# Block-Parallel Programming for Real-time Embedded Applications

David Black-Schaffer
*Department of Information Technology*
*Uppsala University*
*Uppsala, Sweden*
*david.black-schaffer@it.uu.se*

William J. Dally
*Department of Computer Science*
*Stanford University*
*Palo Alto, USA*
*dally@stanford.edu*

***Abstract*—Embedded media applications have traditionally used custom ASICs to meet their real-time performance requirements. However, the combination of increasing chip design cost and availability of commodity many-core processors is making programmable devices increasingly attractive alternatives. Yet for these processors to be successful in this role, programming systems are needed that can automate the task of mapping the applications to the tens-to-hundreds of cores on current and future many-core processors, while simultaneously guaranteeing the real-time throughput constraints.**

**This paper presents a block-parallel program description for embedded real-time media applications and automatic transformations including buffering and parallelization to ensure the program meets the throughput requirements. These transformations are enabled by starting with a high-level, yet intuitive, application description. The description builds on traditional stream programming structures by adding simple control and serialization constructs to enable a greater variety of applications. The result is an application description that provides a balance of flexibility and power to the programmer, while exposing the application structure to the compiler at a high enough level to enable useful transformations without heroic analysis.**

***Keywords*-parallel programming; synchronous data flow; image processing; parallelization; real-time constraints;**

## I. Introduction

Typical embedded media applications (such as video codecs, radio processing, and medical imaging) require vast amounts of computational power and have hard real-time requirements. Traditionally, these have been implemented as fixed-function ASICs. However, as semiconductor technology has scaled to smaller feature sizes, the cost of designing and verifying these ASICs has increased to the point where they are only affordable for the highest volume applications [1]. The fixed-function nature of these designs also makes them extremely expensive in an environment where standards are constantly evolving, as any change in the algorithm requires a re-spin of the chip to accommodate it.

The same trends that are making ASICs prohibitively expensive are making off-the-shelf many-core processors more affordable. These architectures amortize the increased design cost by replicating tens or hundreds of cores across a single chip [2], [3]. As feature sizes scale down, many-core chips become both cheaper to design (greater amortization of design effort across more cores) and more powerful (increased total core count). This trend has led to a strong desire to use commodity many-core architectures in place of custom ASIC designs for computationally demanding embedded applications.

Unfortunately, while the hardware is rapidly becoming capable of supporting the computation rates required by these applications, programming systems have not kept pace [3]. The current ad-hoc SMP-derived programming models advocated for many-core processors are based on manual parallelization via threads and manual inter-process communication via either coherent shared memory or explicit message passing. These approaches fail to incorporate any notion of the physical location of data or processing resources and the real-time requirements of the application. Programming in this manner requires that the programmer have intimate knowledge of the particularities of the architecture to ensure the application meets its real-time constraints and uses its communication resources efficiently. Additionally, these brittle manual approaches suffer similar re-implementation costs as ASICs when algorithms change, thereby losing much of the programmable benefit. As the number of cores and the required throughput of the applications increase, the current approach of manually partitioning, placing, and scheduling threads and data to ensure the required throughput will rapidly become untenable.

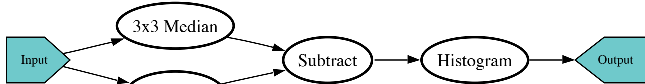
Stream languages [4], [5], [6], [7] have been proposed to meet these challenges. Their application representation exposes and parameterizes the data movement and computation at a high-level, which enables straight-forward manipulation without the extreme complexity of low-level analysis of imperative code. The stream representation frees the programmer from having to determine and implement the correct parallelization, as the data-, task-, and pipeline-parallelism are explicit in the application description. By providing a high-level parameterized view of the application, it also enables automatic placement of the application's tasks on a many-core processor.

However, despite these advantages in program manipulation and representation, streaming languages have not been widely adopted. One reason for this is that they often provide a too Spartan interface for the programmer (e.g., allowing only a single input/output per kernel [7] or only one-dimensional data [7], [6]), or a too general one (e.g., allowing fully $n$-dimensional data with complex access and reuse patterns [4], [8], [9]). The former forces programmers to write code in a complex manner which inhibits compiler analysis and programmer understanding [10], while the latter makes the compiler analysis fundamentally difficult [11], [4].

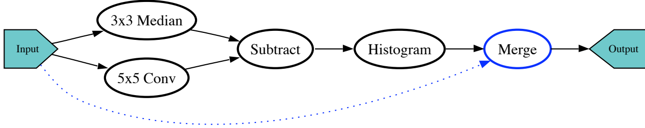
This paper presents:

- A block-parallel streaming language [12] with 2-dimensional data streams, multiple kernel inputs and outputs, and explicit throughput constraints,
- Compiler data-flow analyses for determining computation requirements to meet the throughput constraints throughout the program based on static input and computation rates,
- And automated methods for buffering and parallelizing the application to meet the throughput constraints.

The approach presented here attempts to find a good balance




(a) Block-parallel representation of an image processing application




(b) 1(a) with histogram manually split into parallel and serial portions

Figure 1. Example block-parallel application description for a simple non-linear image processing application. In 1(b) the serial portion of the histogram is explicitly annotated by a data dependency edge (see Section IV-B) from the input, indicating a limit of one instance of the merge kernel per input frame.

between providing a rich programming interface and one that is readily analyzable by the compiler. To motivate the design decisions, consider the stream-based application shown in Figure 1(a). This example is typical of a simple image processing task, which takes a stream of 2D input images at a fixed rate, passes it through two filters, takes the per-pixel difference of the result, and then calculates a histogram over the resulting image. This is a simple approximation of a real-time non-linear image analysis task. In order to automatically compile and map this application with hard real-time constraints to a many-core processor, several analyses need to be undertaken.

The first analysis is with regards to serialization and control. The histogram processing is partially data-parallel, and partially serial. That is, individual partial histograms can be computed in parallel, but the final combination is a serialization step. In addition, the final histogram output is generated only once per input image. This requires that the kernel receive appropriate control information to know when it has finished an image so that it can output the results. To implement this, the histogram kernel is manually divided into parallel and serial portions, and this dependency is annotated in the application graph, as seen in Figure 1(b). Adding this annotation to the application graph enables automatic analysis and parallelization while respecting the limited available parallelism, as described in section IV-B.

