Flexible Filters: Load Balancing through Backpressure for Stream Programs

Rebecca L. Collins and Luca P. Carloni
Department of Computer Science, Columbia University, New York, NY 10027
[rlc2119,luca]@cs.columbia.edu

ABSTRACT

Stream processing is a promising paradigm for programming multi-core systems for high-performance embedded applications. We propose flexible filters *as a technique that combines static mapping of the stream program tasks with dynamic load balancing of their execution. The goal is to improve the system-level processing throughput of the program when it is executed on a distributed-memory multi-core system as well as the local (core-level) memory utilization. Our technique is distributed and scalable because it is based on point-to-point handshake signals exchanged between neighboring cores. Load balancing with flexible filters can be applied to stream applications that present large dynamic variations in the computational load of their tasks and the dimension of the stream data tokens. In order to demonstrate the practicality of our technique, we present performance improvements for the case study of a JPEG encoder running on the IBM Cell multi-core processor.*

Categories and Subject Descriptors

D.1.3 [Programming Techniques] Concurrent Programming; D.3.2 [Programming Languages] Data-flow languages.

General Terms

Design, Performance.

Keywords

Stream programming, dynamic load balancing.

1. INTRODUCTION

Stream processing [4, 15, 20, 25] is a promising model for programming multi-core system-on-chip platforms that is applicable to a wide range of applications including high-performance embedded applications, signal processing, image compression, and continuous database queries [5, 24]. The basic idea of stream processing is to decompose 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 pass through them. Filters communicate with each other explicitly by exchanging the 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. For instance, high-performance implementations can often be obtained by mapping the filters on a pipeline of processing cores that communicate via a message-passing protocol and queue buffers.

Previous works have shown how filters can be mapped to the actual cores to balance the load and optimally utilize the available resources [11, 17]. For example, if there are more filters than cores, several filters may be mapped to a single core, and if there are more cores than filters, stateless filters may be replicated on several cores. In some systems, the cores perform context-switching across several filters based on priority queues. In general, however, it remains a challenge to achieve an optimal mapping that maximizes the stream processing throughput while accounting for the data dependencies among the filters and the available hardware resources (processing cores, memories, and interconnect). Coarse-grained distributions of the filters across the cores can result in uneven processing loads while fine-grained distributions may lead to inefficiencies due to the overhead of synchronization and data transfers among the filters.

In this paper we propose *flexible filters* as a technique to implement stream programs on distributed-memory multi-core platforms that combines 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 run time 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 upstream or downstream filters, or both, may start suffering a loss of throughput and, ultimately, this affects the data processing throughput of the overall implementation. If the bottleneck is caused by a long-latency computation that delays the production of new tokens, the downstream filters may become idle due to the lack of inputs. But even when it continues to produce new tokens, a filter may also become a bottleneck when it cannot sustain the processing rate of the upstream filters. If this is the case, the input buffers of its processing core start filling up. This ultimately leads to the emission of *backpressure* signals that are sent back to the cores running the upstream

EMSOFT'09, October 12–16, 2009, Grenoble, France.
Copyright 2009 ACM 978-1-60558-627-4/09/10 ...$10.00.

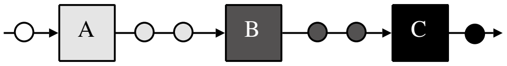


Figure 1: Example of Stream Program Structure.

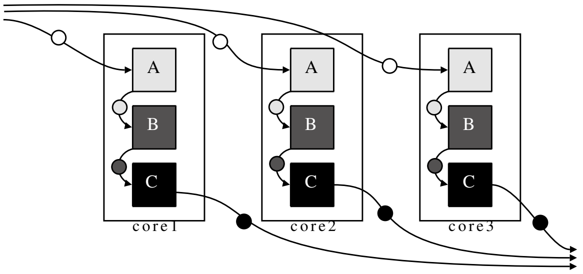


Figure 2: SPMD Mapping.

filters, which are forced to become idle to avoid a loss of data.

The basic idea of flexible filters is precisely to take advantage of the available cycles on these neighboring cores and use them to dynamically accelerate the execution of bottleneck filters. In other words potential bottleneck filters can be balanced by making their mapping to the underlying architecture "flexible" so that for certain periods they can run simultaneously on more than one processing core to execute different substreams of the data stream. This is achieved with the following procedure:

1. Identification of bottleneck filters through profiling of the application with the target multi-core platform (these filters must be stateless; i.e. given an input token x, a stateless filter will produce the same output token regardless of what tokens came before x);

2. Completion of a static *redundant* mapping that replicates the code of the bottleneck filters to deploy them on multiple, typically-neighboring cores;

3. Addition of auxiliary code that 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.

Flexible filters differ from many previous load-balancing approaches because the load balancing is based only on backpressure and the task reassignment to idle cores is guided by data dependencies across the filters in the stream program rather than random selection. No centralized control is required, and no extra messages are sent among cores beyond backpressure messages, which are already present to prevent the communication buffers from overflowing. Since load balancing is driven by the runtime load, 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, e.g. in Bloom filters applications.

2. FLEXIBLE FILTERS

Background. To implement a stream program on a multi-core architecture each of its filters must be mapped to at least one core. A core may host several filters and rely on a scheduler so that it is time-multiplexed among them while minimizing the context-switch overhead. We assume that the ordering of tokens must be preserved in the final output of the program (though the stream may be split and joined), and no tokens can be dropped without affecting the correct functionality of the program. The performance of a given implementation can be measured by its *maximum sustainable throughput (MST)*, i.e. the maximum rate at which data tokens can be processed under the assumption that the environment is always willing to produce new tokens and does not ever require the system to stall through a backpressure signal. Assuming an ideal multi-core architecture where the overhead of inter-core data communication and intra-core context switching is negligible and each core has unlimited local memory, an ideal mapping of filters would have the following properties: no core ever stalls and the MST scales linearly with the number of cores.

