(b) Dedup block diagram.

(c) DES block diagram.

(d) JPEG block diagram.

Fig. 19. Block diagrams of benchmarks used for the experiments of Sections 4 and 5 together with their mapping on the IBM Cell multicore processor (the nonflexible case).

5.1. Benchmark Applications

By examining real-world benchmark applications, including several libraries which were incorporated into our implementations unmodified, we gain perspective on where flexible filters are most helpful as well as insight into practical implementation concerns. Table II lists several benchmarks that were implemented and tested with flexible filters. Figure 19 shows block diagrams of the filters of each benchmark together with how they are mapped to cores of the IBM Cell, and Figure 20 shows profile information for the filters of each benchmark based on their implementation without flexibility. The profiles represent the average execution time of each filter and is the value we use to determine which filters are bottlenecks. In the VAR benchmark, where the complexity of the Cholesky filter depends on the input size, the Cholesky filter profile represents the time when the input size is 128 stocks. In each benchmark, we map filters only to the SPU cores so that the testing environment is effectively homogeneous.

The Dedup, JPEG, and VAR benchmarks all include one bottleneck filter that is significantly more computationally expensive than the others. For example, Figures 21(a) and 21(b) show a Gedae trace table for the VAR benchmark before and after the

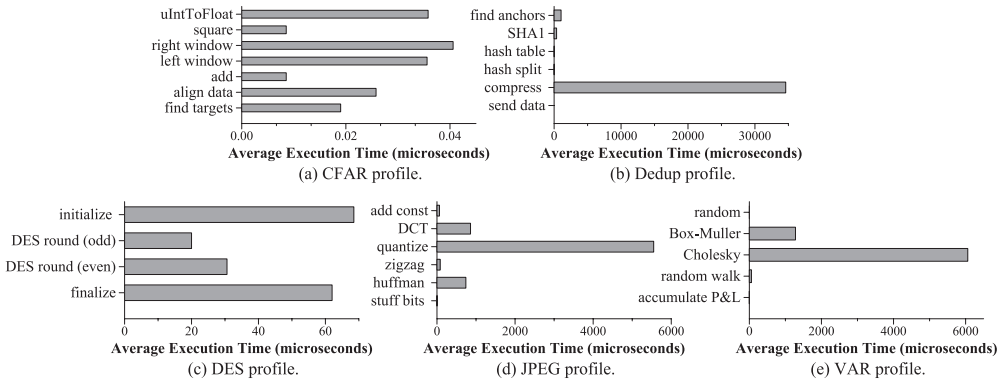
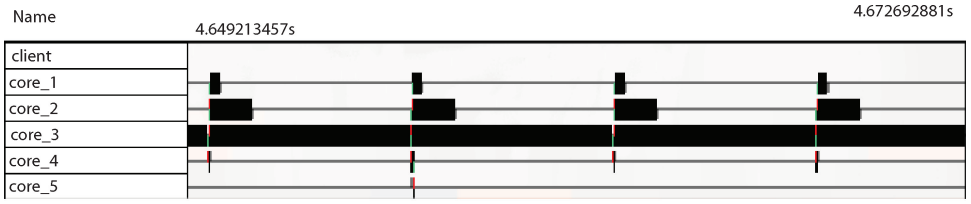
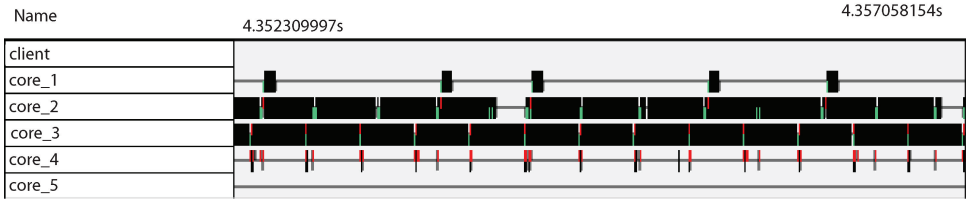


Fig. 20. Profile of tasks for each benchmark.



(a) Trace table of VAR without flexibility.



(b) Trace table of VAR with flexibility.

Fig. 21. Gedae trace tables of the VAR benchmark. A core's timeline is black when it is busy working on a task. Green and red marks show send and receive events.

Cholesky filter has been made flexible. In the trace table, the black rectangles show when a core is busy working on a task, and the smaller red and green rectangles show send and receive events. In Figure 21(a), core_3, which is assigned the Cholesky filter, is always working, while the other cores spend most of their time waiting for data to arrive. In Figure 21(b), core_2 assists core_3 and reduces its load. Notice that the timespan is actually different in the two timelines. When flexibility is added, we found it often necessary to reduce the overall data block granularity to reach optimal speedup. Therefore, compute tasks are broken up more frequently by send and receive events in Figure 21(b).

