Orchestrating the Execution of Stream Programs on Multicore Platforms

Manjunath Kudlur Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{kvman,mahlke}@umich.edu

Abstract

While multicore hardware has become ubiquitous, explicitly parallel programming models and compiler techniques for exploiting parallelism on these systems have noticeably lagged behind. Stream programming is one model that has wide applicability in the multimedia, graphics, and signal processing domains. Streaming models execute as a set of independent actors that explicitly communicate data through channels. This paper presents a compiler technique for planning and orchestrating the execution of streaming applications on multicore platforms. An integrated unfolding and partitioning step based on integer linear programming is presented that unfolds data parallel actors as needed and maximally packs actors onto cores. Next, the actors are assigned to pipeline stages in such a way that all communication is maximally overlapped with computation on the cores. To facilitate experimentation, a generalized code generation template for mapping the software pipeline onto the Cell architecture is presented. For a range of streaming applications, a geometric mean speedup of 14.7x is achieved on a 16-core Cell platform compared to a single core.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors Compilers

General Terms Languages, Algorithms, Performance

Keywords StreamIt, Cell processor, multicore, stream programming, software pipelining

1. Introduction

Multicore systems have become the industry standard from high-end servers, down through desktops and gaming platforms, and finally into handheld devices. Example systems include the Sun UltraSparc T1 that has 8 cores [14], the Sony/Toshiba/IBM Cell processor that consists of 9 cores [10], the NVIDIA GeForce 8800 GTX that contains 16 streaming multiprocessors each with eight processing units [19], and the Cisco CRS-1 Metro router that utilizes 192 Tensilica processors [5]. Intel and AMD are producing quad-core x86 systems today and larger systems are on their near term roadmaps. Putting more cores on a chip increases peak performance, but has shifted the burden onto both the programmer and compiler to identify large amounts of coarse-grain parallelism to effectively utilize the cores. Highly threaded server workloads naturally take advantage of more cores to increase throughput. However, the performance of single-thread applications has dramatically lagged behind. Traditional programming models, such as C, C++, and Fortran, are poorly matched to multicore environments because they assume a single instruction stream and a centralized memory structure.

The stream programming paradigm offers a promising approach for programming multicore systems. Stream languages are motivated by the application style used in image processing, graphics, networking, and other media processing domains. Example stream languages are StreamIt [26], Brook [3], CUDA [19], SPUR [28], Cg [18], and Baker [4]. Stream languages enable the explicit specification of producer-consumer parallelism between coarse grain units of computation. For this work, we focus on StreamIt where a program is represented as a set of autonomous actors (called filters in StreamIt) that communicate through first-in first-out (FIFO) data channels [26]. StreamIt implements the synchronous dataflow model [15] in which the number of data samples produced and consumed by each actor are specified a priori. During program execution, actors fire repeatedly in a periodic schedule [6]. Each actor has a separate instruction stream and an independent address space, thus all dependences between actors are made explicit through the communication channels. Compilers can leverage these characteristics to plan and orchestrate parallel execution.

Stream programs contain an abundance of explicit parallelism. The central challenge is obtaining an efficient mapping onto the target architecture. Often the gains obtained through parallel execution can be overshadowed by the costs of communication and synchronization. Resource limitations of the system must also be carefully modeled during the mapping process to avoid stalls. Resource limitations include finite processing capability and memory associated with each processing element, interconnect bandwidth, and direct memory access (DMA) latency. Lastly, stream programs contain multiple forms of parallelism that have different tradeoffs on when they should be exploited. It is critical that the compiler leverage a synergistic combination of parallelism, while avoiding both structural and resource hazards.

In this work, we propose a modulo scheduling algorithm for mapping streaming applications onto multicore systems, referred to as *stream graph modulo scheduling* or SGMS. Modulo scheduling is traditionally a form of software pipelining applied at the instruction level [22]. We apply the same technique on a coarse-grain stream graph to pipeline the actors across multiple cores. The objective is to maximize concurrent execution of actors while hiding communication overhead to minimize stalls. SGMS is a phase-ordered approach consisting of two steps. First, an integrated actor fission and partitioning step is performed to assign actors to each processor ensuring maximum work balance. Parallel data actors are selectively replicated and split to increase the opportunities for even work distribution. This first step is formulated as an integer linear program. The second step is stage assignment wherein each actor is assigned to a pipeline stage for execution. Stages are assigned to ensure data dependences are satisfied and inter-processor communication latency is maximally overlapped with computation.