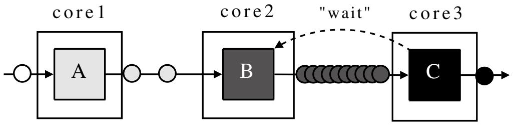


Figure 3: Baseline Pipeline Mapping.

Three Alternative Mappings. Flexible filters can be seen as a mapping approach that sits in between two other implementation techniques: *Single Program Multiple Data (SPMD) Mapping* and *Baseline Pipeline Mapping*.

Consider the simple example of a generic stream program whose structure is shown in Fig. 1: it consists of three filters A, B, and C with data tokens traveling between them on communication channels (A, B) and (B, C). An SPMD mapping of this program on an architecture with three cores is shown in Fig. 2: each core contains the entire program and the data stream is distributed evenly among them. If the filters have *latencies*[1] $L_A = 2$, $L_B = 2$, and $L_C = 3$, respectively, then the MST obtained with this mapping is $\frac{\# \ cores}{L_A+L_B+L_C} = \frac{3}{7} = 0.429$. This value corresponds to the ideal MST for the given architecture and can be theoretically reached by the SPMD mapping because no core receives data tokens from another core in this mapping. In practice, however, the local memory of a core is not unlimited, and it may not be able to accommodate the code of all of the filters. Even if it is possible, code for the filters' instructions reduces the amount of buffer space for data tokens, and may reduce the ability to overlap data transfer with computation (e.g. double-buffering). Furthermore, SPMD mappings require off-chip bandwidth to scale linearly with the number of cores. Therefore, we will use the theoretical SPMD implementation as a benchmark for the stream program's ideal throughput, but we will move on to implementations where each core does not necessarily contain the entire program. In particular, our goal is to achieve ideal MST (like SPMD mappings) using the fewest copies of filters possible in the overall system.

A baseline pipeline mapping is shown in Fig. 3: here, each filter is mapped to a separate core. Using the same latency values as above, this mapping delivers an MST equal to $\frac{1}{3} = 0.333$ because it is bound by the latency of filter C,

[1] The latency of a filter is the time necessary to execute it on a given core as a stand-alone task. In a heterogeneous multi-core architecture the same filter would have different latencies when executed on different cores. However, in this example we assume that cores are mapped only on homogeneous architectures.

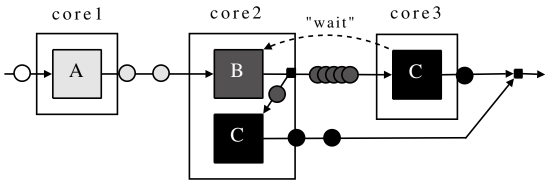


Figure 4: Flexible-Filter Mapping.

Time Steps		t_0	t_1	t_2	t_3	t_4	t_5	t_6
core$_1$	Step	$A_{0,0}$	$A_{0,1}$	$B_{0,0}$	$B_{0,1}$	$C_{0,0}$	$C_{0,1}$	$C_{0,2}$
	Block(s)	0	0	0	0	0	0	0
core$_2$	Step	$A_{1,0}$	$A_{1,1}$	$B_{1,0}$	$B_{1,1}$	$C_{1,0}$	$C_{1,1}$	$C_{1,2}$
	Block(s)	1	1	1	1	1	1	1
core$_3$	Step	$A_{2,0}$	$A_{2,1}$	$B_{2,0}$	$B_{2,1}$	$C_{2,0}$	$C_{2,1}$	$C_{2,2}$
	Block(s)	2	2	2	2	2	2	2

Table 1: SPMD Mapping Timeline.

which can process a new data token only every three time steps. Since some filters might be more computationally intensive than others, it is a challenge to keep all of the cores active when each filter is mapped to a different core. In this example, 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 is increased, and core$_3$ has fewer data tokens to process, and so the system can run faster. This is the idea of the flexible-filter mapping as illustrated in Fig. 4. Filter C is duplicated on core$_2$ so that core$_2$ can share core$_3$'s load. Besides increasing the code footprint in core$_2$ with respect to the baseline pipeline mapping, this adds also some complexity to the program because now the data stream is split and merged around core$_3$. The split and merge steps are accomplished with two *auxiliary filters*: *flex_split* and *flex_merge*. As explained in Section 4, these filters, which are represented as small black boxes in Fig. 4, can be added to the stream program without changing any of the original filters to guarantee the preservation of the order among the data tokens in the stream.

Data Blocks and Buffering Queues. Besides the filter-to-core mapping, the implementation of a stream program onto a target architecture also requires the mapping of the point-to-point communication channels between the filters. Channels are typically realized as input and output buffering queues within the private local memory of each core. Standard handshake protocols can be used between input and output queues to prevent buffer overflow by means of backpressure signals. Each filter has an inherent input data token size which represents the minimum amount of data that the filter requires to "fire", i.e. to consume a token on each of its inputs channels and produce an output token on each of its output channels. Communicating filters may have input tokens of different data types and each incoming communication channel to a filter may be associated with a different type of token. For example, a filter that adds a constant to a stream of integers only requires a minimum of one integer to operate, while a filter that computes a matrix multiplication requires a pair of entire matrices. In this case the first filter's incoming token type is integer, while the second filter's token type is a matrix.

While sometimes a core may only have enough space in its local memory for a single input data token, it is often the case that its memory holds many input tokens at a given time. The core processes the input tokens using the stream filters that are mapped to it and places the resulting tokens back into the local memory to wait until they can be transmitted downstream. At any point in time the local memory holds some unprocessed incoming tokens and some processed outgoing tokens. Flexible filters step in when the local memory is entirely filled with outgoing tokens and the core cannot make progress until some of them are sent out so that it can make room for more incoming tokens. At this point a flexible filter switches to work on the outgoing tokens.