Table III reports the speedup gained in the benchmarks through the application of flexibility in cases where a flexible filter is present. This speedup reflects the performance gain compared to a parallel pipeline implementation. The DES benchmark is an example where there is no bottleneck. Even though there is some variation in the latencies of DES filters, once its eighteen filters are mapped to the six cores of the PS3, the load becomes fairly balanced, and adding flexibility to one or more of the filters does not benefit performance. In general, however, our results show that flexible filters

Table III. Speedup Results for Benchmarks Where One Bottleneck Filter is Made Flexible

Benchmark	Input Data	Speedup
CFAR	% targets/workload	
	7.3/16μs	1.45
	7.3/32μs	1.39
	7.3/63μs	1.47
	1.3/16μs	0.82
	1.3/32μs	1.06
	1.3/63μs	1.27
Dedup	Rabin block/max chunk size	
	4,096/512	2.00
DES	-	(no speedup)
JPEG	image width $\times$ height	
	128 $\times$ 128*	1.31
	256 $\times$ 256*	1.16
	512 $\times$ 512*	1.25
VAR	stocks/walks/timesteps	
	16/1,024/1,024	0.98
	32/1,024/1,024	1.34
	64/1,024/1,024	1.56
	96/1,024/1,024	1.54
	128/1,024/1,024	1.55
	160/1,024/1,024	1.81

*Individual benchmark images, each with different content.

provide speedup to an application whenever there is a bottleneck filter as long as the relative cost of communication to computation is not too high.

In some cases, a better implementation may alleviate the bottleneck. However, in reality, software is often designed using preexisting libraries. This was the case for the Dedup benchmark in these experiments. Our Gedae implementation invoked the same libraries as the original Dedup benchmark.[3]

When a filter is made flexible (one redundant copy) as described here, potential speedup is limited to a factor of two compared to the original parallel performance (the overall parallel speedup may be higher from pipelining). *Flex_split* could be extended to a three- or four-way split to take advantages of other available cores by increasing the data-parallelism of the bottleneck filter mapping. However, a few caveats on higher degree splits should be kept in mind: (1) in an architecture like Cell, where application code and data occupy the same memory, there may not be room for additional code and data buffers to accommodate the flexible filter, even if a core is not as busy with computation; (e.g., the hash table filter in the Dedup benchmark is not compute-intensive, but it is memory-intensive, and the less space available for building the table, the more times the downstream and compute-intensive compress filter must execute); (2) the pipeline nature of stream applications forces dependencies between the buffers of different filters, and adding extra channels within the stream may place higher buffering burdens on those parts of the stream graph.

Although in some cases, speedup is ideal, as with the Dedup benchmark in our experiments, it should be noticed that adding flexibility may not always achieve a full 2X speedup even if the bottleneck filter is much more expensive than its neighbors. The

[3]The hash table library required a minor modification in one of its constants so that the table would be guaranteed to fit within an SPU's local memory.

overhead of communication and changes in data block granularity required by flexible filters are additional costs of flexibility that can impact the performance speedup gained. Going by the profiles, we can sketch out the ideal flexible filter speedup based on the bottleneck filter and its immediate upstream neighbor. Consider the JPEG benchmark, where the DCT and quantize filters have execution times of 859 and 5,550 μs, respectively. The ideal throughput is approximately $\frac{2}{859+5550}$ tokens per microsecond. Compared to the throughput of quantize ($\frac{1}{5550}$), this is a speedup of approximately 1.73. Since the maximum observed speedup for the JPEG benchmark was only 1.31, overhead had a large impact for this benchmark. On the other hand, the ideal throughput estimate for the VAR benchmark (for an input size of 128 stocks) is 1.65, which is closer to the observed speedup of 1.55. Section 5.2 evaluates the impact of communication overhead on the CFAR benchmark in more detail.