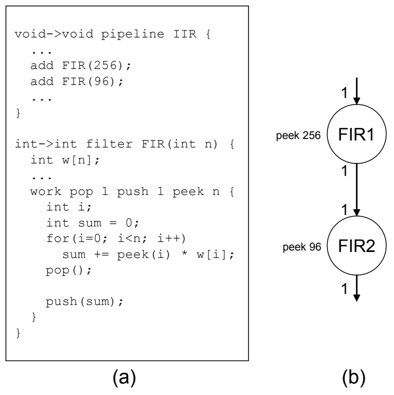
Our target platform is the Cell architecture, which represents the first tangible platform that is a decoupled multicore where there is no shared cache so code-data colocation is necessary [10]. SGMS is part of a fully automatic compilation system, known as StreamRoller, that maps StreamIt applications onto a Cell system. The SGMS schedule is output in the form of a C template that executes an arbitrary software pipeline. This template, combined with C versions of the actors, are compiled with the host compiler to execute on the target system. For our experiments, we use an IBM QS20 Blade Server running Fedora Core 6.0. It is a Cell system equipped with 16 3.2GHz synergistic processing engines (SPEs) on 2 chips, and 1 GB RAM.

Our work has the most overlap with the coarse-grained scheduling used in the StreamIt compiler [7, 6]. The StreamIt scheduler consists of two major phases. First, a set of transformations are applied on the stream graph to combine and split actors to ensure the computation granularity is balanced. Second, a coarse-grain software pipeline is constructed by iteratively applying a greedy partitioning heuristic that assigns filters to processors. Each filter is considered in order of decreasing work and assigned to the processor with the least amount of work so far. To minimize synchronization, the partitioning algorithm is wrapped with a selective fusion pass that repeatedly fuses the two adjacent filters that have the smallest combined work. This process reduces communication overhead by forcing the combined filters to reside on the same processor.

Our work differs along two primary dimensions. First, the StreamIt compiler targets the Raw processor that has a traditional cache on each processor [25]. In [6], intermediate buffers needed by the software pipeline of the stream graph are stored off to the off-chip DRAM banks, and a separate communication stage is introduced between steady states to shuffle data between banks. Our formulation of pipeline stage assignment explicitly models DMA overhead and proactively overlaps data transfers for future iterations with computation on the current iteration. Second, we formulate the partitioning and actor fission step as an integer linear program rather than employing iterative partitioning and fusing to generate a schedule. Our approach combines packing and fission of actors, data transfers, and resource constraints to generate more balanced and higher quality schedules for architectures such as Cell.

This paper offers the following contributions:

- The design, implementation, and evaluation of stream graph modulo scheduling for efficiently mapping streaming applications onto decoupled multicore systems.
- An integer linear program formulation for integrated actor fission and partitioning to assign actors to processing elements maximizing workload balance.
- A pipeline stage assignment algorithm that proactively overlaps DMA transfers with computation to minimize stalls.
- A fully automated compilation system for Cell capable of generating performance results on real hardware.



```
void->void pipeline IIR {
  ...
  add FIR(256);
  add FIR(96);
  ...
}

int->int filter FIR(int n) {
  int w[n];
  ...
  work pop 1 push 1 peek n {
    int i;
    int sum = 0;
    for(i=0; i<n; i++)
      sum += peek(i) * w[i];
    pop();

    push(sum);
  }
}
```

Figure 1: (a) Example StreamIt Program and (b) corresponding stream graph.

2. Background and Motivation

2.1 StreamIt Language

StreamIt [26] is an explicitly parallel programming language that implements the synchronous data flow (SDF) [15] programming model. Actors are specified by parametrized classes, which are similar to Java classes. They can have local variables corresponding to local actor state, and methods that accesses these variables. An actor can have both read-only and read-write state. A stateful actor that modifies local state during the work function cannot be parallelized as the next invocation depends on the previous invocation. However, the SDF semantics allow the parallel replication of stateless actors. A special method called `work` is reserved to specify the work function that is executed when the actor is invoked in steady state. The stream rates (number of items *pushed* and *popped* on every invocation) of the work functions are specified statically in the program.