We abstract the current content of the core's local memory with the notion of a *data block*, which is a substream of data tokens. 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 that can be executed in parallel. A data block is the input unit for *flex_split* and the output unit for *flex_merge*. 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. Coarser granularity can limit the benefits of flexibility in the data stream because it puts more constraints on the possible data flow. For the remainder of this section, we show the details of how data blocks are processed given several different mappings of our example stream program on a three-core system where each core has enough buffer space for two blocks of data and can work on either of them.

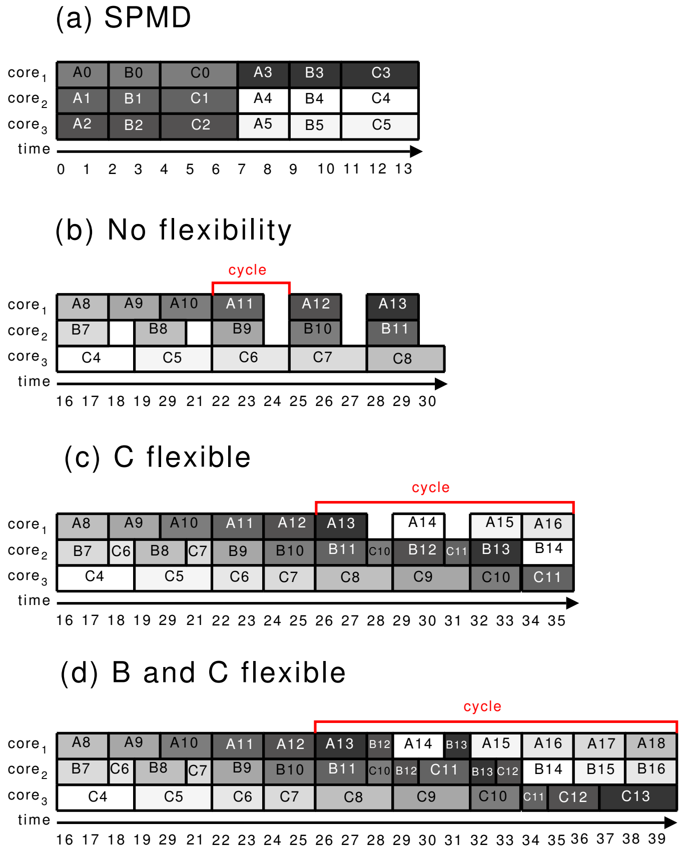
Table 1 shows the timeline for an SPMD mapping where the entire stream program is duplicated separately to each core, with no intercommunication. The table shows both the current step being executed on each core, and the contents of the core's local buffering memory. Filter A, whose latency is assumed equal to two, is computed over two time steps, which for block i are denoted $A_{i,0}$ and $A_{i,1}$, respectively. With an SPMD mapping, cores work on one block until completion and then start the next block. Though not shown in this table, depending on the method of data forwarding, each core may also be holding the next block (3, 4, and 5 for core$_1$, core$_2$, and core$_3$, respectively). As mentioned above, the MST for this mapping is $\frac{3}{7} = 0.429$ since the implementation can process three new input data sets every seven time steps.

Table 2 illustrates the trace when the filters are mapped with a baseline pipeline mapping, i.e. where each filter is assigned to only one distinct core. Even though the filters' latencies are not equal, the buffer capacity allows the faster filters to work ahead. However, at time step t_{18}, core$_2$ must stall. At this step, 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

Time Steps		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
core$_1$	Step	$A_{0,0}$	$A_{0,1}$	$A_{1,0}$	$A_{1,1}$	$A_{2,0}$	$A_{2,1}$	$A_{3,0}$	$A_{3,1}$	$A_{4,0}$	$A_{4,1}$	$A_{5,0}$	$A_{5,1}$	$A_{6,0}$	$A_{6,1}$	$A_{7,0}$	$A_{7,1}$
	Block(s)	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
core$_2$	Step			$B_{0,0}$	$B_{0,1}$	$B_{1,0}$	$B_{1,1}$	$B_{2,0}$	$B_{2,1}$	$B_{3,0}$	$B_{3,1}$	$B_{4,0}$	$B_{4,1}$	$B_{5,0}$	$B_{5,1}$	$B_{6,0}$	$B_{6,1}$
	Block(s)			0	0	1	1	2	2	3	3	4	4	4,5	5	5,6	5,6
core$_3$	Step					$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{3,0}$	$C_{3,1}$	$C_{3,2}$
	Block(s)					0	0	0,1	1	1,2	1,2	2,3	2,3	2,3	3,4	3,4	3,4

Time Steps		t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}
core$_1$	Step	$A_{8,0}$	$A_{8,1}$	$A_{9,0}$	$A_{9,1}$	$A_{10,0}$	$A_{10,1}$	$A_{11,0}$	$A_{11,1}$	-	$A_{12,0}$
	Block(s)	8	8	8,9	9	9,10	9,10	10,11	10,11	10,11	11,12
core$_2$	Step	$B_{7,0}$	$B_{7,1}$	-	$B_{8,0}$	$B_{8,1}$	-	$B_{9,0}$	$B_{9,1}$	-	$B_{10,0}$
	Block(s)	6,7	6,7	6,7	7,8	7,8	7,8	8,9	8,9	8,9	9,10
core$_3$	Step	$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	$C_{5,0}$	$C_{5,1}$	$C_{5,2}$	$C_{6,0}$	$C_{6,1}$	$C_{6,2}$	$C_{7,0}$
	Block(s)	4,5	4,5	4,5	5,6	5,6	5,6	6,7	6,7	6,7	7,8

Table 2: Baseline Pipeline Mapping Timeline.