5.2. Balance of Communication vs. Computation

This section explores the balance of communication and computation with respect to the speedup gained by adding flexibility to an application. Adding flexibility to a stream filter typically adds more communication overhead compared to the original pipeline implementation, because *flex_split* and *flex_merge* require additional buffers, which reduce the space per buffer available on the cores. This may translate to data blocks with fewer data tokens and thus slightly higher communication overhead. There is also additional data movement between the buffers of the filter and *flex_split* and *flex_merge*, even when some are co-located on the same core. The experiments in Figure 22 synthetically vary the latency of the Cholesky and BoxMuller filters in a subset of the VAR benchmark. By increasing the execution time of the filters while the relative ratio between them stays the same, these tests examine how the speedup changes as the relative cost of communication and computation changes. The flexible copy of Cholesky is mapped to the same core as BoxMuller, and the latency of Cholesky is approximately three times that of BoxMuller so that the optimal speedup in any case possible is about 50% (shown with a dashed line in Figure 22). The speedup approaches 50% as the computation cost of the filters becomes very large, overshadowing the cost of communication. At the other end of the spectrum, speedup drops off as the execution time of the bottleneck filter, Cholesky, is reduced. When the execution time of the Cholesky filter is less than 560 microseconds, no improvement is observed. The point at which speedup tapering-off occurs in this benchmark is a result not only of the execution latency of the bottleneck filter, but also of the data block size (vector of 128 floats) and task granularity (3 data blocks in the flexible case, 50 data blocks in the nonflexible case).

5.3. Adapting to Data-Dependent Flow

One of the strengths of flexible filters is that they can adapt load dynamically at runtime when there are data-dependent spikes of activity that may cause temporary bottlenecks in the flow of execution. The CFAR benchmark provides an example to explore data-dependent flow volume. As shown in Figure 20(a), all filters of CFAR have a relatively lightweight execution time with respect to the communication overhead, and initially, the program did not benefit from the addition of flexibility. In particular, the *find targets* filter in our original implementation does no additional work after a target is detected, and so has relatively constant execution time regardless of the content of the data stream. However, in practice, it is possible that once a target is found, additional processing, such as target classification and tracking is needed [Novak et al. 1997]. To capture this fact, we add an additional synthetic workload to *find targets* every time a target is detected in the CFAR experiments reported in Table III. Since

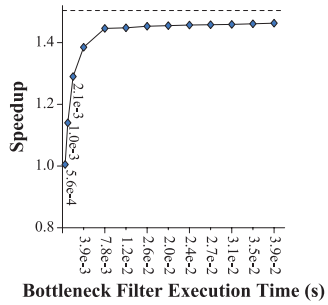


Fig. 22. Speedup as the relative cost of a bottleneck filter increases with respect to the cost of communication.

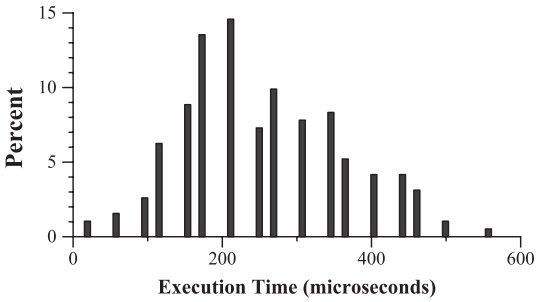


Fig. 23. Histogram of workload per 114 cells, with % targets/workload = 7/32μs.

Table IV. Size of the Flexible Filter Model Graph Compared to the Original Task Graph

Benchmark	(Orig) Tasks	(Orig) Arcs	Transitions	Places
VAR	5	4	9	11
VAR-flexible	5	4	15	21

the location of targets is data dependent and may not be uniformly distributed in the stream, the workload of *find targets* may change dynamically, and spikes in the number of targets detected could cause bottlenecks. The percent of targets detected is varied by adjusting the sensitivity threshold for the input data sets provided by the HPEC challenge [Haney et al. 2005]. Figure 23 plots a histogram of the time it takes to process a data block of 114 cells, where 7% of the cells are targets, and an additional workload of 32 microseconds is added for each target. The speedup gained by applying flexible filters in this case depends both on the percentage of data tokens that require extra processing and on the amount of extra work required. While it would likely be possible to achieve similar results for any one instance of CFAR with a static stream split and enough buffering, the strength of flexible filters is that the same implementation will adapt to changing load in the same stream without modification (i.e., a stream that switches from one distribution of execution times to another).

5.4. Model Experiments

For the remainder of the experimental section, we evaluate the compositional performance model for flexible filters to understand how closely it approximates actual experiment data collected. The experiments draw from the set of benchmarks described in Section 5.1 as well as synthetic benchmarks generated with the Task-Graphs for Free (TGFF) tool [Dick et al. 1998]. The performance on given instances is estimated using a Petri-Net simulator that tracks the movement of tokens around the graph as transitions become enabled and fire. As described in Section 4, the model adds new transitions and arcs to the task graphs of applications. Table IV reports the total number of transitions and places in the experiments for the VAR benchmark.