The second set of analyses are with regards to data movement. The input to this application is defined as a two-dimensional image which is processed by two windowed filters (a $3 \times 3$ median and a $5 \times 5$ convolution). The parameterized inputs and outputs are shown on the application graph in Figure 2. In this example, the input data arrives one pixel at a time, which implies that the data needs to be appropriately buffered to provide enough rows of data for the two filters. In addition, the two filters have different output "halos", such that the result of the median filter will be one pixel larger on each side than the convolution filter. In order for it to make sense to take the difference of the two outputs, the input to the convolution filter needs to be padded or the output from the median filter needs to be trimmed. (See Section III-C.) The result of applying these automatic transformations for buffering and trimming is seen in Figure 3.

The third analysis is for parallelization to meet the real-time constraints imposed by the input. In this case the input is arriving at a fixed rate. To ensure that the application processes data at this rate, the computation kernels must be appropriately parallelized to meet the throughput rates on the target hardware. For data-parallel kernels, such as the image filters shown here, this requires replicating the kernels across multiple processors and correctly distributing and collecting the data from them. After parallelization and insertion of split/join kernels (Section IV), the final application is as seen in Figure 4.

To make this language successful, the aforementioned analyses should be as automated as possible. That is, the programmer should not have to address issues of buffering, data sizing, parallelization for different input rates, and analysis of control and serialization. Instead, the application description should expose the structure of the application to the compiler in such a way as to facilitate automated analysis and manipulation.

Figure 4 shows the result of these automated analyses and manipulations when applied to the program in Figure 1(b). The data has been appropriately buffered and the kernels parallelized to meet the real-time constraints of the input. The final application was then simulated on a timing-accurate functional simulator to verify the throughput met the real-time requirements. This paper describes how the block parallel programming language parameterizes the application description to enable these automatic analyses and manipulations, while providing a flexible and intuitive programming framework.

## II. Application Description

The application definition for a stream language consists of an *application graph* that connects *computation kernels* via data stream channels[1]. The channels act as FIFOs, moving data in streams between the kernels. When a kernel receives sufficient data it executes its computation code on the data and produces output which is fed to down-stream kernels. The block parallel programming approach presented here extends this model by adding data dependency edges to express limited parallelism, multiple execution methods per kernel to allow for more flexible data sharing and control handling, and control tokens to enable flexible and analyzable control. As with most streaming languages, the input sizes, rates, and kernel resource requirements are assumed to be statically known at compilation.

### A. Input/Output Parameterization

The application description presented here tries to make an explicit tradeoff between overly simplistic [7], [6] and overly general [4], [11] data stream representations. The result is a data parameterization that supports two-dimensional windowed accesses in a fixed (left-to-right, top-to-bottom) scan line order. This approach provides programmers with significant flexibility, without overcomplicating the compiler analysis. In particular, this parameterization directly addresses the significant fraction of applications that process two-dimensional images, without inhibiting one-dimensional signal handling.

The parameterization defines the two-dimensional size for each input and output, as well as its step size, which determines how

[1]In SDF parlance kernels are referred to as "actors" which receive "tokens".

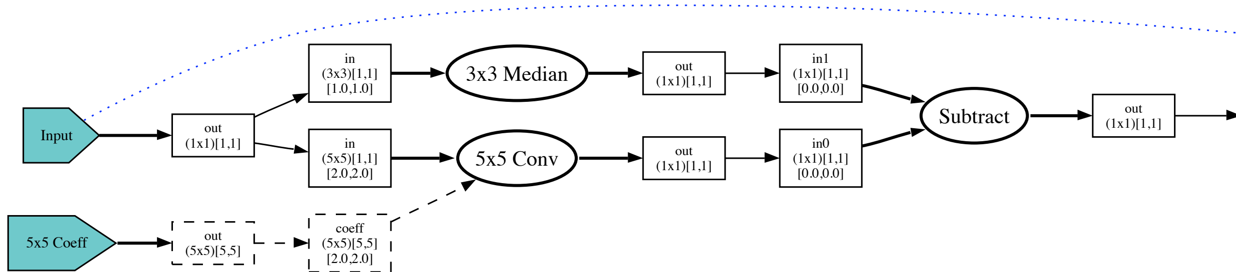



Figure 2. Partial application graph for the application from Figure 1(a) with the parameterized inputs and outputs annotated for each kernel. Note the addition of the explicit coefficient and bin inputs to the convolution and histogram kernels, respectively, with dashed edges indicate they are replicated inputs. Input/Output parameterization is discussed in Section II-A.

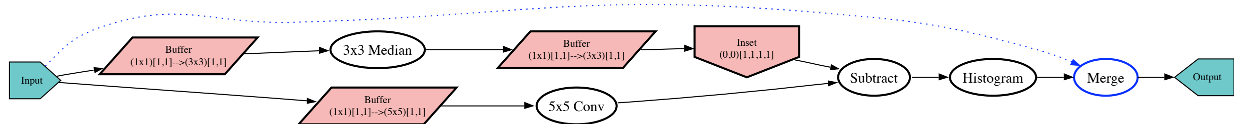



Figure 3. Application graph with automatically inserted buffers (parallelograms, see Section III-B) and inset kernels (inverted houses, see Section III-C).

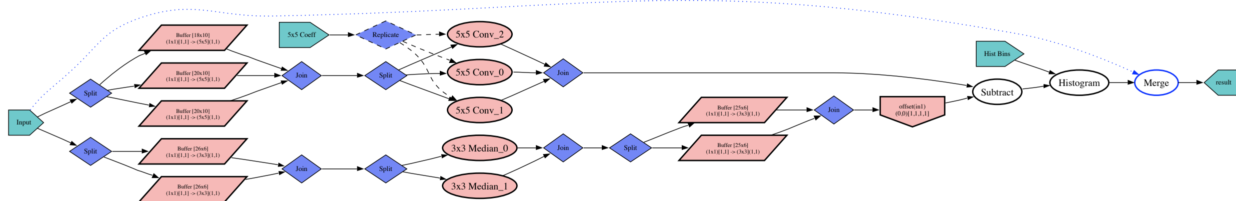



Figure 4. Application automatically parallelized for the given input size and rate and hardware capabilities. Parameterized split/join kernels (blue diamonds, see Section IV) have been added to distribute and collect the data. Replicated inputs (dashed lines) have their inputs replicated rather than distributed.

far the input/output window moves on each iteration in $X$ and $Y$ dimensions. Together, these specify the data usage and reuse for regular windowed processing kernels, such as the convolution and median filters discussed earlier. To complete the parameterization, an offset from the input data to the output is also specified for each input. This is required to enable correct padding or trimming, as discussed in Section III-C. The inputs to the application as a whole specify an input rate which determines the real-time constraint for the overall application processing.