(a) SPMD

(b) No flexibility

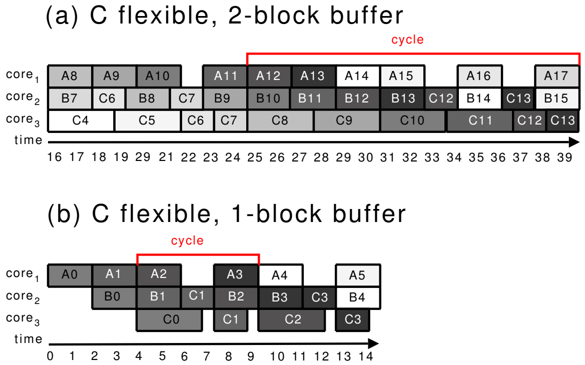
(c) C flexible

(d) B and C flexible

Figure 5: Flexible Filter Timelines.

cycle through a pattern of states, in this case the pattern from t_{22} to t_{24}. By analyzing the behavior during one iteration of the cycle, we can have confirmation that the baseline pipeline implementation has an MST equal to $\frac{1}{3}$. 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.

Fig. 5 summarizes timelines for several mappings in a more abbreviated format that does not include the current memory state. For each case other than SPMD, the timelines start at t_{16} using the same state of t_{16} in Table 2 and continue until a cyclic pattern emerges. Fig. 5(a) and (b) depict the same timelines as in Tables 2 and 1, respectively. Fig. 5(c) 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 Fig. 4). The cyclic pattern for this mapping begins at time step t_{26} and continues until t_{35}. Fig. 5(d) shows an alternative flexible-filter mapping where both filters B and C have been made flexible. In particular, filter C is again mapped to core$_2$ and core$_3$, while filter B is mapped to core$_1$ and core$_2$. Here, the pattern goes from time step t_{26} to t_{39}.


(a) C flexible, 2-block buffer

(b) C flexible, 1-block buffer

Figure 6: Time-line when filter C has a granularity of 2 tokens per block.

In our example, when no filters are flexible, the MST is degraded by 22%. When only filter C is flexible, the MST is increased to $\frac{4}{10} = 0.400$ (only 7% degradation). When both filter B and C are flexible, the MST reaches its ideal limit of 0.429, thus matching the MST of the SPMD mapping. Note that we are not simply duplicating a filter to achieve data parallelism (e.g. as in [10]); instead, data parallelism is used to balance load dynamically as an alternative to stalling one of the processing cores in response to backpressure. In an implementation with optimal MST, such as the examples of Fig. 5(a) and (d), all of the cores are always busy, but by using flexible filters only one copy of filter A, and two copies of filters B and C are necessary instead of the three copies of each filter that are used in the SPMD implementation.

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 and B's data blocks up into halves. 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. Fig. 6 repeats the mapping from Fig. 5(c) to show the timeline when C has this constraint. There are two cases shown. In Fig. 6(a), we assume buffers of size two just like in the previous examples, while in Fig. 6(b) we assume that the buffer has capacity for

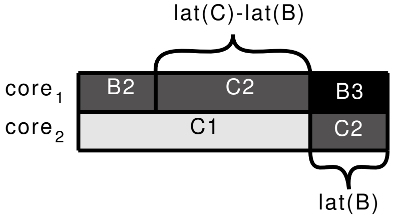


Figure 7: One flexible filter.

one data block only. At t_7 in Fig. 6(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.

Flexible Filter Design Flow. These examples illustrate how to determine the throughput of a system given the stream program, the flexibility of its filters, and their mapping onto an underlying architecture. In the next sections we examine how to decide what flexibility to use and how to modify the original stream program to incorporate this flexibility. Briefly, the Flexible Filter design flow consists of three steps:

1. **Design:** Programmer "designs" a stream program.
2. **Compilation:** ILP-based analysis (Section 3) identifies good places to add flexibility to the program and alters the stream program to include this flexibility by adding Flexible Filter Split and Merge functions (Section 4). We perform this step manually in this work to demonstrate our idea without building a compiler.
3. **Run-time:** By default, the dataflow pattern designed by the programmer is followed. In times of greater load, however, excess traffic is redirected to the flexible filter's secondary copy.

3. CALCULATING THE MST

In this section we describe a method to calculate the MST of a mapping of a stream program on a given multi-core architecture. We assume constant filter latencies. This assumption simplifies the analysis, and allows us to compare MST across different filter configurations. However, notice that dynamic load balancing with flexible filters does not require that the filters have a constant latency. If the latencies vary at runtime, the system automatically settles into the best possible schedule given the new latencies and the static mapping, assuming that the system is running a single stream. If the system resources are shared between multiple simultaneous streams, additional care may need to be taken to coordinate parallel flexibility.

One Flexible Filter. When only one filter is flexible across two cores, the MST of the two cores together can be calculated as follows: given filters B and C with latencies L_B and L_C, where B is fixed and C is flexible, if $L_C > L_B$ then MST $= \frac{2}{L_B+L_C}$. The timeline in Fig. 7 shows how filters B and C overlap when $L_C > L_B$. In the opposite case, when $L_B > L_C$, the same performance improvement is possible if B is flexible and is pushed downstream.

A Pipeline of Flexible Filters. Consider the case where two consecutive filters are flexible next to each other. For instance, consider the example of Fig. 5(d) with filters A, B, C, where B and C are flexible. We would like to know how close the performance of a system with this flexible configuration can come to the ideal MST for an architecture with a given number of cores. Based on which filters are flexible, we can define the following linear programming problem to determine the system's MST: For each core core$_i$, we construct a variable X_i based on the percentage of time that core$_i$ spends on each filter. In the above example, core$_1$'s variable $X_1 = X_{1A} + X_{1B} + X_{1W}$, where X_{1A} and X_{1B} correspond to the time that core$_1$ executes filters A and B, respectively, and X_{1W} corresponds to the time that core$_1$ waits. The execution time of a filter partitioned across multiple cores is defined to be equal to the sum of the execution times of the parts of the filter executed on each core. For example, core$_2$ and core$_3$ may work on a filter C and the sum of times spent on filter C by both cores must equal L_C. The X_i variables correspond to the cyclical firing pattern illustrated in Fig. 5. The complete linear programming problem for our example is the following:

Minimize $X_{1W} + X_{2W} + X_{3W}$ (wait time), subject to:

$$\begin{aligned} X_{1A} &= L_A \\ X_{1B} + X_{2B} &= L_B \\ X_{2C} + X_{3C} &= L_C \end{aligned}$$

which ensure that the total time spent working on a block for a filter by the cores matches the latency of that filter, and:

$$\begin{aligned} X_1 - X_2 = X_{1A} + X_{1B} + X_{1W} - X_{2B} - X_{2C} - X_{2W} &= 0 \\ X_2 - X_3 = X_{2B} + X_{2C} + X_{2W} - X_{3C} - X_{3W} &= 0 \end{aligned}$$

which ensure that the wait times are selected so that all cores work (and wait) over the same period of time. One solution for the example program is $X_{1A} = 2$, $X_{1B} = \frac{1}{3}$, $X_{2B} = \frac{5}{3}$, $X_{2C} = \frac{2}{3}$, $X_{3C} = \frac{7}{3}$, and $X_{1W}, X_{2W}, X_{3W} = 0$.

We can now give the constraints for general applications. For each filter f,

$$\sum_i X_{if} = L_f$$

for all i such that f is mapped to core$_i$. And for each core$_i$ (except the core with the highest index),

$$\sum_{f \in filters\ on\ core_i} X_{if} \quad - \quad \sum_{f \in filters\ on\ core_{i+1}} X_{(i+1)f} = 0$$

Additional constraints can be added for granularity restrictions and the problem can be solved as an integer linear program. Then, using the solution to the problem we can compute the MST as function of the ideal MST as follows:

$$\begin{array}{ccc} \%\ of\ time\ doing\ useful\ stuff & * & ideal\ MST \\ \dfrac{X_1 + X_2 + X_3 - (X_{1W} + X_{2W} + X_{3W})}{X_1 + X_2 + X_3} & * & \dfrac{3}{L_A + L_B + L_C} \end{array}$$

4. SPLIT AND MERGE

As mentioned in Section 2, once the filters that will be made flexible have been selected the original stream program is transformed into a flexible stream program by duplicating

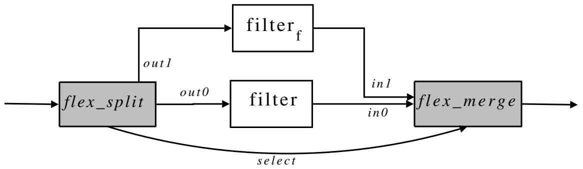


Figure 8: Making a filter flexible.

these filters and by adding pairs of *flex_split* and *flex_merge* auxiliary filters around the duplicated ones. Fig. 8 shows the connection among the filters. *Flex_split* and *flex_merge* can be provided by an application-independent library and added to a program without the need for modifications to the original filters. They are added to the stream program in the same way regardless of whether the filter's upstream or downstream neighbor is made flexible. The direction of flexibility is determined based on where the duplicate filter is mapped.

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 filter f's primary copy 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 $out1$). It 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 filter copy 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 needs to check that there is buffering space for the data in the receiving core. A typical handshake protocol ensures 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.

Streaming programming languages typically abstract away the backpressure mechanism that is implemented at the lower level of the inter-core communication stack [1, 25]. 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 provided by Gedae [1] that returns how much buffering space is available in the next core's buffer. If the programmer does not use *avail()* to check the buffering availability of its output channels then at runtime 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 b 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.

Algorithm 1 flex_split
Input: stream *in*; Output: streams $out0, out1, select$

```
pop data block $b$ from $in$
$n0 \leftarrow avail(out0)$
$n1 \leftarrow$|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 $out$;
```

If a flexible filter is inherently slower than the rest of the filters, then the imbalance will cause the input buffering queue of its primary copy to be full often, and the data flow will be redirected to the secondary copy at regular intervals. Instead, if the flexible filter experiences only occasional spikes of activity that cause it to slow down—or if its upstream neighbor occasionally creates extra data tokens on its output— then the data flow will mostly follow the baseline pipeline behavior and *flex_split* will intervene sporadically to adapt the data flow.

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

Filter Mapping. The last step in our approach is mapping the entire set of filters to cores including duplicate flexi-

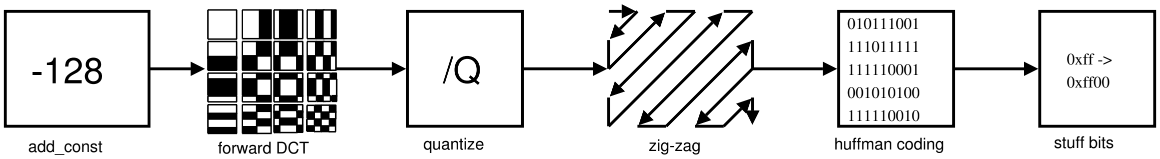


Figure 9: JPEG block diagram.

filter	description
add_const	shifts values by 128
forward DCT	discrete cosine transform
quantize	normalizes results, outcome is that many values become zero and are easier to compress
zigzag	converts 2D block to 1D vector by listing block elements in zigzag order
coding	applies Huffman coding to data stream
stuff_bits	packs bits into integers and adds some padding bits

Table 3: Description of JPEG filters.

ble filters and auxiliary filters. Flexible filter mapping starts with a baseline pipeline mapping for the filters of the stream. Then, a duplicate copy of each flexible filter is mapped to the forward or backward neighbor of the core that the original filter is mapped to. If several consecutive filters are co-located on a single core, it may be useful to treat them as a single filter for flexibility purposes to cut back on the overhead of data queueing between filters. As filters are mapped to cores, one design decision is where to map the split and merge filters. All of the data stream must pass through both filters. In the case of a single flexible filter, which is likely a performance bottleneck, it is natural to map the split and merge filters to the same cores as the flexible filter's upstream and downstream neighbors, respectively. When there are several consecutive flexible filters, however, the best choice is less clear. One possibility is to map the split/merge pair between two flexible filters onto a separate core. We explore a few possibilities in our experiments in the next section.