5.4.1. Mutual Exclusion. Figure 24 illustrates how effectively the model captures mutual exclusion. The benchmarks tested are synthetic stream graphs generated by TGFF and implemented with Gedae for the Cell processor. The mappings tested are illustrated in Figure 25. TGFF provides relative execution times of the different tasks. For this experiment, they are set large enough so that the communication latency is completely absorbed into double buffering in the communication channels. The communication buffers between the pipelined stream tasks are also set large enough so that

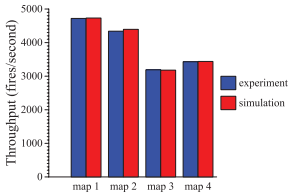


Fig. 24. Estimated vs. actual throughput testing mutual exclusion.

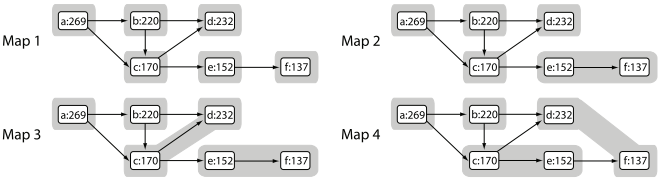


Fig. 25. Estimated vs. actual throughput testing alternative mapping options.

backpressure does not impact performance and the model of mutual exclusion is highlighted by the results. For all mappings, the flexible filter model's simulated throughput came within 2% of the throughput of the experimental model.

5.4.2. Communication Latency. Based on the Cell BE profiling experiments of Kistler et al. [2006], we estimate constants α and β from Equation (1) and use the following equation,

$$latency(nanoseconds) = 91 + 0.04 * message\ size\ (bytes), \tag{2}$$

to calculate the latency of a DMA data transfer between cores. The effective latency added by a communication operation in a pipeline communication, accounting for double buffering, may be calculated with Equation (3) where the computational operation has execution time t.

$$pipeline\ latency = max(latency(message\ size) - t, 0). \tag{3}$$

Equations (2) and (3) provide a lower bound for the latency of programs running on the Cell written with the Cell SDK. In practice, we observed larger latencies in the benchmarks run on the Cell on top of Gedae, which adds more runtime operations. Latency is better approximated for Gedae applications using Equation (4).

$$latency(microseconds) = 56 + 0.15 * message\ size\ (bytes). \tag{4}$$

Note that the time units of Equation (4) are in microseconds instead of nanoseconds. It is likely that there are some computational operations taking place in the Gedae runtime environment that account for the increase, and there may also be some inefficiencies in the implementation of our benchmarks (e.g., how data is packed into 128-bit chunks).

Profile data collected from the flexible filter benchmarks reported in Section 5.1 populate the flexible filter model for the following sets of experiments (with the granularity remaining fixed when flexibility is added in the model). Figure 26 shows the comparison of the estimated and measured throughput for all of the VAR benchmark's input data sets when communication latency is not included in the model. The discrepancy between the simulator and experiments is largest for the smallest portfolio size which is equal to 16. Figure 27 plots the same data when communication latency is included in the flexible filter model, using the communication latency estimate from Equation (4). Most of the simulations are fairly accurate, with greater accuracy in the data points which correspond to larger portfolio size. The benchmark requires steps that grow with the square of the portfolio size. Thus, with the smaller portfolio sizes, the cost of communication plays a greater role.

5.4.3. Flexibility. Figure 28 shows how well the simulator predicts trends of speedup when flexible filters are applied to the stream programs for the CFAR, JPEG, and VAR benchmarks. In most cases, the simulation accurately captures trends in performance gains when flexibility is added to a benchmark. The CFAR benchmark results demonstrate the largest differences when comparing the simulation and experiments. Notice

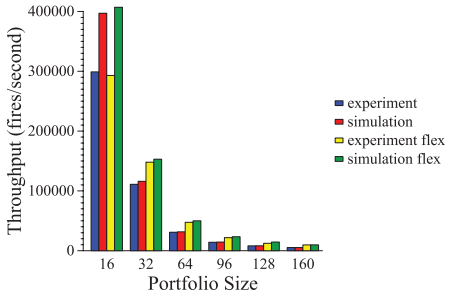


Fig. 26. Estimated vs. actual throughput for VAR (no communication latency.)

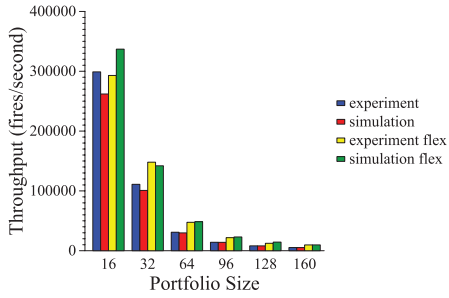


Fig. 27. Estimated vs. actual throughput for VAR.