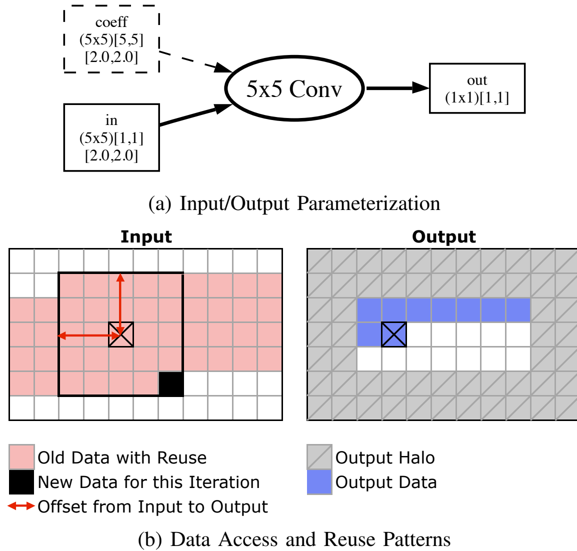
An example parameterization is shown for a $5 \times 5$ convolution kernel in Figure 5(a). Here the data input "in" is defined to have a size of $(5 \times 5)$ and a step size of $(1, 1)$. For this kernel, the offset is [2.0,2.0][2], which indicates that each output is generated 2 pixels over and down from the upper-left input value, as seen in Figure 5(b). The output "out" is defined to have a size of $(1 \times 1)$ and a step of (1,1) as each invocation of the kernel produces one output result. The remaining input, "coeff", is used to load coefficients into the kernel, and is independent of the data input "in". The coefficient data is used to initialize the convolution, but can also be reloaded whenever a change in filter is required. The coefficient input's size is $(5 \times 5)$, as $5 \times 5$ coefficients are needed for a $5 \times 5$ convolution, but because new coefficients should replace the old, its step size is (5,5), indicating no data is reused. The "coeff" input is further defined to be *replicated*, indicating that if the kernel is parallelized, the data for the "coeff" input should not be split across the parallel kernel instances but rather replicated. This ensures that each parallel instance of the kernel receives the same coefficients. Replicated inputs are indicated in the application graphs by dashed edges.

By using this parameterization, and defining the data ordering to be scan-line based (e.g., left-to-right, top-to-bottom), it becomes straight forward to determine the data movement, reuse, and iteration size for each kernel. For example, for the convolution kernel, the parameterization implies that the window of inputs used by the kernel moves over by one column in the $X$-direction for each iteration (step size 1 in $X$), and therefore reuses the first 4 columns of data $(width - step)$. In the steady-state, that is, where the previous rows can be reused as well, this translates into a maximum data-reuse of 24 of 25 elements, as shown in Figure 5(b).

### B. Kernel Definitions

Computation kernels are defined by their input and output parameterizations, computation method(s), resource requirements (cycles and memory), and mappings between inputs and methods, and methods and outputs. These mappings and resource requirements

[2]Fractional offsets may be required for downsampling kernels.




(a) Input/Output Parameterization

(b) Data Access and Reuse Patterns

Figure 5. Convolution kernel input/output parameterization and data access patterns. The parameterization of the inputs and outputs, combined with the fixed scan-line data input order, determines the data access and reuse. The inputs and outputs contain implicit buffer space for one iteration, providing a small amount of channel buffering.

are defined in the `configureKernel` method (see Figure 6) which is called when the kernel is instantiated in the application graph. The intent of this design is to provide more flexibility than previous languages by permitting multiple inputs, outputs, and methods, without inhibiting analysis. As will be seen in the following examples, this added flexibility directly translates into more intuitive and understandable code.

As with all stream processing languages, the core of a kernel definition is the code, or *method*, that it executes on the incoming data. The block-parallel programming model presented here extends this to allow multiple computation methods per kernel, each of which can be triggered on a different set of disjoint inputs. For example, the convolution kernel consists of two methods: `runConvolve`, which executes when data arrives on the "in" input, and `loadCoeff` which executes when new coefficients arrive on the "coeff" input. Figure 6 shows how the `runConvolve` method is registered as requiring the "in" input and generating the "out" output, while the `loadCoeff` method only requires the "coeff" input as it generates no output. However, since the methods within a given kernel share data private to the kernel, the `loadCoeff` method can set the kernel's coefficients, which will then be used on subsequent invocations of the `runConvolve` method. By allowing different methods to execute on different sets of inputs, but still share data, this language can more readily encompass kernels that are not purely data-parallel and require control of state or initialization.

Kernels must also specify the resources they require for each execution of a method. This is necessary to allow the compilation system to determine the degree of parallelism required given the input rate and the processing resources. For this implementation, the required resources (computation cycles, memory, and input/output buffers) are specified in the `configureKernel` method by the calls to register the methods and inputs/outputs, as shown in Figure

```
public void configureKernel() {
    /*
     * Define the Inputs and Outputs, register the method, and assign
     * resources consumed.
     */
    createInput("in", width, height, 1, 1,
            Math.floor((double)width/2), Math.floor((double)height/2));
    createOutput("out", 1,1);
    registerMethod("runConvolve", 0, 3, 10, 10+3*height*width);
    registerMethodInput("runConvolve", "in");
    registerMethodOutput("runConvolve", "out");

    /*
     * Define the Input for coefficient loading, register the
     * method called when the coefficients are present,
     * and mark that input as begin replicated. (I.e., inputs
     * to it should be copied, not parallelized.)
     */
    createInput("coeff", width, height, width, height,
            Math.floor((double)width/2), Math.floor((double)height/2));
    registerMethod("loadCoeff", 0, 3, 10, 10+2*height*width);
    registerMethodInput("loadCoeff", "coeff");

    /*
     * When parallelizing, the coefficient input should be replicated,
     * not distributed.
     */
    getInputByName("coeff").setReplicateInput(true);
}

private double[][] coeff;
private double[][] result = new double[1][1];

public void runConvolve(){
    double[][] in = readInputData("in");
    for (int x=0; x<width; x++)
        for (int y=0; y<height; y++)
            result[0][0] += in[x][y]*coeff[width-x-1][width-y-1];
    writeOutputData("out", result);
}

public void loadCoeff() {
    coeff = readInputData("coeff");
}
```

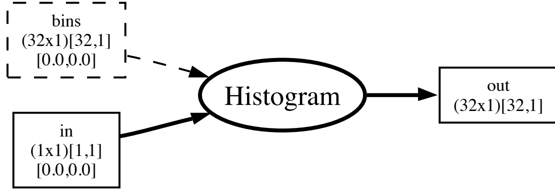
Figure 6. Convolution kernel code. The `configureKernel` method defines the inputs and outputs and mappings between them and the methods when the kernel is instantiated. The two methods for this kernel, `runConvolve` and `loadCoeff` access the shared private array `coeff`.