The stream graph is constructed by instantiating objects of the actor classes. StreamIt provides ways to construct specific structures like *pipeline*, *split-join*, and *feedback loop*. Using these primitives, the entire graph can be constructed hierarchically. Note that feedback loops provide a means to create cycles in the stream graph. Feedback loops are naïvely handled by fusing the entire loop into a single actor. More intelligent ways to handle nested loops is beyond the scope of this paper. Further, the feedback loop pattern does not appear in any of the benchmarks that we evaluate. Hence, the rest of the paper assumes an acyclic stream graph.

Figure 1 shows an example StreamIt program and its corresponding stream graph. StreamIt provides the `peek` primitive to the programmer, which can be used to non-destructively read values off the input channel. Note that this is only for convenience, and does not make StreamIt deviate from the pure SDF model. This is because a program with `peek` can always be reimplemented with just `pushes` and `pops`, and some local state that holds a subset of values seen so far.

2.2 Cell Broadband Architecture

Our compilation target in this paper is the Cell Broadband Engine (CBE) shown in Figure 2. The CBE is a heterogeneous multicore system, consisting of one 64bit PowerPC core called the Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). Each SPE has a SIMD engine called the synergistic processing unit (SPU), 256 KB of local memory and a memory flow control (MFC) unit which can perform DMA operations to and from the local stores independent of the SPUs. The SPUs can only access the local store, so any sharing of data has to be per-

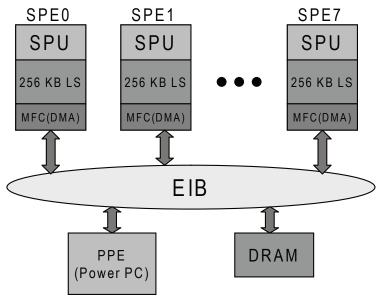


Figure 2: The Cell broadband architecture.

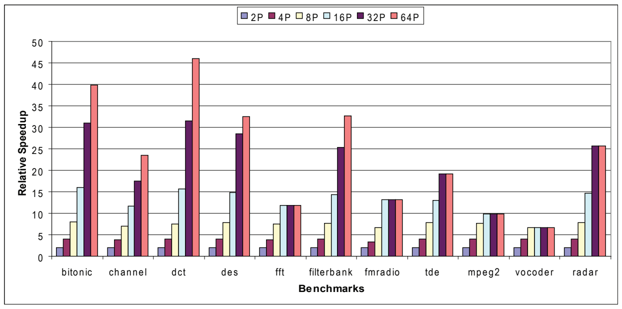


Figure 3: Theoretical speedup for unmodified programmer-conceived stream graph.

formed through explicit DMA operations. The SPEs and PPE are connected via a high bandwidth interconnect called the Element Interconnect Bus (EIB). The main memory and peripheral devices are also connected to the EIB. The feature of the CBE most relevant to this paper is the ability of the MFCs to do non-blocking DMA operations independent of the SPUs. The SPUs can issue DMA requests that are added to hardware queues of the MFCs. The SPU can continue doing computation while the DMA operation is in progress. The SPU can query the MFC for DMA completion status and block only when the needed data has not yet arrived. The ability to perform asynchronous DMA operations allow overlap of computation and communication, and is leveraged for efficient software pipelining of stream graphs.

2.3 Motivation

Stream programs are replete with pipeline parallelism. An actor can start working on the next data item as soon as it is done with the current item, even when other actors in the downward stream of the graph are still working on the current item. In a multiprocessor environment, by running different actors on different processors and overlapping iterations, the outer loop can be greatly sped up. Trying to exploit pipeline parallelism requires (1) a good distribution of work among the available processors and (2) managing the communication overhead resulting because of producers and consumers running on different processors.

