Expression of Software Modality and Thread Control Using a Stream Processing Language

Abstract

Stream processing languages have been used effectively to specify parallel processing on modern hardware with high efficiency. The concept’s key benefits are the abundant use of efficient compute-intense vector operations to achieve high performance, the expression of data dependence and locality to aid in memory management, and the exposure of data and thread parallelism in the language. However, the application domain of these languages has been seen as limited to signal processing, and the languages have not been used for general purpose programming. In traditional stream processing, streams of data are acted upon by kernel functions. We have extended stream processing with a highly expressive type system to define the port characteristics for the streams. These port characteristics allow for a compact means of expressing software control. Included in these port characteristics is the means for specifying the runtime management of streams. If chosen, the developer can programmatically determine how much data is required for a kernel to execute and how much data is produced and consumed on each stream for each execution. These dynamic streams become thread boundaries in the generated application. Port characteristics are also included for managing the software state for modal software. The expression of stream segmentation, that is breaking an infinite data stream into finite length portions, allows the developer to specify when software modes begin and end, and the language provides a means for specifying how the state is managed when transitioning between modes of operation. This paper will describe how the addition of new port characteristics to stream processing is able to extend the number of problems that can be addressed with the concept and discuss how the compiler is designed to transform programs into efficient MIMD-parallel applications.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language classifications—concurrent, distributed, and parallel languages; data-flow languages.

General Terms Performance, Design, Languages.

Keywords stream processing; dynamic data-flow; data-flow; modal processing; stream segmentation

1. Overview of Stream Processing

Traditional programming languages like C and C++ are geared towards von Neumann architectures that provide a sequential program order with a single arithmetic logic unit (ALU) and a single monolithic memory unit. Single chip processors matured and evolved, yet programming was predominantly done using von Neumann languages. As Moore’s Law reaches its physical limitations, multicore and multiprocessor systems are becoming more commonplace. Many programmers continue to use traditional programming languages to code for these architectures, building the software architecture into the code. We define the software architecture as the organization of the software into components and the interface between those components, whether it be on a single processor or across a bus or grid interconnect. Because it is not natural to use von Neumann-based tools to program these new hardware systems, it is difficult to achieve high productivity while programming, and, once the software is created, it is difficult to manage the software, perform upgrades, and perform hardware refreshes.

Because von Neumann languages are not a natural method for programming multiprocessor systems, stream processing is being considered as an alternative programming language paradigm. Stream processing is a data-centric programming model. Given a set of input data (streams), a series of compute intense operations (kernels) are applied to each token in the stream. Programs are constructed by creating the kernels that act on the streams, and organizing the data-flow through the kernels, as shown in Figure 1.

Stream processing provides a convenient method for specifying distributed processing because the concurrency is easily visible, both to the programmer and the compiler. In the example in Figure 1, kernel 3 can be applied concurrently with kernels 1 and 2 because it requires no data from kernel 1 or 2. Also, this specification of data dependency can be used to predefine the order of execution of kernels on a single processor, code generating the set of kernels into a thread of execution. In the example in Figure 1, kernel 1 needs data A and data B in order to execute, kernel 2 needs kernel 1’s output to execute, etc. A stream processing compiler uses this data dependency to organize and optimize the structure of the application.

Another aspect of the stream processing specification that can be utilized by the compiler is data locality. The data-flow defined in Figure 1 also specifies when stream data can be discarded. Once kernel 1 is done processing data A and B, the data is no longer needed, and new tokens can be accepted on those inputs. Using this data locality, the memory use over the execution of this thread can be preplanned.

Figure 2 shows how the execution of the data-flow from Figure 1 can be preplanned by the compiler. The first column in the table identifies the kernels from the original flow graph. A, B, and C are used to identify the source kernels that form the streams, e.g., reading from disk or an analog-digital converter’s buffer. S identifies the data sink, e.g., displaying the data. The figure shows the order of kernel execution vertically and the memory buffers used during each execution horizontally. The downward pointing arrows indicate the data must be held in the buffer for a later kernel execution. For example, the data output

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Figure 1. Stream processing involves specifying kernels which operate on streams of data.
from kernel 2 must be held for the execution of kernel 4 while kernel 3 executes. Through analysis of the data-flow, an implementation is created which ensures correct results using the smallest memory footprint possible. 