6. Here the resource requirements are explicitly specified, but they could be estimated automatically [13] or determined from profiling.

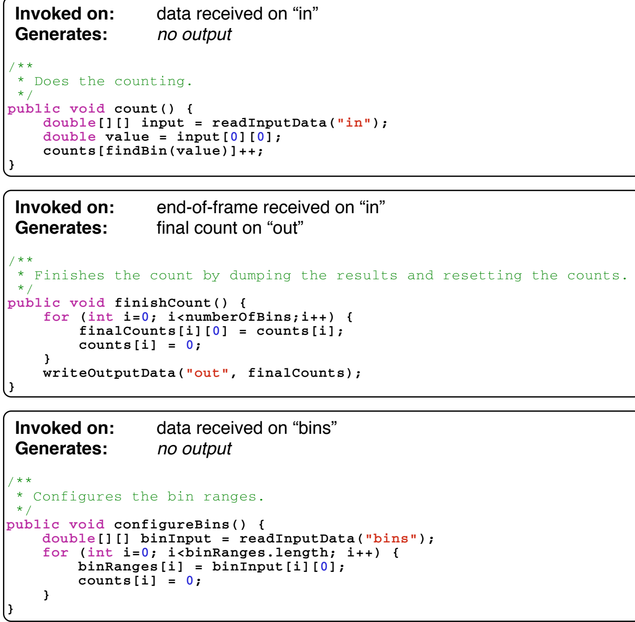
### C. Control Tokens

In addition to sending data between kernels via inputs and outputs, the block-parallel programming approach supports sending *control tokens*. These tokens allow kernels to send and receive irregular (but still statically bound) control messages, either along the same streams as the data or via separate kernel outputs. Kernels are free to define their own control tokens as long as they specify the maximum rate at which they can be generated, which is necessary to allow the compilation system to allocate sufficient resources to guarantee real-time execution. This differs from other implementations [14] where control is treated as a purely asynchronous event that is not analyzed by the compiler. The benefit of enabling such compiler analysis of control signals is that the compiler can account for the resources consumed handling them. This allows programmers to write methods that handle the control signals that do more than simply set local flags, as the time and resources spent in them are appropriately accounted for.

Two types of control tokens are generated automatically by the data inputs to an application: end-of-line and end-of-frame. These tokens are sent out in order with the data from the input at appropriate times. To process these tokens, a kernel defines a method which is triggered when an input receives the appropriate token. For example, Figure 7(b) shows the code for the histogram kernel. It invokes one method (`count`) when it receives data on the input "in" and another (`finishCount`) when it receives an end-of-frame token on the same input. The `finishCount` method generates the final output and resets the bin counts while the




(a) Input/Output Parameterization




(b) Kernel Methods

Figure 7. Histogram kernel definition. Two methods are mapped as being invoked by the same input ("in"), but `finishCount` executes when an end-of-frame control token is received and `count` when data is received. The `configureBins` method is mapped as being invoked when data is received on the "bins" input. Only the `finishCount` method generates data on the kernel's "out" output. By making this mapping explicit, the compiler can appropriately allocate resources for handling each method within the real-time constraints of the application.

`count` method only increments the appropriate bin count. This approach conveniently separates the control and data processing code within the kernel, but allows them to communicate through private variables.

While control tokens enable significant flexibility by adding a simple, analyzable control model to data-parallel stream processing, most kernels will not have any need to deal with most control tokens. For these kernels, such as the convolution kernel discussed earlier, the unhanded control tokens are automatically passed on to the appropriate outputs for the given input in order with the received data. This effectively means that kernels need only pay attention to the control tokens they care about, and can safely ignore any others. In the case where two inputs trigger the same method, such as the subtract kernel in Figure 1(a), the same control token must arrive on both inputs for it to be passed to the output.

## III. Application Analysis

The static nature of the application description, that is, the static input sizes and rates combined with the known kernel resource requirements and application graph, enables a straightforward data flow analysis to determine the required computation at each kernel within an application. This data analysis propagates the application's inputs' size and rate information through the application graph and returns the *iteration size* and *rate* at each kernel in the application, as well as the *inset* from its input data to each original application input. The iteration size and rate allow the compiler to calculate the degree of parallelism and amount buffering required to meet the input sizes and rates, while the inset enables analysis for automatic padding or trimming adjustment to ensure appropriate data alignment across inputs.

### A. Iteration Sizes and Rates

The iteration size and rate for each kernel define the number of times each kernel needs to be executed for each input frame given the inputs' sizes and rates. To determine this, the application's inputs' sizes and rates are propagated through the application graph via a data flow analysis that calculates the iteration size and rate at each kernel, and then propagates this on to subsequent kernels using the parameterization of the kernel's outputs.

For example, if the input to a $5 \times 5$ convolution is a $100 \times 100$ image at 50Hz, the kernel will have an iteration size of $96 \times 96$ at 50Hz (the kernel has a $4 \times 4$ halo, as calculated by subtracting the input step (1,1) from the input size $(5 \times 5)$). The size of the output can then be calculated by multiplying the iteration size $(96 \times 96)$ by the output size $(1 \times 1)$, which for this example will be $96 \times 96$, at the input rate of 50Hz. This output size can then be propagated to the next downstream kernel to determine its iteration size and rate. Once this information has been propagated through the application, the required iteration sizes, data sizes, and rates are known at each kernel.

### B. Buffering

The only channel buffering implicitly included in the application model comes from the single iteration buffers in each kernel input and output. The calculation of iteration sizes at each kernel in the application enables the automatic insertion of buffers to match the output data size of the source kernel to the input size of the sink kernel. This can be seen in Figure 2, where the data coming out of the application's data input is in $1 \times 1$ chunks, while the median kernel needs to process it in $3 \times 3$ chunks. This implies that a buffer must be inserted to buffer a sufficient number of rows of the input data to allow the kernel to run.

The buffering is accomplished by inserting a parameterized buffer kernel, which is a regular computation kernel that implements a two-dimensional circular buffer. The buffers are sized such that they can double-buffer the larger of either the input or output, since the two may be differently sized. The required buffer size is readily determined from the parameterization of the input and outputs and the data-flow analysis. (See [12] for more details.) The result of automatically buffering the application in Figure 2 is shown in Figure 3.