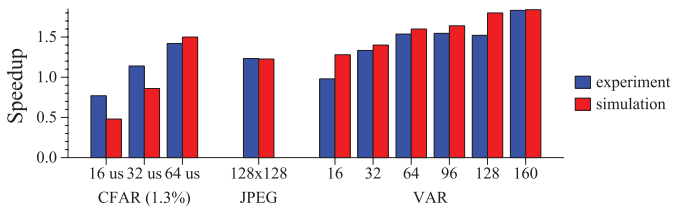


Fig. 28. Estimated vs. actual speedup across several benchmarks.

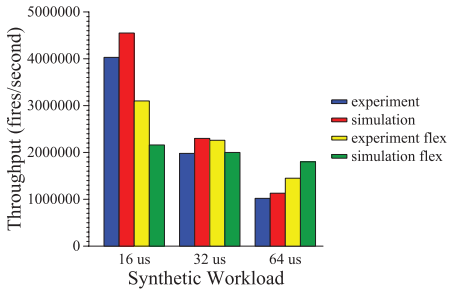


Fig. 29. Estimated vs. actual throughput for CFAR.

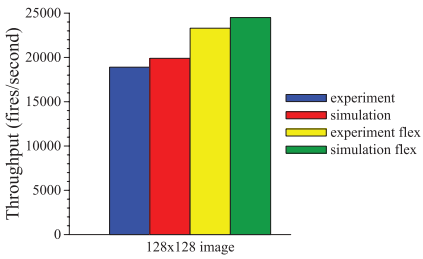


Fig. 30. Estimated vs. actual throughput for JPEG.

that CFAR presents challenges to simulation since it is a data-dependent benchmark and its filters have a fairly light workload, and thus communication plays a bigger role. Figure 29 and Figure 30 report the estimated and experimental throughput for the CFAR and JPEG benchmarks that Figure 28 is based on.

6. RELATED WORK

Work Stealing. Flexible filters balance load using a version of work stealing for stream programs. Work stealing is a technique used in a variety of parallel systems to balance load by allowing idle cores to "steal" tasks from busy cores [Bender and Rabin 2002; Frigo et al. 1998; Kakulavarapu et al. 2001]. Most work-stealing techniques go through stages of load evaluation, reassignment, and task migration, and their "victim" processors (from whom tasks will be stolen) are selected randomly. In contrast, flexible filters do not steal randomly but use the knowledge that neighbors of a bottleneck filter will be idle because they depend on this filter to continue processing data tokens. Items are never migrated between buffering queues of different processors; instead, when queues become full, new items are redirected elsewhere. With flexible filters, tasks

are not "stolen" per-se, but rather the dataflow is rerouted when a bottleneck arises. Whichever filters have been mapped with flexibility determine the available routes for data during runtime. Flexible filters are specialized to pipeline dataflow because the pipeline stream dependencies narrow down good candidates for redundant-code placement by exposing which tasks will become idle when another becomes a bottleneck.

Load Balancing with Pipeline Parallelism. Load balancing approaches specific to stream programs can be categorized depending on whether the stream models rely on data parallelism or pipeline parallelism (in practice both approaches can be used simultaneously [Gordon et al. 2006]). In data parallel stream systems, there can be many producers that feed many consumers, and there may be many instances of producer and consumer functions [Arpaci-Dusseau et al. 1999; Ranger et al. 2007]. Load balancing is achieved by routing data to different instances of consumers based on their current load and productivity. On the other hand, in pipeline-parallel stream systems, the data may need to flow through a series of pipelined filters, where each filter can be viewed as a producer and consumer of input and output data. The order of filters constrains the order in which tasks may be executed.

Flexible filters are a solution for load balancing in pipeline-parallel stream programs. Many related works addressing this problem involve a central control and/or phases where the compute nodes collect statistics which are used by the control to direct reorganization [Fellheimer 2006; Shah et al. 2003; Xing et al. 2005]. The number of filters is designed to outnumber the number of cores, and load balancing is typically achieved by moving filters from nodes with heavy loads to nodes with lighter loads, similar to work stealing. Flexible filters simulate filter migration by duplicating some filters on the cores and invoking duplicates when the load becomes unbalanced.

Combining Static and Dynamic Scheduling. Several works explore the combination of static and dynamic scheduling. There is a great variety in the possibilities in this space. With respect to these works, if there is a scale based on the weight of the dynamic online scheduler, flexible filters exists at the lightweight end of the spectrum. These techniques complement each other and are combined for cumulative benefit.