Stream processing also allows for the specification of state data. Kernels can have state or be stateless. A simple example of a kernel with state is a ramp data generator

$$\text{out}(i) = \text{out}(i-1) + K$$

which is not output. This state data must be managed and maintained during execution.

While stream processing is seen as a new technology, it is very closely related to data-flow languages that have been developed from the 1970s and 1980s through today, as surveyed in [1]. We have developed a data-flow language for specifying a wide range of software behaviors, targeting multicore or multiprocessor systems. Regular, deterministic, synchronous flow of data is defined as static data-flow. While the use of static data-flow to specify stream processing provides efficient implementation of many applications, its scope is often seen as limited to signal processing. Static data-flow is a limitation in problems that require runtime variability and reconfigurability and prevent stream processing languages from being considered in a wider application space. This paper presents how the concepts of stream processing and data-flow languages have been extended by developing a type system for specifying the port characteristics for the inputs and outputs of the kernels and how this port information is used to automate the global structure of the generated application, making it applicable to a wider range of problems. We will illustrate how this type system provides the expression of software modality, allowing the compiler to manage the use of resources and sharing between modes.

2. Dynamic Data-Flow

Static data-flow requires the usage of each data stream to be specified before compilation to allow this information to be used during the generation and optimization of the application. Dynamic data-flow loosens this restriction by allowing the developer to programmatical set the number of tokens produced or consumed on each output or input that is declared as dynamic. When specifying a dynamic stream, the developer only specifies a threshold for execution. At development time, the programmer specifies the number of tokens that must be available on the inputs (and/or empty space on the outputs) in order for the kernel to execute, and at runtime, the program must specify how many tokens are produced or consumed. [2]

The programming models for static and dynamic data-flow can be utilized together. [3] [4] If a set of kernels are connected and use only static streams, then their execution and memory use can be preplanned into a single thread. If a kernel is added to this set with a dynamic stream on one of its ports, then this dynamic stream becomes a barrier between connected components. Everything upstream from the dynamic stream can be compiled into a single thread, and everything downstream from the dynamic stream can be compiled into a separate thread. This natural decomposition into threads is shown in Figure 3. The red line indicates a dynamic output in kernel 2. Because of this boundary, kernels 1 and 2 form one thread, and kernels 3 and 4 form a second thread.

The introduction of dynamic streams and the creation of multiple threads require that a runtime component be added to the application to manage the threads during execution. This thread scheduler must manage the dynamic streams and determine which thread will execute next based on the status of the data queues associated with each stream. Further, the thread scheduler must determine the processing granularity, that is, how many tokens are processed by the thread during each execution. Static data-flow allows the specification of the processing granularity during compilation, and the compiler scales, or flexes, the size of the memory allocated to compensate for the number of tokens specified. Dynamic data-flow is still structured to process an arbitrary granularity, however this amount is now dependent on upstream threads and can change during runtime. In Figure 3, the number of tokens thread B processes is dependent on how many tokens thread A produces on kernel 2’s dynamic output. Thread A may produce more tokens one execution than it does on another execution. The thread scheduler must flex the granularity of thread B at runtime to compensate for the tokens available from thread A using the process shown in Figure 4. This flexing involves scaling the memory offsets encoded in the thread to process the data available.

Figure 3. The use of a dynamic stream, such as the output of kernel 2, forms a natural boundary between threads.

Figure 4. Runtime management of dynamic data-flow includes flexing downstream threads based on the amount produced by source threads.
3. Variable Threshold Data-Flow

These restrictions of static data-flow can also be loosened by allowing the threshold for execution to be set at runtime. This variable threshold data-flow adds another aspect of runtime queue management on top of dynamic data-flow. If an input or output stream is marked as variable threshold, the only requirement of the queue that is set is its size. The kernel that uses the stream may execute regardless of whether data is available on the queue or not, and the kernel’s function must query the thread scheduler for information about the queue.