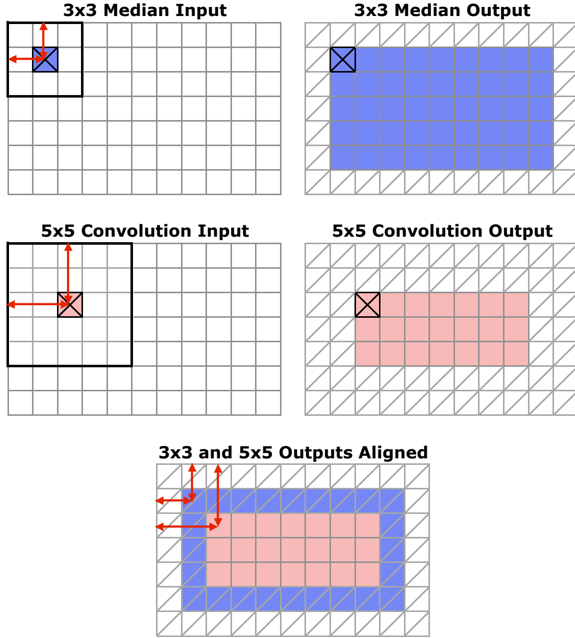



Figure 8. The different insets resulting from the convolution and median kernels are shown here overlaid. For the subtraction kernel to be consistent, both inputs to it must be of the same size. This requires either insetting the larger median output or zero-padding the input to the convolution filter to make its output larger. The resulting application with an automatically inserted inset kernel is shown in Figure 3.

### *C. Trimming and Padding*

In addition to calculating the iteration size and rate at each kernel, the application analysis must determine how much each output is offset from the original application input. This consists of propagating each input's offset through the application, and is necessary to detect when data is unaligned across multiple inputs. For example, the output from the convolution and median kernels in Figure 2 are of different sizes due to their different halos (Figure 8). Feeding these two different sized results into the inputs to the "subtract" kernel is inconsistent, as the "subtract" kernel takes a per-pixel difference, and therefore requires that the inputs be the same size.

In order to automatically adjust the application to correct this inconsistency the compiler needs to know both the different sizes of the output, and how they relate to any shared inputs that generated them. The compiler can then choose to either zero-pad or mirror the input to the convolution filter to make its output larger, or discard output from the median filter to make its output smaller. In this case the insets for the median and convolution filters are (1,1) and (2,2), so the compiler can determine that the outputs should line up as shown in Figure 8. From this analysis the compiler can either pad evenly around the input to the convolution filter by 1 pixel on each side, or trim 1 pixel of the output from the median filter on each side. The result of this transformation can be seen in the added inset kernel (inverted house) in Figure 3. The choice as to whether to pad or trim must be made by the programmer as it effects the final result, but the details can be handled automatically by the compiler.

### *D. Feedback*

Implementing support for feedback in the framework described here requires two main modifications: changing the data-flow analysis to handle loops in the application graph and providing the programmer with the ability to define the initial values for data held in the feedback loops. The modifications to the data-flow analysis can be accomplished by breaking the feedback loops in the graph using special feedback kernels and/or by using a work-list to traverse the graph. Providing the initial values for a feedback loop can be accomplished by using an initialization kernel which outputs the initial values once and then passes on its input values thereafter. Such modifications would enable feedback at the cost of a more complex application analysis.

## IV. Parallelization

From the kernel resource parameterization, the rate information gathered from the data flow analysis, and the resources available on each processing element, the degree of parallelization required for each kernel to meet the real time requirements can be determined. To a first-order this calculation simply takes the required execution rate for the kernel (from the data-flow analysis) times the resources required per iteration (computation cycles and memory) and divides it by the resources provided by a processing element in the chosen architecture. This gives a rough estimate of the number of kernels that must be run in parallel to meet the required rates. However, in order to parallelize the kernels appropriately, the compiler must insure that the data is distributed to the parallel kernels and collected in order.

### *A. Data-parallel Kernels*

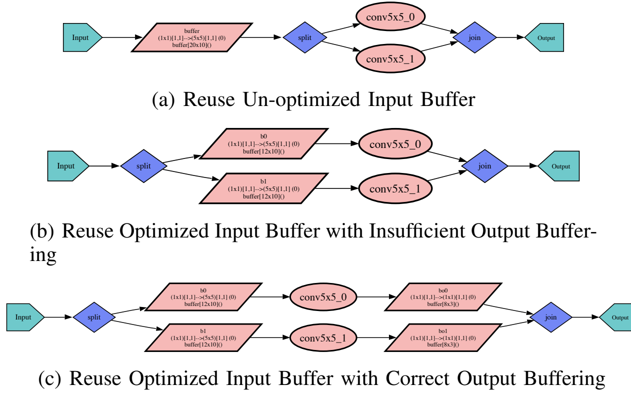
For fully data-parallel kernels, parallelization is quite straightforward: the kernels are replicated and data is distributed to them in a round-robin fashion. To implement this, the compiler replicates the computation kernel as needed, and adds the replicated kernels to the application graph using *split* and *join* kernels to appropriately distribute and collect the data from the now-parallelized computation kernels.

The split and join kernels are regular kernels that implement a finite state machine for collecting or distributing data. In the case of data-parallel kernels, the split kernel simply sends each input to the appropriate instance of the parallelized kernel in round-robin order, and the corresponding join kernel collects the results from the kernels in the same order. While this simple-minded approach is correct, it ignores the possible data reuse that can occur at the computation kernel if iterations are executed in order. However, to take advantage of this reuse, sufficient data must be buffered on both sides of the kernel to ensure that all of the parallelized kernels can run at the same time, or the application will not meet its real-time requirement. This latter transformation is described in Figure 9, but was not implemented for the results presented here.

### *B. Limiting Data-Parallelism with Data Dependency Edges*

By default all kernels in the application graph are assumed to be data-parallel. However, the application graph allows the addition of *data dependency edges* which enable the programmer to explicitly limit the allowed degree of parallelism in an application.

For example, for the histogram kernel in Figure 1(b), multiple instances of the kernel may be instantiated and build up partial




(a) Reuse Un-optimized Input Buffer

(b) Reuse Optimized Input Buffer with Insufficient Output Buffering

(c) Reuse Optimized Input Buffer with Correct Output Buffering

Figure 9. Replicating the input buffer to a kernel sufficiently can enable higher reuse of the data from the buffer to the kernel for windowed filters. However, such optimizations may require parallelizing the buffers more than is needed given the processor resources. In order to ensure that the kernels do not stall, however, sufficient output buffering must also be provided to enable each kernel to run continuously. More detail can be found in [12].