Flextream is a compilation framework that combines static and dynamic scheduling to adapt to changing resource availability. The static scheduler creates a schedule for a virtual architecture, and the dynamic scheduler adapts at runtime to the actual architecture where the program is running by adjusting filter mappings, schedule, and memory allocation [Hormati et al. 2009].

Zhang et al. designed a stream programming framework for the Cell processor based on StreamIt, which takes advantage of the Cell's high memory bandwidth by maintaining filter buffers in main memory instead of the individual cores' local memories and dynamically swaps filters to the cores at runtime based on the overall progress of the program. In this framework, filters must declare their rates in advance, and during static offline scheduling, the compiler creates a steady state schedule based on the rates [Zhang et al. 2008].

Chen et al. perform load balancing for stream programs by compiling several alternative filter mappings [2005]. During runtime, the system can "context-switch" between the alternatives based on the properties of the data. Flexible filters, on the other hand, dynamically adapt to the current flow behavior of the system.

In the DIAMOND system, data tokens are forwarded based on threshold values in the input and output queues [Huston et al. 2005]. Load balancing with flexible filters, similarly, is an outcome of the state of the queues but differs in that flexible filters balance the load based on backpressure. Moreover, DIAMOND is optimized for distributed search which relaxes several constraints of stream programs, namely, that the filters

need not be executed in a particular order because they are used to eliminate unwanted data (rather than transform the data) and that data can be processed in any order.

Split and Join Operations. Many stream programming languages, such as StreamIt include *split* and *join* nodes in their supporting library that are used to transform the stream programs [Fellheimer 2006; Gordon et al. 2006; Thies et al. 2001]. *Split* and *join* nodes in StreamIt can be used in two ways. First, the programmer may use them while writing a new stream program. Second, the StreamIt compiler may introduce *split* and *join* nodes to optimize the program by increasing data parallelism. This accomplishes static load balancing because the dataflow is split at runtime regardless of the loads on the various cores. In contrast, the Flexible-Filter *flex_split* and *flex_merge* filters described in Section 3 are not intended for use when building a stream program but are application-independent library filters that are introduced at a later stage when flexibility is added. Dynamic load balancing in our approach is based only on the insertion of *flex_split* and *flex_merge*. These are statically added during compilation but achieve dynamic load balancing via the backpressure mechanism applied to the dataflow.

Modeling Dataflow. The performance model for flexible filters described in Section 4 which uses Petri Nets is based on a combination of previous works, many of which use synchronous dataflow (SDF) to model the flow of data. SDF represents a subset of Petri nets with fixed rates of data production and consumption [Lee and Messerschmitt 1987]. Static analysis can be used to derive the maximum sustainable throughput of a dataflow graph [Karp 1978; Dasdan and Gupta 1998]. An extensive body of literature exists on throughput analysis of SDFs; here we mention a few that are relevant to this research. The modeling of schedulers by Wiggers et al. [2007] and Staschulat and Bekooij [2009] can be used to represent precise arbitrated communication channels, and resource trade-offs are evaluated by Moreira et al. [2005] and Stuijk et al. [2006].

Modeling mutual exclusion by adding additional arcs to the SDF is an approach used in several previous works [Poplavko et al. 2003; Moreira and Bekooij 2007]. In particular, our compositional performance model for flexible filters is based on work by Bonfietti et al., who modify the SDF to include multicore mapping by the addition of arcs in order to create a cycle between filters mapped to the same core such that the cycle has only one token since only a single filter can execute one the core at a time [2010, 2009]. The mutex loops added to the flexible filter model allow arbitrary order of execution of the filters, depending on when they are enabled. The technique of modeling buffer space with the addition of back edges and tokens to the SDF has been also applied in several previous works including [Poplavko et al. 2003; Hölzenspies et al. 2007; Moreira and Bekooij 2007].

7. CONCLUSIONS

Stream processing is a promising paradigm for programming multicore systems for high-performance embedded applications. Flexible filters combine static mapping of stream program tasks with dynamic load balancing of their execution in order to improve system-level processing throughput of the program when it is executed on a distributed-memory multicore system as well as the local (core-level) memory utilization. The flexible filters technique is scalable because it is based on distributed point-to-point handshake signals exchanged between neighboring cores. Flexibility may be applied to any stateless filter without any modification to the filter itself, and only altering the overall stream program with the addition of the application-independent auxiliary filters *flex_split* and *flex_merge* around the filter and its flexible duplicate. The experiments in this article apply flexible filters to five stream benchmarks and achieve performance speedup higher than 30% in most cases.