The protocol for using variable threshold streams is shown in Figure 5. During the first execution of the kernel, the programmer has access to data on all the static and dynamic streams. He may also query the thread scheduler for the amount of data available on the variable threshold streams. From this information, the program must determine how much data is required on the variable threshold streams to complete execution of the kernel. This amount is specified to the thread scheduler, and the kernel does not execute again until that requirement is satisfied. When the kernel is re-invoked, the program uses the data on the variable threshold stream and informs the thread scheduler how much data is produced and consumed on those streams, as well as any dynamic streams.

The inclusion of dynamic and variable threshold streams in a stream processing language enables many more applications to be developed. A key construct that can now be specified directly in the language, and thus open to optimizations from the stream processing compiler, is branch and merge, as shown in Figure 6. In traditional programming languages like C, a branch and merge operation is implemented as a conditional, e.g.,

```
if (flag) { out = AlgorithmA(in); }
else { out = AlgorithmB(in); }
```

However, this conditional cannot be specified in static data-flow because the flow of data from stream in to AlgorithmA or AlgorithmB cannot be determined until runtime when the value of flag is checked.

To implement the branch and merge construct using stream processing, dynamic outputs are used in the branching kernel and variable threshold inputs are used in the merging kernel. The two-output branching kernel executes when both its output streams are empty and it has tokens on its input data stream and flag stream. At runtime, the kernel’s function determines based on the flag stream which dynamic output stream to produce tokens on. The function of the branching kernel can be specified in C code as

```
N0=N1=0;
for (i=0; i<granularity; i++) {
    if (flag[i]==0) {
        *out0++ = in[i]; // copy to output 0
        N0++;
    } else {
        *out1++ = in[i]; // copy to output 1
        N1++;
    }
}
produce(out0,N0); // produce data
produce(out1,N1);
```

The merging kernel can execute whenever its static output stream is empty; because the inputs are both variable threshold, the developer must query the thread scheduler at runtime to determine the data availability on the streams. When the function first executes, the function scans the flag stream to determine how many tokens are coming from each of the input data streams and informs the thread scheduler of these amounts. When these queue requirements are met, the function executes again and merges the data onto the output stream. The function of the merging kernel can be specified in C code as

```
if (!waiting_for_thread_scheduler) {
    for (i=0; i<granularity; i++) {
        if (flag[i]==0) N1++;
        else N0++;
    }
    amount(in0,N0); // specify data requirements
    amount(in1,N1);
    waiting_for_thread_scheduler=1;
}
if (queue_ready) { // set by thread scheduler
    for (i=0; i<granularity; i++) {
        if (flag[i] == 0) {
            out[i] = *in0++;
        } else {
            out[i] = *in1++;
        }
    }
    consume(in0,N0); // consume data
    consume(in1,N1);
    waiting_for_thread_scheduler=0;
}
```

4. Segmentation

Augmenting stream processing’s port classification with dynamic and variable threshold streams greatly increases the types of applications that can be expressed using stream processing. It
provides fine control – at the token level – over the execution of kernels based on data availability. For each token on the flag stream, the branching kernel determines which branch the data will pass through, and the merging kernel determines which branch the data is coming from. Coarser control over streams allows for the compact expression of complex software structure.

A stream segment is defined as a finite length portion of an infinite data stream. If a stream is noted as a segmented stream, the programmer can place markers on the stream to indicate boundaries between the segments. When downstream kernels receive these markers, side effects are invoked to manage the segment. With this concept of segmentation we break away from traditional stream processing and data flow languages that do not generally allow for side effects and transients. These side effects provide a method for programming state and state transition. They also provide a method to specify iteration of arbitrary length, producing tokens at the end of the segment.