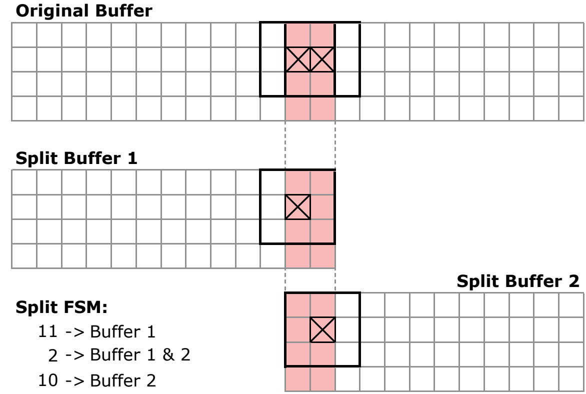



Figure 10. Parallelizing a buffer requires replicating data between the parallelized buffers. Here the data shared between the last output of the first buffer and the first output of the last buffer (shown shaded) is replicated between the two due to the kernel's need to access the overlapping data. The FSM for the split kernel is shown, demonstrating that 2 samples for each line are sent to both buffers.

histograms in parallel, but the final reduction step of merging them into one histogram must be done serially, once per frame. This limited-parallelism, or data-dependency, is specified in the application graph by adding a data dependency edge from the input to the application to the histogram's merge kernel, as shown in Figure 1(b), thereby limiting the parallelism of the sink (the merge kernel) to that of the source (per-frame input), as seen in the parallelization in Figure 4.

Data-dependency edges can also be used to define pipelines of kernels where the parallelism of the kernels within the pipeline is limited, but where the whole pipeline can be parallelized as required. This can be done by adding a data-dependency edge between each kernel in the pipeline, but allowing the first kernel in the pipeline to be data-parallel. The compiler can then instantiate multiple instances of the first kernel to meet the computation requirements, but the data-dependency edges will limit the parallelization of the subsequent pipeline stages, leading to the creation of multiple parallel pipelines.

### C. Non-data-parallel Kernels

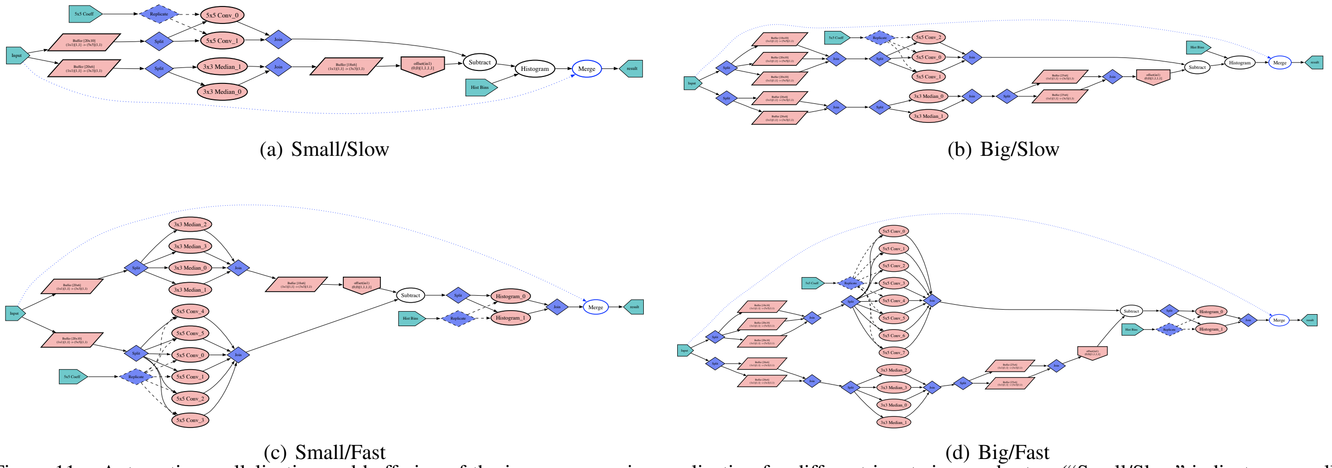
Unlike data-parallel kernels, parallelization and data distribution and collection for non-data-parallel kernels is not straightforward. The compiler handles such kernels by allowing the programmer to specify how a given kernel should be parallelized either manually (e.g., instantiating and connecting kernels when defining the program) or programmatically (e.g., by providing a routine that implements the parallelization in a parameterized fashion).

One example of this flexibility is in the parallelization of buffer kernels. While these kernels are unlikely to be CPU-bound (as they do very little computation) they are likely to be limited by the available storage at a the processor element. This results in the need to split a buffer across multiple processing elements to allocate sufficient storage. However, distributing data to the buffers in the same round-robin fashion as is used for data-parallel kernels would result in out-of-order data and incorrect execution. Instead the buffers need to be parallelized by splitting them in a column-wise manner, and possibly replicating shared data at the split point. This is illustrated in Figure 10, where the split kernel for the parallelized buffer must send the data shared between the last output of the first buffer and the first output of the second buffer to both. The buffer splitting is implemented by providing a specialized parallelization routine to the compiler that is called when buffers need to be split.

### D. Results of Automatic Transformations to Meet Real-Time Constraints

The image processing application from Figure 1(b) is shown automatically parallelized to meet the requirements imposed by a variety of different input sizes and rates in Figures 11a-d. As the input *size* is increased ("*Small*/Slow" to "*Big*/Slow") the required buffering increases, and the buffers are automatically replicated to handle the larger input within the fixed resources of the target architecture. When the input *rate* is increased ("Small/*Slow*" to "Small/*Fast*") the computation kernels are automatically parallelized to handle the increased computation requirement. And, as expected, when both the input rate and size are increased ("Big/Fast") the resulting application has both more buffers and computation kernels.