The partitioning problem. Figure 3 shows the theoretical speedup possible for a set of unmodified stream programs for 2 to 64 processors.[1] The actors present in the programmer-conceived stream graph are assigned to processors in an optimal fashion such that the maximal load on any processor is minimized. Speedup is calculated by dividing the single processor runtime by the load on the maximally loaded processor. The programmer-conceived stream graph has ample parallelism that can be exploited on up to 8 processors. Beyond 8 processors, the speedup begins to level off. Most benchmarks just do not have enough actors to span all processors. For example, `fft` has only 17 filters in its stream graph, therefore no speedup is possible beyond 17 processors. The other reason is that work is not evenly distributed across the actors. Even though the computation has been split into multiple actors, the programmer has no accurate idea of how long an actor's work function will take to execute on a processor when coding the function. This combined with the fact that work functions are indivisible units leads to less scaling on 16 or more processors. For example, in the `vocoder` benchmark, the longest running actor contributes to 12% of the work, thus limiting the theoretical speedup to $\frac{100}{12} = 8.3$.

Most of the benchmarks are completely stateless, i.e., all actors are data parallel [6]. In fact, only `mpeg2`, `vocoder`, and `radar` have actors that are stateful. Data parallel actors can be replicated (or fissed) any number of times without changing the meaning of the program. The longest running actor in `vocoder` benchmark is stateless, and can be fissed to reduce the amount of work done in a single actor. Fissing data parallel actors not only allows work to span more processors, it also allows work to be evenly distributed across processors by making the largest indivisible unit of work smaller.

Even though data parallel actors provide ample opportunity to divide up work evenly across processors, it is not obvious how many times an actor has to be fissed to achieve load balance. An actual partitioning has to be performed to decide if actors have been fissed enough number of times. On the other hand, a good partitioning is achieved only when actors have been fissed into suitably small units. This circular cause and consequence warrants an integrated solution that considers the problems of fission and partitioning in a holistic manner.

Communication overhead. When an actor that produces data and the actor(s) that consume that data are mapped to different processors, the data must be communicated to the consumers. In our implementation on the Cell system, actors are mapped to the SPEs that have disjoint address spaces. Therefore, communicating data to consumers is through an explicit DMA. When such transfers are not avoided, or not carefully overlapped with useful work, the overhead could dominate the execution times.

The next section addresses the problem of partitioning and communication overhead. First, an integrated fission and partitioning method is presented that fisses the actors just enough to span all processors, and also obtain an even work distribution. Next, the stage assignment step divides up the actors into pipeline stages in which all communication is overlapped with computations.

3. Stream Graph Modulo Scheduling

This section describes our method for scheduling a stream graph onto a multicore system. The objective is to obtain a maximal throughput software pipeline taking both the computation and communication overhead into account. The stream graph modulo scheduling (SGMS) algorithm is divided into two phases. The first phase is an integrated fission and processor assignment step based on an integer linear program formulation. It fisses data parallel actors as necessary to get maximal load balance across the given number of processors. The second phase assigns actors to pipeline stages in such a manner that all communication is overlapped with computation on the processors.

3.1 Integrated Fission and Processor Assignment

Consider a dataflow graph $G = (V, E)$ corresponding to a stream program. Let $|V| = N$ be the number of actors. Let the basic repetition vector be r, where r_i specifies the number of times v_i is executed in a static schedule. Let $t(v_i)$ be the time taken to execute

[1] More details of the applications are provided in Section 4.

r_i copies of v_i. The rest of the section assumes r_i executions of v_i as the basic schedulable unit. Given P processors, a software pipeline needs some assignment of the actors to the processors. The throughput of the software pipeline is determined by the load on the maximally loaded processor. As shown in Section 2, even an optimal assignment on the unmodified programmer conceived stream graph does not provide linear speedups beyond 8 processors. Some data parallel actors need to be fissed into two or more copies so that there is more freedom in distributing work evenly across the processors. For each actor in the stream graph, the following ILP formulation comes up with the number of times the actor has to be fissed, and an assignment of each copy of the actor to a processor. The objective function is the maximal load on any processor, which is minimized.