5. EXPERIMENTS

In this section we describe a case study on the application of flexible filters to the implementation of a JPEG encoder program. We performed all of our experiments on a QS21 CellBlade which hosts two IBM Cell processors [21]. The Cell architecture is a heterogeneous multicore system-on-chip originally designed for high-performance embedded applications [13, 22]. 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 205GB/s of data transfers [2]. 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 tradeoff between program code and data buffering when we make filters flexible.

To program the IBM Cell we took advantage of Gedae, a data flow 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 [1]. Gedae's API contains functions to implement the communication channels, including an *avail()* function that gives information on how much space is available in the input and output buffers.

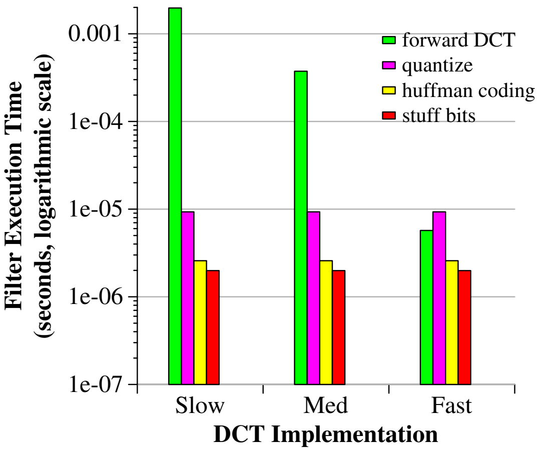


Figure 10: Profile of JPEG filters for three DCT implementations.

Fig. 9 shows a block-level diagram of a baseline gray-scale JPEG encoder. Table 3 describes the functionality of its filters, most of which operate on 8x8 pixel blocks. The following filters are stateless between blocks: *add_const*, *forward DCT*, *quantize*, and *zigzag*. Filter *coding* does retain a small amount of state (one integer: the previous pixel block's DC coefficient). Since there is state, it is slightly more difficult to make this filter flexible, but in this particular case, the DC coefficient of the previous pixel block can also be captured in *zigzag* or *quantize* and then tagged on to the current pixel block (effectively moving the state information out of the *coding* filter). Filter *stuff_bits* packs the bits produced by Huffman coding into bytes so they can be written to output. In addition, *stuff_bits* checks for `0xff` words in the data stream and adds padding bits when it finds them (`0xff` is reserved as a tag marker in the bitstream). It would be difficult to make this filter flexible because codes produced by *coding* have variable length and may span across byte boundaries in the output of *stuff_bits*.

Fig. 10 shows some profiling data that we collected empirically with a baseline pipeline implementation of the JPEG encoder. Since our first implementation of the *DCT* filter was rather slow in comparison to the other filters, we made two additional implementations. We used all three versions to evaluate how the benefits of flexibility vary depending on the relative latency of the flexible filter. Filters *add_const* and *zig-zag* are omitted because their measured latencies are negligible. Indeed, they are lower than the latency of the function used to measure the latency on the Cell processor (200-300 ns, called twice).

[2] Notice that in our experiments we mapped the filters only to the SPU processors.

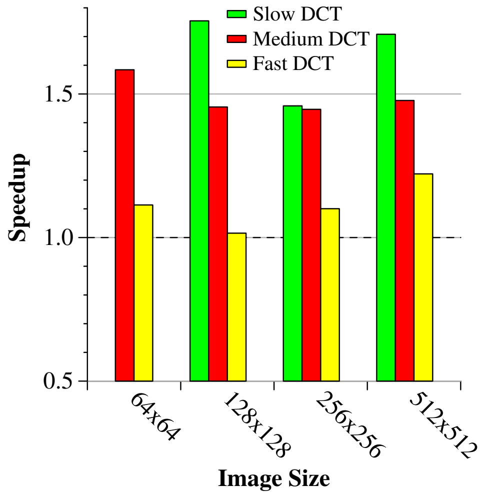


Figure 11: Speedup gained from giving the DCT filter flexibility degree 1. Image sizes are 64x64, 128x128, 256x256, and 512x512 pixels.

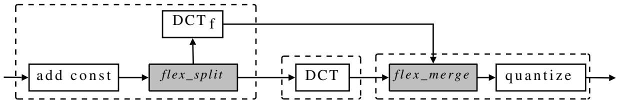


Figure 12: Mapping Flexibility to Cores.

Below is a brief description of the DCT implementations:

- *Slow.* Implements the 2-D DCT based on its standard definition, $f(x,y) = \sum_{u=0}^{n-1}\sum_{v=0}^{n-1} T(u,v)h(x,y,u,v)$ [9].
- *Medium.* Computes the 2-D DCT by computing 1-D DCT on each column of the 8x8 pixel block and then on each row of the block. The 1-D DCTs are implemented with *for* loops.
- *Fast.* Similarly to the medium algorithm the 2-D DCT is computed as 1-D DCTs over the columns and rows of the pixel block. The 1-D DCT is implemented in a pipeline fashion, based on the algorithm described by Kovac and Ranganathan [16].

Notice that our implementation is not highly optimized for the Cell architecture, e.g. we did not make use of the intrinsic vector operations that could certainly be used to write a more efficient encoder. With regard to flexible filters, the implementation of an application can change the relative costs of different filters with respect to each other and to the overhead of data transfer, but it would not change the algorithms or strategies of flexible-filter load balancing. We stop at a basic implementation because our goal is not to make the most efficient JPEG encoder for the Cell, but to test the proposed idea.