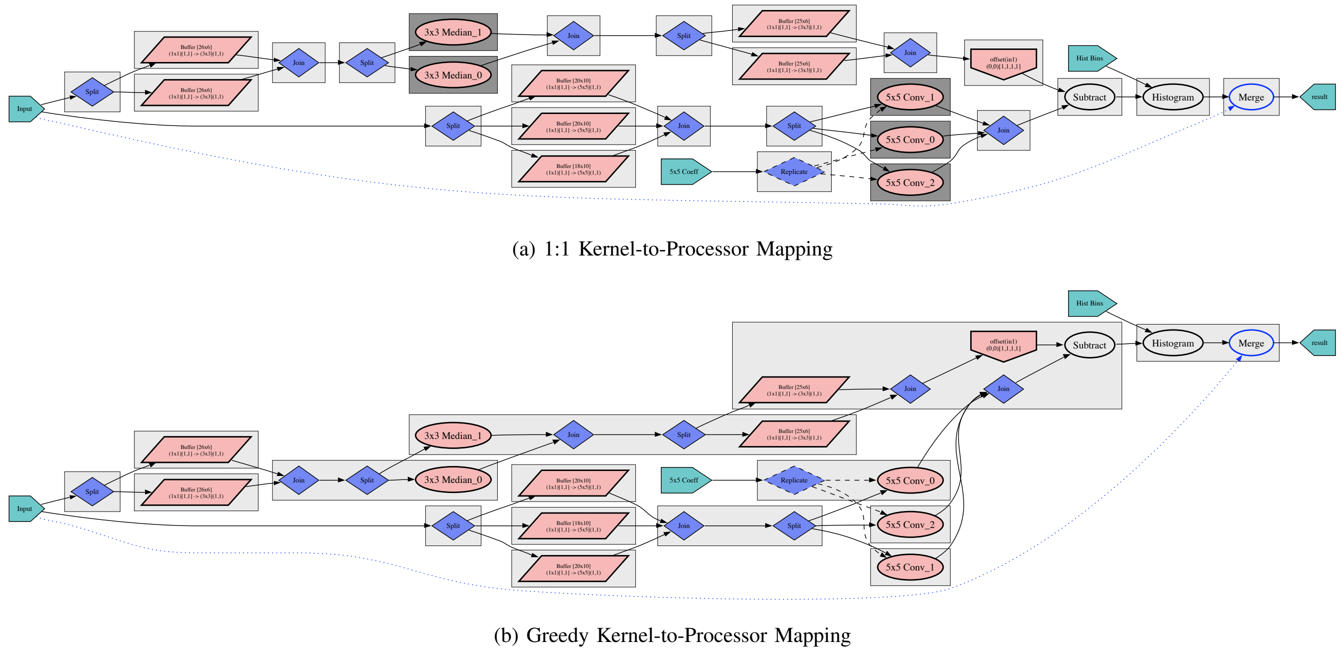
The four different parallelizations in Figure 11 were simulated to verify that they meet their real-time constraints on a functional simulator that took into account execution time, data access time, buffer transfer time, and scheduling, but not placement and communication delays. This is a reasonable simplification for a throughput-based application where communication delays will only increase the latency for the first output, but will not impact the throughput. The problem of optimal placement is complicated by the interaction between parallelization and placement decisions. (E.g., increasing the number of kernels beyond what is required to meet the real-time rate may allow a more optimal placement, resulting in a lower overall energy consumption.) Several of the related tradeoffs in this regard are discussed in [13]. A simulated annealing approach to placement has been implemented, but not integrated within the simulator.




(a) Small/Slow (b) Big/Slow

(c) Small/Fast (d) Big/Fast

Figure 11. Automatic parallelization and buffering of the image processing application for different input sizes and rates. ("Small/Slow" indicates a small input at a low rate; "Big/Fast" indicates a large input at a high rate.) As the input size is increased, the required buffering increases and the compiler automatically replicates them. As the input rate is increased, the required throughput increases and the compiler automatically replicates the computation kernels.




(a) 1:1 Kernel-to-Processor Mapping

(b) Greedy Kernel-to-Processor Mapping

Figure 12. Kernel-to-processor mappings for the application from Figure 4. Each box encloses the kernels that will run on a single processor core. The high-utilization computation kernels in Figure 12(a) are shown with dark backgrounds. In Figure 12(b), the low-utilization kernels have been merged to run in a time-multiplexed manner with other kernels on a single core to increase overall utilization. The initial input buffers are not multiplexed because they may block the input if they are not serviced in time to handle the input.

## V. MULTIPLEXING

The parallelization techniques discussed in Section IV result in an application that is properly parallelized to meet the real-time requirements of the inputs given the specifications of the kernels and the target architecture. However, with a naive 1:1 kernel-to-processor mapping, the processors are inefficiently utilized. The reason for this can be seen in Figure 12(a) where a large number of low CPU-utilization buffers and split/join kernels have been inserted into the application. When these are each mapped to their own CPU, the overall utilization, and hence efficiency, of the application decreases.

To overcome this inefficiency, a simple algorithm was implemented which attempts to greedily time-multiplex low-utilization kernels on a single processor. The algorithm looks at neighboring kernels and merges them onto the same processor if their combined CPU/memory utilization does not exceed that of the processor. For the example discussed here, this increases the CPU utilization from 20% to 37%, resulting in the kernel-to-processor mapping as shown in Figure 12(b). This simple algorithm improves the utilization by $1.5\times$ across a variety of test programs ranging in size from fewer

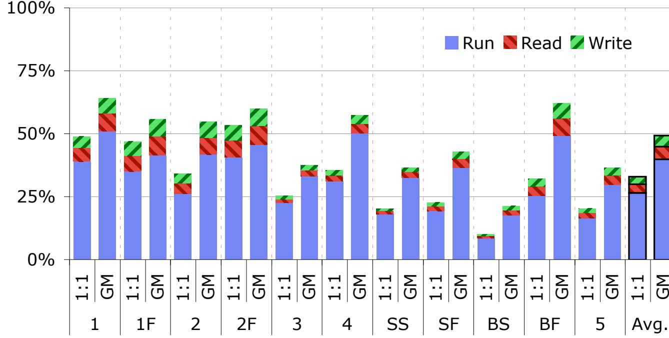



Figure 13. Processor utilization for applications mapped to multiple processor cores using one-to-one kernel-to-core (1:1) and greedy mappings (GM). Utilization is broken down by kernel execution time (run) and time spent accessing kernel inputs/outputs (read/write). Average utilization improvement is 1.5× for the greedy mapping over the 1:1 mapping. (*Benchmarks:* 1 & 1F: Bayer demosaicing with baseline and faster input rates; 2 & 2F: Image histogram with baseline and faster input rates; 3: parallel buffer test; 4: multiple convolutions test; SS, SF, BS, BF: image processing example (Figure 11) with small/big input size and slow/fast input rates; 5: application from Figure 1(b).)

than 10 kernels to more than 50, as seen in Figure 13.

## VI. Related Work

The most similar programming system to the one presented here is the MIT StreamIt[7] system developed for the Raw [15] processor. This programming model set out to explicitly develop an efficient, easy-to-use system that would allow programs to be automatically mapped to many processors. The compiler systems for StreamIt explored a variety of sophisticated application-level and kernel-level optimizations for throughput, communications, and load-balancing.

StreamIt and the Raw backend focus on a slightly different problem than the one chosen here. Rather than finding the minimum number of processors to meet a fixed rate, they try to use a fixed number of processors to obtain the highest rate possible. Here the minimum number of processors is set by the real-time requirements, but the optimal number may be greater. StreamIt further provides explicit hierarchy in its applications through the use of pipeline, feedback, and splitjoin constructs. These constructs were intended to provide an intuitive textual program description, but it is unclear how useful this truly was as later generations of the compiler spent a great deal of effort determining how to appropriately flatten the hierarchy.