A set of 0-1 integer variables $a_{i,j,k,l}$ is introduced for every actor v_i. The meaning of the four suffixes is explained below:

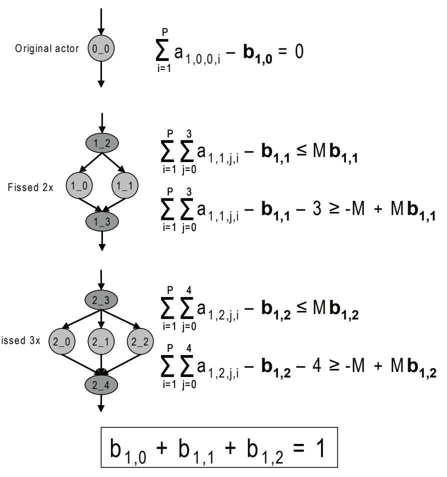
- i identifies the actor.
- j identifies the *version* of the actor that would appear in the final graph. For every actor v_i, the formulation considers multiple versions of the actor. Version 0 of the actor is fissed 0 times (no copies made), version 1 of the actor is fissed once so that two copies of the actor are considered for scheduling, and so on.
- k identifies the copy of the jth version of the actor v_i. Version 0 has only one copy. Version 1 has 2 copies of the actor and a splitter and joiner. The splitter and joiner have to run on some processor, therefore, they are considered as independent schedulable units. Thus there are $(j + 3)$ schedulable actors in the jth version. We have either $0 \le k < j + 3$ when $j \ge 1$, or $k = 0$ when $j = 0$.
- l identifies the processor to which the kth copy is assigned.

Let Q be the maximum number of versions considered for an actor. Actors with carried state cannot be fissed at all and $Q = 1$ for such actors. On the other hand, stateless actors can be fissed any number of times. The choice of Q affects the load balance obtained from the processor assignment. Choosing a low value for Q would inhibit the freedom of distributing copies of an actor to many processors. We observed that the maximum number of copies of an actor that appear in the best partitions is always less than P for all benchmarks. Therefore, in the experiments Q was set to P, the number of processors under consideration. The following equation ensures that a copy of an actor is either assigned to one processor or not assigned to any processor at all, implying that a different version was chosen.

$$\sum_{l=1}^{P} a_{i,j,k,l} \le 1 \qquad \forall i, 0 \le j < Q, 0 \le k < j + 3 \tag{1}$$

When a copy of an actor is indeed assigned to a processor, all other copies in the same version have to be assigned to processors, and all other versions should not be assigned to processors. To ensure this, a set of Q indicator variables, $b_{i,q}, 0 \le q < Q$, are introduced for every actor v_i. These indicator variables are 0-1 variables which serve two purposes. First, they indicate which version of the actor was chosen. Second, by virtue of being either 0 or 1 only, ensure that either all copies of a version are assigned to processors, or no copy is assigned to any processor. The following set of equations show the relation between the indicator variables $b_{i,q}$ and the assignment variables $a_{i,j,k,l}$.

$$\sum_{l=1}^{P} a_{i,0,0,l} - b_{i,0} = 0 \qquad \forall i \tag{2}$$



Original actor
0_0
$\sum_{i=1}^{P} a_{1,0,0,i} - \mathbf{b_{1,0}} = 0$
1_2
Fissed 2x
1_0
1_1
1_3
$\sum_{i=1}^{P}\sum_{j=0}^{3} a_{1,1,j,i} - \mathbf{b_{1,1}} \le M\mathbf{b_{1,1}}$
$\sum_{i=1}^{P}\sum_{j=0}^{3} a_{1,1,j,i} - \mathbf{b_{1,1}} - 3 \ge -M + M\mathbf{b_{1,1}}$
2_3
Fissed 3x
2_0
2_1
2_2
2_4
$\sum_{i=1}^{P}\sum_{j=0}^{4} a_{1,2,j,i} - \mathbf{b_{1,2}} \le M\mathbf{b_{1,2}}$
$\sum_{i=1}^{P}\sum_{j=0}^{4} a_{1,2,j,i} - \mathbf{b_{1,2}} - 4 \ge -M + M\mathbf{b_{1,2}}$
$b_{1,0} + b_{1,1} + b_{1,2} = 1$

Figure 4: Example illustrating ILP formulation.