At the beginning of a segment, functions are invoked to reset the state of the kernel. For example, if a ramp is used to increment over time, the kernel can be constructed to reset the counter to zero or another initial value at the beginning of a new segment. Each kernel has two methods for initialization, one method that occurs at application startup and one method that occurs at the startup of each segment in the stream.

Functions are also invoked at the end of the segment. These end-of-segment functions can be used to produce data cumulated over the duration of the segment. For example, if a kernel is used to calculate the mean of a segment, variables are added to the state of the kernel to store the tally of the values and a counter for the number of values. The reset function will reset the tally and count

\[
\text{Tally} = 0; \\
\text{Count} = 0;
\]

The steady-state function will increment the tally and count

\[
\text{for (i=0; i<\text{granularity}; i++) \{ \text{Tally} += \text{in}[i]; \}} \\
\text{Count} += \text{granularity};
\]

And the end-of-segment function will calculate the average and produce the result on the output
\[
*\text{out} = \text{Tally}/\text{Count}; \\
\text{produce(out,1); // produce 1 token}
\]

To create a segmented data stream, we mark the outputs of a kernel as segmented. Segmented streams are dynamic. Segment markers are placed on the outputs in the kernel function, and the location of the markers are defined by the next occurrence of a produce. For example, if segments are broken apart according to a flag stream, the segmenter can be specified in C code as

\[
\text{N}=0; \\
\text{for (i=0; i<\text{granularity}; i++) \{ } \\
\text{out}[i] = \text{in}[i]; \\
\text{N}++; \\
\text{if (flag[i]) \{ } \\
\text{segment(out,SEGMENT_END); // add marker } \\
\text{produce(out,N); // produce with end marker } \\
\text{N} = 0; // reset counter } \\
\text{\}} \\
\text{produce(out,N); // produce with no marker}
\]

The key utility of segmentation is that it provides a method for structuring modal software. Software is often based on distinct processing modes. A simple example of such modal behavior is a radar system that switches between search mode and tracking mode as targets are located, as shown in Figure 7. Once a target is found, the segmenter ends the stream segment directed towards the Search Mode kernels and begins the stream segment directed towards the Track Mode. In complex software systems, the system may have dozens of modes, including sub-modes, forming a deep hierarchy, and levels of segmentation can be readily nested to form this structure.

The scope of the segmentation is defined by the hierarchical structure of the application. That is, only the subgraph or kernel immediately downstream (connected by a single arc) from the segmented output are affected by the segmentation. Fan-out from the segmented output is supported, and this fan-out allows for the specification and creation of modes.

Figure 7 depicts two subgraphs, Search Mode and Track Mode, immediately downstream from a segmenter. The segmentation does not apply to the Merge kernel. Because the segmented output must be a dynamic stream, this output creates a thread boundary during application generation. The Search and Track Modes become separate threads in the application. The Segmenter branching kernel and the Merge kernel can be in the same thread or different threads, depending on the desired functionality and distribution. As possible with any thread, these mode threads can be further decomposed into threads for parallel execution, allowing each mode to execute on multiple cores.

Nested segmentation can be employed to mirror the hierarchical structure of the modes.

5. Exclusion and Shared State

Segmentation provides a means of structuring modal software, but the concept must be carried one step further in order to create a state machine-like behavior when transitioning between modes. Segmentation defines when an output is actively producing data, i.e., when software is in a certain state. If each output stream is associated with a different software state, an end-of-segment marker defines when the software leaves the state associated with that output, and the software does not re-enter that state until it produces data – thus beginning a new segment – on the stream. However, segmentation alone does not prevent the programmer from starting segments on multiple outputs at the same time. In fact, the ability to create segments of different lengths on different streams is essential to many algorithms that can be programmed with segmentation.

Exclusive segmentation (or exclusion) is defined as segmentation where only one output stream is active at a time. So, for example, exclusive branches will never simultaneous change state
The compiler can take advantage of this explicit exclusivity in constructing the application. Resources can be shared between software modes. When planning memory usage, the compiler can allocate space sufficient for the largest mode and reuse this space for each of the modes.