Implementation with One Flexible Filter. Fig. 11 shows the speedup from using a flexible filter for *forward DCT* with the mapping shown in Fig. 12. All other filters are not flexible. We observe that adding flexibility to the stream program can significantly improve performance and also that the benefits of flexibility are greater when there is a greater imbalance between the latency of *forward DCT* and the latencies of other filters. Performance improved nearly 85% for the slow DCT implementation and up to 20% for the fast DCT implementation. One unexpected result is that the performance shows improvement for the fast implementation where *forward DCT* is not the system bottleneck. Table 4 shows the actual time to process the images. In some cases, the medium DCT implementation with flexibility outperforms the fast DCT implementation with no flexibility.

	Encoding Time per Image (seconds)			
Implementation	64x64	128x128	256x256	512x512
Slow DCT	0.099	0.388	1.542	6.253
Slow DCT, flex	0.059	0.231	0.871	3.395
Med DCT	0.022	0.086	0.347	1.334
Med DCT, flex	0.014	0.059	0.240	0.903
Fast DCT	0.013	0.057	0.236	0.954
Fast DCT, flex	0.012	0.056	0.215	0.781

Table 4: Performance of JPEG Encoder.

Implementation with Two Flexible Filters. In the fast DCT implementation, filter *quantize* is the system bottleneck. Fig. 13 shows three possible mappings for the filters when we make both *forward DCT* and *quantize* flexible. The first mapping follows the convention of mapping *flex_split* to the upstream neighbor of the flexible filter and *flex_merge* to the downstream neighbor. Since two filters in a row are flexible, the first *flex_merge* must come before the second *flex_split*. The second mapping attempts to reduce the amount of data being transferred in exchange for reduced flexibility. The third mapping offloads *flex_merge* and *flex_split* between *forward DCT* and *quantize* to a separate core. As reported in Fig. 14 the performance of all cases is roughly the same. Speedup is as high as 25% for the 512x512 image, though there is no speedup for the 128x128 image. To illustrate how the profile data translates to dynamic load we collected the following measures: for the 512x512 image, when only *forward DCT* is flexible, *flex_split* redirects around 35% of the data tokens to the secondary filter. When both *forward DCT* and *quantize* are flexible using the first mapping from Fig. 13, 47% of the data is redirected to the secondary *forward DCT* and 34% is redirected to the secondary *quantize.*

Overhead of Flexibility. In the absence of overhead from data transfer and inefficiencies from data granularity, we would expect that making both *forward DCT* and *quantize* flexible should improve performance. As shown in Fig. 14, however, the extra communication overhead outweighs the benefits of flexibility in this instance. Besides the overhead of moving the data, flexible filters incur some additional overhead. First, *flex_split* and *flex_merge* require a couple extra steps of data copying; then, streams that can push and pop arbitrary numbers of data tokens according to user-level input are more complex than streams that always push and pop the same number of tokens; finally, flexible filters require some code duplication compared to a baseline pipeline (though less code overhead than an SPMD mapping).

6. RELATED WORK

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 [3, 8, 14]. 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 ran-

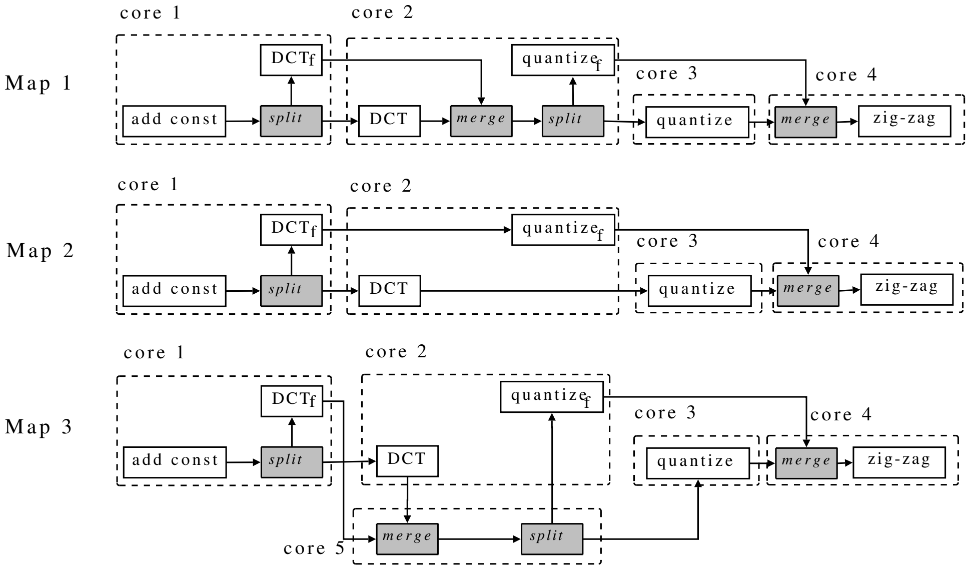


Figure 13: Mapping Pipeline Flexibility.

domly. In contrast, flexible filters do not steal randomly, but use the knowledge that neighbors of a bottleneck filter will be idle because they dependent 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 data flow is re-routed when a bottleneck arises. Flexible filters are identified when filters are mapped to the system and determine the available routes for data during runtime. In addition, we focus on the case of a distributed memory system where the code for tasks is also distributed. Flexible filters are specialized to stream programs because dependencies in the stream allow us to narrow down good candidates for redundant-code placement.

Load balancing approaches specific to stream programs can be categorized depending whether the stream models rely on data parallelism or pipeline parallelism (in practice both approaches can be used simultaneously [10]) 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 [2, 23]. 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 for balancing the load of pipeline parallel stream programs involve a central control and/or phases where the compute nodes collect statistics which are used by the control to direct reorganization [7, 24, 26]. 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. Flexible filters simulate filter migration by duplicating some filters on the cores and invoking duplicates when the load becomes unbalanced.

Chen *et al.* perform load balancing for stream programs by compiling several alternative filter mappings [6]. During run-time, 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 developed by Huston *et al.*, data tokens are forwarded based on threshold values in the input and output queues [12]. Load balancing with flexible filters similarly is an outcome of the state of the queues, but the difference is that flexible filters balance 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.