$$\sum_{l=1}^{P}\sum_{k=0}^{j+2} a_{i,j,k,l} - b_{i,j} \le M \times b_{i,j} \qquad \forall i, 1 \le j < Q \tag{3}$$

$$\sum_{l=1}^{P}\sum_{k=0}^{j+2} a_{i,j,k,l} - (j+2) - b_{i,j} \ge -M + M \times b_{i,j} \qquad \forall i, 1 \le j < Q \tag{4}$$

M in Equations 3 and 4 is a constant that is larger than the upper bound of $\sum_{l=1}^{P}\sum_{k=0}^{j+2} a_{i,j,k,l}$. Note that Equations 3 and 4 are standard ILP tricks to ensure that a linear sum either equals a constant or is zero. In this case, the sum $\sum_{l=1}^{P}\sum_{k=0}^{j+2} a_{i,j,k,l}$ either has to be $(j + 3)$, denoting that all copies of a version were assigned to some processor, or has to be 0, denoting that none of the copies were assigned to any processor. $b_{i,j}$ conveniently takes on 1 or 0, respectively. The following equation ensures that one and only one version of an actor is chosen in the final assignment.

$$\sum_{j=0}^{Q} b_{i,j} = 1 \qquad \forall i \tag{5}$$

Figure 4 illustrates the above set of equations for an example actor. Q is chosen to be 3 in the example. Three versions of the actor are shown in the figure. The labels on the nodes indicate the version number and copy number. The last equation $b_{1,0} + b_{1,1} + b_{1,2} = 1$ ensures that only one version is chosen, and the rest of the equations ensure that all copies of the chosen version are assigned to processors.

To determine the quality of an assignment, the amount of work assigned to each processor has to be calculated. The following equation computes the work (in terms of time) done by a copy of an actor.

$$W_{i,j,k,l} = \begin{cases} t(v_i) & \text{if j = 0} \\ \frac{t(v_i)}{j+1} + \epsilon & \text{if j > 1 and k < j+1} \\ splitter\_work(v_i) & \text{if j > 1 and k = j+1} \\ joiner\_work(v_i) & \text{if j > 1 and k = j+2} \end{cases} \tag{6}$$

Version 0 of the actor is same as the original actor. Therefore, the work done by version 0 is the original work $t(v_i)$. In version 1, there are 2 copies of the actor that do half the work as the original actor. Note that there is a small overhead of ϵ when fissing actors which peek more elements than they pop. This is due to the introduction of a decimation stage on each copy which just pops and ignores part of the data to maintain correct semantics. In addition, there is additional work done by the splitter and joiner in version 1. The last three cases in Equation 6 compute the work done by copies of the actor, splitter, and joiner. Note that the work done in splitter and joiner depends on the implementation. However, they both are constants given the number of items popped by the corresponding actor. For some assignment of actors to processors, the following equation computes the total work TW_p that gets assigned to a processor p.

$$TW_p = \sum_{i=1}^{N} \sum_{j=0}^{Q} \sum_{valid\ k} a_{i,j,k,l} \times W_{i,j,k,p} \qquad (7)$$

The processor p with maximum work TW_p assigned to it constitutes the bottleneck processor, and thus TW_p denotes the inverse of the throughput of the overall pipeline. We borrow the terminology from operation-centric modulo scheduling used in compiler backends, and use the term Initiation Interval (II) to denote the inverse of the throughput. The following set of equations compute II from the TW_p's.

$$TW_p \le II \qquad 1 \le p \le P \qquad (8)$$

The ILP program that minimizes II subject to constraints given by Equations 1 to 8 provides the following information.

- The value of j for which $b_{i,j} = 1$ identifies the version of the actor chosen. Note that Equation 5 ensures that only one of the $b_{i,j}$'s have the value 1.
- Given a copy k of the chosen version j, the set of values $a_{i,j,k,l}$ that are 1 identify the processors to which the copy is assigned. For example, if $a_{i,j,k,4} = 1$, then the kth copy the actor is assigned to processor 4.

The above formulation does not account for any communication overhead. The data produced by an actor has to be communicated to a consuming actor if that actor was assigned to a different processor. The following section shows how all such communication can be hidden, thus achieving the exact throughput obtained from the processor assignment step.