6. Implementation Specification, Compilation and Runtime Management

The applicability of this stream processing language to multicore and multiprocessor systems depends on three other components, a separate language for specifying the implementation of the application, a compiler to combine the implementation with the stream processing language and create the implementation, and a runtime thread scheduler. The process of creating an application using these components is shown in Figure 8.

The separation of the stream processing language and the implementation language is a key feature to the language. The implementation language contains many parameters to aid and guide the compiler, including but not limited to the mapping of kernels to processors, the mapping of streams to physical memory, the strip mining of memory (e.g., tiled processing) to increase cache performance or target resource-limited processors, the selection of transfer methods between processors. Traditional programming languages require these choices and optimizations be built into the code, reducing portability.

The compiler is tasked with creating an efficient implementation. The language makes data dependency and locality explicit, as well as exclusion between software modes, and the compiler must analyze this information to optimize the application’s resource utilization. The compiler must form the threads based on the dynamic and variable threshold stream boundaries. Additionally, the mapping of kernels to processors may cause some threads to decompose into several threads. During runtime, the thread scheduler resides on each processing element and manages when each thread is run, including responding to queries from the variable threshold streams and flexing threads based on the produce and consume amounts from dynamic and variable threshold streams.

7. Deadlock Avoidance

The introduction of parallelism and nondeterminism creates additional opportunities for deadlock. The compiler takes several steps to organize the application to avoid deadlock. Static data gain imbalance is detected during scheduling to eliminate it as a potential cause of deadlock. Gain imbalance can occur when multiple streams meet in a kernel. The ratio of tokens received from each stream is defined by the port characteristics of the kernel. In static data flow, the compiler must insure that these ratios are observed. A simple case is shown in Figure 9. An adder is fed by two collections of kernels, one produces 20 tokens per firing and the other 10. The compiler addresses this situation by forcing the second collection of kernels to fire twice per execution of the thread schedule in order to compensate for the difference in natural rates. The sizes of the buffers that store the streams are adjusted to accommodate the extra tokens.

Note there are also cases where the compiler cannot correct the data flow. For example a single source is interpolated by different rates and then fed into an adder, as shown in Figure 10. The compiler cannot affect the rate of the second path without also affecting the rate of the first.

Improper send/receive ordering is another possible cause of deadlock that is avoided through analysis during compilation. Sends and receives are inserted automatically by the compiler. Because the compiler defines the order of execution of kernels in the thread, it can adjust when each send and receive occurs in relation to other sends and receives as well as other kernels. A simple degenerative case is shown in Figure 11. If blocking transfers are used, the right processor is waiting for data B while the left processor is waiting for the other to receive A. Race conditions are also prevented. For example if two processors send data to each other and wait until the other receives the data, neither can make process. The compiler orders the thread schedules to insure cases such as these do not happen.
Stream’s kernel will only execute after upstream threads have produced data.

8. Overhead Analysis

The overhead of using a stream processing language is defined as the amount of time spent managing kernels during runtime. The efficiency is defined as the ratio of the time spent executing kernel code to the total execution time (time spent executing kernel code plus overhead). Note that in practice a stream processing language has the same overhead as any program; the work of managing kernels at runtime is analogous to thread management, and time is spent managing threads whether that control code is autocodered by the compiler or written by the user.

To analyze the overhead for using dynamic and segmented data flow, the runtime thread scheduler is instrumented with timers that measure the time spent executing each part of the runtime kernel, as first presented in [5]. Three benchmark applications are studied, one that is fully static, one with dynamic behavior, and one using segmentation. The first benchmark is a fully static application using 66 kernels that implements the Space-Time Adaptive Processing (STAP) algorithm. The second benchmark implements an Offset Quadrature Phase-Shift Keying (OQPSK) modem simulation which requires three dynamic output streams, one during modulation, one during demodulation, and one during the decoding of bits into characters for output. The OQPSK simulation uses 19 kernels broken into four threads by the three dynamic streams. The third benchmark implements noise removal using dynamic streams, segmentation, and one external state variable.