The flexible filter model performs well in matching the absolute throughput and performance trends for different mappings and flexibility assignments. In future work, more research to understanding how buffering and granularity impacts performance in the system would be necessary, both experimentally and in the flexible filter model. In these tests, the buffer sizes and granularity for each filter was selected manually through trial and error. An algorithm for determining the best buffer and granularity through the performance model could simplify and automate this design stage.

ACKNOWLEDGMENT

We would like to thank IBM for providing us with access to Cell-Blade servers and Gedae, Inc., for granting us a software license through their university program.

REFERENCES

ARPACI-DUSSEAU, R. H., ANDERSON, E., TREUHAFT, N., CULLER, D. E., HELLERSTEIN, J. M., PATTERSON, D., AND YELICK, K. 1999. Cluster I/O with River: Making the fast case common. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*. 10–22.

BARNES, K. B., CHEN, Y. N., LUNDGREN, W. I., PRIDMORE, J. S., RIVERA, J. A., SCHAMING, W. B., AND TOOMBS, L. E. 1993. Data flow graph-programming environment for embedded multiprocessing. *Proc. SPIE,* Vol. 1957, 297–304.

BENDER, M. A. AND RABIN, M. O. 2002. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory Comput. Syst. 35,* 3, 289–304.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 72–81.

BONFIETTI, A., BENINI, L., LOMBARDI, M., AND MILANO, M. 2010. An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 897–902.

BONFIETTI, A., LOMBARDI, M., MILANO, M., AND BENINI, L. 2009. Throughput constraint for synchronous data flow graphs. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. 26–40.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. 777–786.

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference on Innovative Data Systems Research*. 668–668.

CHEN, J., GORDON, M. I., THIES, W., ZWICKER, M., PULLI, K., AND DURAND, F. 2005. A reconfigurable architecture for load-balanced rendering. In *Proceedings of the SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hardware*. 71–80.

CHEN, T., SURA, Z., O'BRIEN, K., AND O'BRIEN, J. K. 2007. Optimizing the use of static buffers for DMA on a CELL chip. In *Proceedings of the International Conference on Languages and Compilers for Parallel Computing*. 314–329.

COLLINS, R. L. AND CARLONI, L. P. 2009. Flexible filters: Load balancing through backpressure for stream programs. In *Proceedings ACM International Conference on Embedded Software (EMSOFT)*. 205–214.

COLLINS, R. L. AND CARLONI, L. P. 2010. Flexible filters for high-performance embedded computing. In *Proceedings of the High Performance Embedded Computing Workshop*.

DÄCKER, B. 2000. Concurrent functional programming for telecommunications: A case study of technology introduction. In Licentiate Thesis, KTH Royal Institute of Technology.

DASDAN, A. AND GUPTA, R. K. 1998. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Trans. Comput. Aid. Design Integr. Circuits Syst. 17,* 10, 889–899.

DICK, R., RHODES, D., AND WOLF, W. 1998. TGFF task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Co-Design (CODES)*. 97–101.

FELLHEIMER, E. T. 2006. Dynamic load-balancing of StreamIt cluster computations. M.S. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN Conference on Program Language Design and Implementation*. 212–223.

GEDIK, B., ANDRADE, H., WU, K.-L., YU, P. S., AND DOO, M. 2008. Spade: The system S declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1123–1134.

GONZALEZ, R. C. AND WOODS, R. E. 2001. *Digital Image Processing*. Addison-Wesley Longman Publ. Co., Inc., Boston, MA.

GORDON, M. I., THIE, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Oper. Syst. Rev. 40,* 5, 151–162.

GRAEF, A., KERSTEN, S., AND ORLAREY, Y. 2006. DSP programming with Faust, Q and Supercollider. In *Proceedings of the Linux Audio Conference*.

GUMMARAJU, J. AND ROSENBLUM, M. 2005. Stream programming on general-purpose processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 343–354.

HANEY, R., MEUSE, T., KEPNER, J., AND LEBAK, J. 2005. The HPEC challenge benchmark suite. In *Proceedings of the High-Performance Embedded Computing Workshop*.

HÖLZENSPIES, P. K. F., SMIT, G. J. M., AND KUPER, J. 2007. Mapping streaming applications on a reconfigurable mpsoc platform at run-time. In *Proceedings of the International Symposium on System-on-Chip (SoC '07)*. 74–77.

HORMATI, A., CHOI, Y., KUDLUR, M., RABBAH, R. M., MUDGE, T. N., AND MAHLKE, S. A. 2009. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 214–223.