The StreamIt programming model suffers from two deficiencies: it allows only a single input/output pair for each kernel, and it permits only one-dimensional data. These two design choices make it unintuitive to write many basic applications, in particular anything that manipulates two-dimensional images or uses multiple streams of data. Furthermore, and perhaps more importantly, the contortions needed to map image processing to a one-dimensional data stream and manually multiplex multiple data streams into one prohibit the compiler from readily analyzing the data movement and usage.

Along similar lines, the StreamC/KernelC [6] language developed for the Stanford Imagine [16] project suffered from its use of one-dimensional data. While it did allow multiple stream inputs and outputs for each kernel, its target architecture forced it to take an explicitly SIMD approach to processing. GPGPU languages, such as Brook [17], have similar single-kernel, SIMD-style approaches to throughput. This focus makes it difficult to take advantage of pipeline and task-level parallelism in the application, and instead shifts the focus to data staging to keep a single kernel maximally productive. This model is a poor match for the MIMD capabilities of many-core architectures, and the diverse types of parallelism found in embedded applications.

The synchronous data flow (SDF) [5] approach has been very successful in frameworks such as MATLAB's Simulink environment, particularly for targeting DSPs, but primarily for one-dimensional data. The extensions to SDF to multiple dimensions in the form of MDSDF [8], GMDSDF [11], and Array-OL [4] are more general, including explicit support for re-sampling and complex access patterns within the data, at the cost of significantly more complex compiler analysis. Windowed SDF [9] introduces a simpler approach that provides significant flexibility without much of the complexity of the other approaches. However, with these approaches the compilation system must both analyze and understand a wide range of data access and tiling patterns and also determine the correct ordering for the data and processing. This extensive flexibility does enable very sophisticated application description, but its lack of adoption and exploration suggests that it is not a good tradeoff for the complexity it incurs in compiler analysis. Similar problems are found with the cyclostatic extensions to SDF which enable control flow within the application, but at the cost of exponentially increased complexity.

Apart from work on languages there has been much work on optimal assessment and mapping of task and data-parallel computations. Subholk and Vodran have presented a dynamic programming approach to determine the optimal division of data- and pipeline-parallelism in an application to meet throughput and latency constraints [18]. This work uses a slightly more restrictive programming model of a single pipeline of tasks, but demonstrates a more complete analytical analysis than that presented here or in StreamIt [13].

## VII. Conclusions

This work uses a high-level parameterized description of the application in terms of computation kernels, data streams, dependencies, and control patterns to enable automatic application manipulation and parallelization. By parameterizing the input and output data and choosing a scan-line input order, the analysis of data use, reuse, and movement is significantly simplified, while remaining relevant to many image- and signal-processing algorithms. The effectiveness of this approach can be seen in the ability to automatically adjust programs to meet different real-time requirements, and the intuitive manner in which control and serialization are handled through control tokens and multiple kernel methods. These capabilities enable the programmer to focus on the application and leave the details of parallelization for a particular set of processors to the compiler system.

For this approach to be more broadly applicable, however, the requirement that all data sizes, rates, and kernel resources be statically known at compilation time must be addressed. This limitation makes analysis straight forward, and enables an accurate hard real-time guarantee, but inhibits a range of useful applications.

A canonical example might be a motion vector search, where the number of motion vectors, the data required to process them, and the processing time per motion vector vary from frame to frame. Incorporating such a kernel into this framework requires extending the system to support bounds on real-time processing requirements and runtime exceptions to indicate when a kernel has exceeded its allocated resources.

While there is still much work to be done to make this programming model sufficiently general to handle the broad range of embedded applications in use today, the lesson from this system is significant: by exposing the application structure and data movement at such a high-level, the task of mapping processing and communications to an array of processors can be greatly simplified. This can ensure efficient usage of the distributed computational and storage resources on many-core chips, and by making real-time constraints an integral part of the program description, the programmer can be relieved of the task of having to understand the details of the hardware to ensure real-time performance.

## ACKNOWLEDGMENT

This work was funded by SRC Contract 2007-HJ-1591, Intel, and the Stanford Center for Integrated Systems, with additional support provided by the CoDeR-MP project.

## REFERENCES

[1] M. Lapedus. (2007, March) Sockets scant for costly asics. [Online]. Available: http://www.eetimes.com/showArticle.jhtml?articleID=198500400

[2] (2008, June). [Online]. Available: http://www.clearspeed.com/products/csx700.php

[3] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, “On-chip interconnection architecture of the tile processor,” *Micro, IEEE*, vol. 27, no. 5, pp. 15–31, Sept.-Oct. 2007.

[4] A. Amar, P. Boulet, and P. Dumont, “Projection of the Array-OL specification language onto the kahn process network computation model,” *Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 6 pp.–, 7-9 Dec. 2005.

[5] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[6] P. Mattson, “A programming system for the imagine media processor,” Ph.D. dissertation, Stanford University, March 2002.

[7] W. Thies, M. Karczmarek, and S. P. Amarasinghe, “Streamit: A language for streaming applications,” in *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 179–196.

[8] E. Murthy, P.K.; Lee, “Multidimensional synchronous dataflow,” *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 50, no. 8, pp. 2064–2079, Aug 2002.

[9] J. Keinert, C. Haubelt, and J. Teich, “Windowed syncrhonous data flow (WSDF),” University of Erlangen-Nuremberg, Institute for Hardware-Software-Co-Design, Germany, Technical Report 02-2005, 2005.

[10] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe, “MPEG-2 decoding in a stream programming language,” *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, 25-29 April 2006.

[11] P. Murthy and E. Lee, “An extension of multidimensional synchronous dataflow to handle arbitrary sampling lattices,” *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, vol. 6, pp. 3306–3309 vol. 6, 7-10 May 1996.

[12] D. Black-Schaffer, “Block parallel programming for real-time applications on multi-core processors,” Ph.D. dissertation, Stanford University, April 2008.

[13] M. I. Gordon, W. Thies, and S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 151–162.

[14] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, “Teleport messaging for distributed stream programs,” in *PPoPP ’05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2005, pp. 224–235.

[15] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, “Baring it all to software: Raw machines,” *Computer*, vol. 30, no. 9, pp. 86–93, Sep 1997.

[16] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner, “Imagine: media processing with streams,” *Micro, IEEE*, vol. 21, no. 2, pp. 35–46, Mar/Apr 2001.

[17] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: stream computing on graphics hardware,” *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[18] J. Subholk and G. Vondran, “Optimal use of mixed task and data parallelism for pipelined computations,” *Journal of Parallel and Distributed Computing*, no. 60, pp. 297–319, 2000.