The three benchmarks were run on Intel x86 single core systems using both Linux and Windows operating systems, a SPARCstation running Solaris, as well as two PowerPC AltiVec embedded systems from different vendors. The results are shown in Table 1.

| Table 1. Overhead Analysis of Three Benchmarks |
|-----------------|-----------------|-----------------|-----------------|
|                  | STAP             | OQPSK           | Noise Removal   |
| Proc             | Eff. % (µs/fire) | Eff. % (µs/fire) | Eff. % (µs/fire) |
| Linux            | 99               | 1.4             | 72              |
| Windows          | 99               | 1.6             | 79              |
| SPARC            | 98               | 1.4             | 79              |
| AltiVec1         | 97               | 1.7             | 90              |
| AltiVec2         | 93               | 1.1             | 92              |

On the embedded PowerPC processors, the overhead for using dynamic data-flow and segmentation is modest when compared to the fully static benchmark. Note that for the Intel x86 systems and SPARCstation, non-real time operating systems are used. As a result, there is limited control over context switches by the operating system, which may have impacted the results. This context switching may explain the extra overhead incurred by managing multithreaded stream processing programs like OQPSK and Noise Removal.

The reduced efficiency of the fully static STAP application on the AltiVec targets may be explained by the enhanced vector efficiency provided by these SIMD processors. If the vectorized code inside the kernel is much faster and the control code is equally fast, the efficiency ratio is lower.

9. Performance Analysis

Benchmarks have been implemented in the language to illustrate its ability in programming modern multicore processors. The presented benchmarks do not utilize the dynamic and segmented
language features which are the focus of the paper. However, if an application is built using stream processing around common algorithms like these, the software development can enjoy the productivity and performance benefits of stream processing along with the automated resource management possible by using the these language features.

The Cell Broadband Engine™ (Cell/B.E.) processor is used to illustrate the capability of the stream processing language. The current implementation of the Cell Broadband Engine (Cell/B.E.) processor combines one Power Processing Element (PPE) with 8 identical Synergistic Processing Elements (SPE). The PPE is a dual-threaded PowerPC core with an instruction set that has been extended to include SIMD (Single-Instruction, Multiple-Data) instructions known as VMX. Each SPE contains a high-speed SIMD processor capable of 25.6 GFLOPs with its own 256 kB local store and DMA engine. The nine cores and the on-chip memory controller and I/O controller are interconnected with the high speed Element Interconnect Bus (EIB). The EIB provides a measured peak bandwidth of over 200 GB/s at 3.2 GHz. [6]

The performance for a matrix multiplication benchmark should approach the maximum FLOP count for the Cell/B.E. hardware. The matrix multiply’s core operation is a multiply-accumulate which lends itself well to the SPE’s VMX SIMD instruction set. To perform a parallel matrix multiply, we tile the matrices and multiply a row of tiles from the first matrix with a column of tiles from the second matrix, accumulating the results. The source and result matrices are stored in system memory, and tiles are brought into SPE local storage to be processed by the SPEs. To improve the performance of the communication, we use a block data layout.

A key issue in implementing this benchmark is overlapping the communication with the processing. The theoretical maximum performance of a single Cell/B.E. chip is 25.6×8=204.8 GFLOPs, and the matrix multiplication algorithm can be coded to achieve very close to this maximal throughput. [7] The implementation of the matrix multiply exceed 150 GFLOPs for 1024×1024 matrices and reaches 193.9 GFLOPs for a 4096×4096 matrix, sustaining a process rate at 95% of the peak performance of the chip.

The second benchmark is a multi-stage algorithm common to the defense market, the Synthetic Aperture RADAR (SAR) algorithm. The SAR algorithm has two key stages: the range processing of the rows of the matrix, and the azimuth processing of the columns of the matrix. The range processing consists of two vector operations: a complex-by-real vector multiply and an FFT. The azimuth processing consists of three vector operations, an FFT, complex vector multiply, and an inverse FFT.