Many stream programming languages, such as StreamIt include *split* and *join* nodes in their supporting library that are used to transform the stream programs [7, 10, 25]. *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 data flow is split at run-time regardless of the loads on the various cores. In contrast, the Flexible-Filter *flex_split* and *flex_merge* filters described in Section 4 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.

Synchronous Data Flow (SDF) is a well-studied model of computation which like the stream programming model defines a computation as a network of processing nodes through which data flows [18, 19]. We support a less restrictive set of applications than SDF because flexible filters tolerate variable token production and consumption rates and adapt to

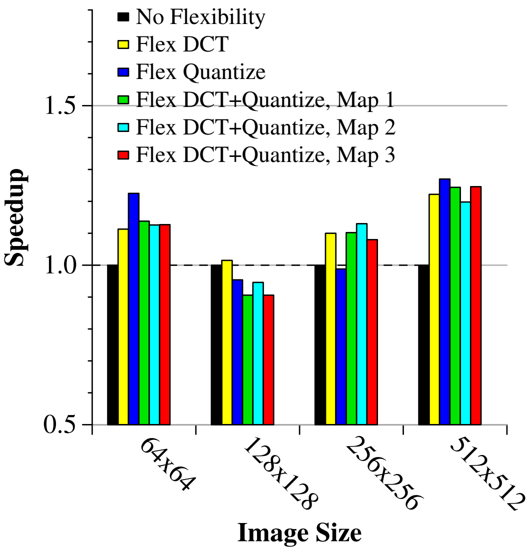


Figure 14: Speedup gained when both DCT and Quantize are flexible using the Fast DCT Implementation and mappings shown in Fig. 13.

improve performance in spite of them. Our MST analysis and ILP formulation do currently impose a temporary assumption of constant data rates, and thus may overlap with algorithms used to perform parallel SDF scheduling. However, once flexibility is added to the system, it does not require or depend on constant data rates. If a system needs to switch between several different patterns of data flow, each pattern can be analyzed separately and multiple "schedules" tuned to each separate pattern can be concurrently included in the final flexible-filter mapping.

7. CONCLUDING REMARKS

Flexible filters can significantly improve the performance of stream programs. They are especially effective in cases where one filter has a relatively high execution latency compared to other filters in the program. Since our approach automatically adapts the data flow to the filter latencies, it can reduce the need to break large filters up by hand. Further, load balancing is determined solely by backpressure signals and can be applied both to systems with static filter latencies and systems with dynamically-varying latencies.

We demonstrated the capabilities of flexible filters by presenting the JPEG-encoder case study. While this program is characterized by a linear pipeline structure, flexible filters can be applied to any pipelined segment of stream programs that may have more complex graph structures in an attempt to improve their throughput. Adding flexibility to stateful filters and across a filter that splits a stream or joins multiple streams is a topic for future investigation. In future work we plan also to automate the steps for evaluating where to add flexibility to optimize the system's throughput while minimizing the code overhead and communication costs.

Acknowledgements

This research is partially supported by the National Science Foundation under Award #: 0644202 and Award #: 0811012. 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.

8. REFERENCES

[1] Gedae, http://www.gedae.com/.

[2] R. H. Arpaci-Dusseau et al. Cluster I/O with River: Making the fast case common. In *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 10–22, May 1999.

[3] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems*, 35(3):289–304, 2002.

[4] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. In *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 777–786, Aug. 2004.

[5] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, pages 668–668, Jan. 2003.

[6] J. Chen et al. A reconfigurable architecture for load-balanced rendering. In *Proc. of the SIGGRAPH/ EUROGRAPHICS Conf. on Graphics Hardware*, pages 71–80, July 2005.

[7] E. T. Fellheimer. Dynamic load-balancing of StreamIt cluster computations. Master's thesis, Massachusetts Institute of Technology, 2006. Department of Electrical Engineering and Computer Science.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the SIGPLAN Conference on Program Language Design and Implementation*, pages 212–223, June 1998.

[9] R. C. Gonzalez and R. E. Woods. *Digital Image Processing.* Addison-Wesley Longman Publ. Co., Inc., Boston, MA, 2001.

[10] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of the Intl. Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, Oct. 2006.

[11] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 343–354, Nov. 2005.

[12] L. Huston et al. Dynamic load balancing for distributed search. In *Proc. of the Intl. Symp. on High Performance Distributed Computing (HPDC)*, pages 157–166, July 2005.

[13] J. Kahle et al. Introduction to the CELL multiprocessor. *IBM J. Res. Develop.*, 49(4-5):589–604, Sept. 2005.

[14] P. Kakulavarapu, O. Maquelin, J. N. Amaral, and G. R. Gao. Dynamic load balancers for a multithreaded multiprocessor system. *Parallel Processing Letters*, 11(1):169–184, 2001.

[15] U. J. Kapasi et al. Programmable stream processors. *IEEE Computer*, 36(8):54–62, Aug. 2003.

[16] M. Kovac and N. Ranganathan. Jaguar: A fully pipelined VLSI architecture for JPEG image compression standard. *Proc. of the IEEE*, 83(2):247–258, Feb. 1995.

[17] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, 2008.

[18] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan. 1987.

[19] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, Sept. 1987.

[20] M. D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conference*, Oct. 2006.

[21] A. K. Nanda et al. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. of Research and Development*, 51(5):573–582, Sept. 2007.

[22] D. Pham et al. The design and implementation of a first-generation CELL processor. In *ISSCC Digest of Technical Papers*, pages 184–185, Feb. 2005.

[23] C. Ranger et al. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of the Symp. on High Performance Computer Architecture*, pages 13–24, Feb. 2007.

[24] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Intl. Conf. on Data Engineering (ICDE)*, pages 25–36, Mar. 2003.

[25] W. Thies et al. StreamIt: A compiler for streaming applications, Dec. 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[26] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *Intl. Conf. on Data Engineering (ICDE)*, pages 791–802, Apr. 2005.