HUSTON, L., NIZHNER, A., PILLAI, P., SUKTHANKAR, R., STEENKISTE, P., AND ZHANG, J. 2005. Dynamic load balancing for distributed search. In *Proceedings of the International Symposium on High Performance Distributed Computing*. 157–166.

KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. 2005. Introduction to the cell multiprocessor. *IBM J. Res. Develop. 49,* 4-5, 589–604.

KAKULAVARAPU, P., MAQUELIN, O., AMARAL, J. N., AND GAO, G. R. 2001. Dynamic load balancers for a multithreaded multiprocessor system. *Parallel Process. Lett. 11,* 1, 169–184.

KAPASI, U. J., RIXNER, S., DALLY, W. J., KHAILANY, B., AHN, J. H., MATTSON, P., AND OWENS, J. D. 2003. Programmable stream processors. *IEEE Computer 36,* 8, 54–62.

KARP, R. M. 1978. A characterization of the minimum cycle mean in a digraph. *Discrete Math. 23,* 3, 309–311.

KISTLER, M., PERRONE, M., AND PETRINI, F. 2006. Cell multiprocessor communication network: Built for speed. *IEEE Micro 26,* 3, 10–23.

KUDLUR, M. AND MAHLKE, S. 2008. Orchestrating the execution of stream programs on multicore platforms. *ACM SIGPLAN Not. 43,* 6, 114–124.

LEE, E. AND MESSERSCHMITT, D. 1987. Synchronous data flow. *Proc. IEEE 75,* 9, 1235–1245.

LUNDGREN, W., STEED, J., AND BARNES, K. 2005. Integrating the hardware description with gedae's single sample language to generate efficient code. In *Proceedings of the Electro Magnetic Remote Sensing Defence Technology Centre Conference*.

MCCOOL, M. D. 2006. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proceedings of the GSPx Multicore Applications Conference.*

MINSKY, Y. AND WEEKS, S. 2008. CAML trading - experiences with functional programming on Wall Street. *J. Functional Program. 18,* 4, 553–564.

MOREIRA, O. AND BEKOOIJ, M. 2007. Self-timed scheduling analysis for real-time applications. *EURASIP J. Adv. Signal Process.*

MOREIRA, O., MOL, J.-D., BEKOOIJ, M., AND VAN MEERBERGEN, J. 2005. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Proceedings of the IEEE Real Time on Embedded Technology and Applications Symposium*. 332–341.

NANDA, A. K., MOULIC., J. R., HANSON, R. E., GOLDRIAN, G., DAY, M. N., D'ARNORA, D. B., AND KESAVARAPU, S. 2007. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. Res. Develop. 51,* 5, 573–582.

NOVAK, L. M., OWIRKA, G. J., BROWER, W. S., AND WEAVER, A. L. 1997. The automatic target-recognition system in SAIP. *Lincoln Lab. J. 10,* 2, 187–202.

PETRI, C. A. 1962. Kommunikation mit automaten ("communication with automata"). Ph.D. thesis, Darmstadt University of Technology.

PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. N., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., STASIAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D.,

YAMAZAKI, T., AND YAZAWA, K. 2005. The design and implementation of a first-generation CELL processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. 184–185.

POPLAVKO, P., BASTEN, T., BEKOOIJ, M., VAN MEERBERGEN, J., AND MESMAN, B. 2003. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 63–72.

RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. 2007. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the Symposium on High Performance Computer Architecture*. 13–24.

SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the International Conference on Data Engineering*. 25–36.

STASCHULAT, J. AND BEKOOIJ, M. 2009. Dataflow models for shared memory access latency analysis. In *Proceedings of the International Conference on Embedded Software*. 275–284.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference* 899–904.

STUIJK, S., GEILEN, M., THEELEN, B. D., AND BASTEN, T. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation*. 404–411.

THIES, W., KARCZMAREK, M., GORDON, M., MAZE, D., WONG, J., HOFFMANN, H., BROWN, M., AND AMARASINGHE, S. 2001. StreamIt: A compiler for streaming applications. Tech. rep., MIT-LCS Technical Memo TM-622, MIT, Cambridge, MA.

WIGGERS, M. H., BEKOOIJ, M. J. G., AND SMIT, G. J. M. 2007. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*. 11–22.

XING, Y., ZDONIK, S., AND HWANG, J.-H. 2005. Dynamic load distribution in the Borealis stream processor. In *Proceedings of the International Conference on Data Engineering*. 791–802.

ZHANG, D., LI, Q. J., RABBAH, R., AND AMARASINGHE, S. 2008. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News 36*, 18–27.

Received July 2011; revised January, April 2012; accepted July 2012