To distribute the SAR algorithm we must add three stages: the partitioning of the data to distribute the rows across the processors, a corner turn of the data (distributed matrix transpose) to transition between range and azimuth processing, and the concatenation of the column-based results. With these added processing requirements due to the distribution, the distributed SAR algorithm is as shown in Figure 12.

A key issue in implementing this distributed SAR algorithm efficiently on this architecture is the implementation of the corner turn. While the EIB provides a high bandwidth between the SPEs for corner turn operations, the data sets typical for a SAR algorithm cannot fit in the local storage of the SPEs. To do the work of a corner turn on a large SAR data set, the results of the range operation must be transferred to system memory, and then – once full columns are available – transferred back to the SPEs in transposition. Because this approach places demands on the bandwidth between the SPEs and system memory as opposed to the relatively fast bandwidth between SPEs, the processing speed becomes IO bound according to sustained throughput through the system memory controller.

A second issue in implementing this algorithm on this architecture is maximizing the amount of work that is computed on the SPE’s. The SPE’s can perform vector processing with high efficiency, so the developer must maximize how much work is done on the SPEs and minimize the interaction between the SPEs and the PPE. As the SPE’s local storage must be used as both program memory and data memory, its size is a limiting factor. Program overhead must be minimized and unused code must be removed so that the SPEs can perform both the compute intense stages of the SAR algorithm – range and azimuth processing. Additionally, the transfers must be constructed to perform multibuffering such that data is consistently available for processing on the SPEs.

The implementation was run on two different architectures to determine the speedup possible for this algorithm on the Cell/B.E. The application was originally developed for embedded Digital Signal Processor (DSP) platforms. This same flow graph was run on a quad DSP board with 500MHz PowerPC AltVec processors. The PowerPC implementation achieved 3 frames per second, while the Cell/B.E. implementation on an IBM QS20 blade server using 8 SPEs achieved 287 frames per second, as first presented in [8]. Further performance results are available in Table 2.

The reliance on system memory to store intermediate results limits further performance improvement. The bandwidth between the XDR storage and the EIB bus is 25.6 GB/s [6]. Sustained measured performance for this algorithm reaches approximately 88% of this bandwidth. An analysis of the transfer times shows a standard deviation of up to 100% of the mean value, providing evidence of contention over the memory controller as all 8 SPEs continuously attempt to put and get data from the XDR memory. The removal of overhead done during the optimization process provides minor savings when compared to the potential performance improvement possible if either the memory controller bandwidth is increased or the SPE local storage is increased.

Table 2: SAR Performance According to the Stage in the Algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>FLOPS</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>71.1 GFLOPS</td>
<td>19.9 GB/s</td>
</tr>
<tr>
<td>Corner Turn</td>
<td>-</td>
<td>22.6 GB/s</td>
</tr>
<tr>
<td>Azimuth</td>
<td>127 GFLOPS</td>
<td>14.3 GB/s</td>
</tr>
<tr>
<td>TOTAL SAR</td>
<td>81.1 GFLOPS</td>
<td>16.9 GB/s</td>
</tr>
</tbody>
</table>

10. Use in Practice

The language has been commercialized since January 2001 and has been used to develop many deployed applications, mostly in the aerospace and defense market. The largest software product developed using the language is a nose RADAR for a European jetfighter. This project utilizes over 20 modes organized using segmentation, using five levels of nested segmentation along with exclusion and shared state. Integrated into the segmented mode control is automated load balancing. Approximately 500,000 kernels are used in the application. [9][10][11]

Figure 12 - Stages of the distributed SAR algorithm.
The productivity achieved by using the language was measured by the defense company. Because the product is a reimplementation of code previously developed by the company by another development team, the productivity is measured by comparing implementation time using existing software process for the original implementation to using the stream processing language for the new implementation. These two products are referred to as Tranche 1 and Tranche 2 by the defense company.

Tranche 1 implemented the software using Real-Time Structured Analysis and Structured Design (RTSA/SD). The C language was used to program a custom processor that consisted of a number of cores implemented in ASICs, some creating SPARC processors. Because this bespoke processor can no longer be manufactured, requiring a software reimplementation, the defense company’s goal in Tranche 2 was to implement the software such that it is obsolescence proof.

Tranche 2 implemented the software using the stream processing language. The target hardware is a chassis with multiple custom boards, each with multiple G4 PowerPC processors. As the hardware was developed by the defense company, it was not available at the beginning of the project, and COTS (commercial off the shelf) DSP hardware was used for initial testing.

In 2003, the defense company measured the productivity improvement for Tranche 2 at between 4.1 and 4.8x. [12] resulting in a savings of hundreds of man-years. It is noted that this measurement was done before Integration and Acceptance Testing. In these two phases, the stream processing language provides even higher benefit due to the modularity of stream processing components and the tool’s ability to simplify optimization of the distribution through iterative changes to the compiler’s implementation settings.

Several other organizations have found success using the stream processing language for deployed real-time software. An American defense contractor used the language to program a laser RADAR (LADAR) seeker with automatic target recognition algorithms. [13] A military research lab used the language to create an audio processing toolbox with front-end graphical user interface. [14] Other defense contractors have used the language to create technology demonstrators for synthetic aperture radar (SAR) [15] and Electro-Optical (EO) image processing. [16]

11. Related Work
Several other stream processing languages are being developed to target DSPs, multicores, and Graphic Processing Units (GPU). These include Ptolemy, StreamIt, Brook, CUDA (Compute Unified Device Architecture), and others. While all these languages can be classified as stream processing, the incorporation of GPU languages under this term creates two different subclasses of languages with different properties.

Ptolemy is being developed by the University of California at Berkeley and supports DSPs and general purpose processors. The language allows for the specification of the branch and merge construct (“switch” and “select”) using dynamic data flow. [17] The Ptolemy project addresses the management of modularity through the integration of finite state machines with both static and dynamic data flow. [18] However, the language and compiler do not provide the automated resource management that segmentation provides. There is no formal recognition of the boundaries between stream segments or modes.

StreamIt is being developed by the Massachusetts Institute of Technology and supports multicores and general purpose processors. StreamIt provides a structured method of specifying dataflow by limiting the types of behavior that can be implemented. For example, kernels (“filters”) have at most one input and one output, and fan-in/fan-out is limited to static multiplexing constructs (“split/join”s). StreamIt 2.1 introduced dynamic rate, and their implementation allows for variable threshold. [19] However their restrictions on graph structure limit the applicability of these behaviors.

CUDA is being developed at Stanford University and provides a method for creating streams and calling stateless kernels from C. This method of stream programming is useful for targeting processors like GPUs that do many simple operations in parallel. To perform general purpose programming, the onus is on the programmer not the compiler to plan execution and manage resources. Brook supports dynamic behavior, including variable threshold, but only on kernel outputs; [20] variable threshold must be provided on input streams in order to implement the branch-and-merge construct.

CUDA is provided by the Nvidia Corporation for programming their GPUs. [21] CUDA is a stream processing language but similar to Brook, due to the nature of GPUs, execution is planned one kernel at a time. A C program is written surrounding the stream processing to manage resources and control the software.

12. Conclusions
Programming languages and compilers based on von Neumann architectures have dominated computer science for the past 50 years. When the realization of Moore’s Law created a long lineage of newer, faster single processor chips, the widespread use of these languages was not threatened. With the advent of multi-core processors as a means to continue to deliver faster, more powerful chips, sequential code is no longer suitable. Stream processing and data flow languages have existed nearly as long and provide a means of programming that is more closely related to these new architectures, however stream processing is often considered only applicable to signal processing defined by steady-state vector processing. By expanding the type system for specifying ports of kernels, we have diverted from data flow and stream processing concepts to produce a practical language that greatly expands the amount of problems that can be addressed with these types of languages. This type system simplifies the specification of software control and modality, placing the onus of global resource management and control structure onto the compiler, increasing productivity and maintainability while still creating efficient executables that exploit the advantages of the high performance architectures.

References