3.2 Stage Assignment

The processor assignment obtained by the method described in the previous section provides only partial information for a pipeline schedule. Namely, it specifies how actor executions are overlapped across processors. It does not specify how they are overlapped in time. To realize the throughput, which is the load on the maximally loaded processor obtained from processor assignment, all actors assigned to a processor including the necessary DMAs have to be completed within a window of II time units. The only goal of processor assignment step is load balance, therefore it assigns actors to different processors without taking any data precedence constraints into consideration. An actor assigned to a processor could have its producer assigned to a different processor, and have its consumer assigned to yet another processor. To honor data dependence constraints and still realize the throughput obtained from processor assignment, the actor executions corresponding to a single iteration of the entire stream graph are grouped into *stages*. Note that the concept of stage is adapted from traditional VLIW modulo scheduling. Across all processors, stages of a single iteration execute sequentially, thus honoring data dependences. Within a single processor, no stages are *active* at the beginning of execution. During the initial few iterations, stages are activated sequentially, thus filling up the pipeline and enabling executions of data dependent actors belonging to earlier iterations concurrently with actors from later iterations. In steady state, all stages are active on a processor, thus realizing the throughput obtained from processor assignment. The

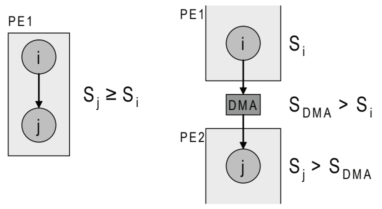


Figure 5: Properties of stages.

pipeline is drained by deactivating stages during the final few iterations.

The overarching goal of the stage assignment step is to overlap all data communication (DMAs) between actors. To achieve this, the stage assignment step considers the DMAs as schedulable units. To honor data dependences and ensure DMAs can be overlapped with actor executions, certain properties are enforced on the stage numbers of actors. Consider a stream graph $G = (V, E)$. The stage to which an actor v_i is assigned to is denoted by S_i. In addition, the processor to which v_i is assigned to is denoted by p_i. The following rules enforce data dependences and ensure DMA overlap.

- $(v_i, v_j) \in E \Rightarrow S_j \ge S_i$, i.e., the stage number of a consuming actor should come after the producing actor. This is to preserve data dependence.
- If $(v_i, v_j) \in E$ and $p_i \ne p_j$, then a DMA operation must be performed to get the data from p_i to p_j. The DMA operation is given a separate stage number S_{DMA}. As shown in Figure 5, the inequality $S_i < S_{DMA} < S_j$ is enforced between the stages of the different actors and the DMA operation. The DMA operation is separated from the producer by at least one stage, and similarly, the consumer is separated from the DMA operation by one stage. This ensures decoupling, and allows the overlap of the producer and the DMA, as well as the DMA and the consumer.
- Within the set of actors assigned to some processor p, the inequality $\sum_{S_j = s} t(v_j) \le II, \forall s$ is enforced. In other words, the sum of execution times of actors (S_j) assigned to a stage (s) should be less than the desired II. This is the basic modulo scheduling constraint, which ensures that the stages are not overloaded, and that a new iteration can be initiated every II time units.

A simple data flow traversal of the stream graph is used to assign stages to actors as shown in Algorithm 1. For each actor in dataflow order, the **FindStage** procedure assigns a stage to the actor. The `for` loop beginning on the line marked 1 computes the maximum stage of the producers of the actor under consideration. If any of the producers are assigned to a different processor, the earliest stage considered for actor is *maxstage* + 2, which leaves room for DMAs in *maxstage* + 1. Otherwise, the actor could be placed on *maxstage*. The `while` loop beginning on the line marked 4 finds a stage number later than *stage* on which the load is less than the II obtained from processor assignment.

3.3 Code Generation for Cell

This section describes a code generation strategy to implement the modulo schedule obtained for a stream program on a Cell system. The target of our code generation are the multiple SPEs, as opposed to the PPE. This section describes the general code generation schema, the buffer allocation strategy, and provides a complete example.

Code generation schema. The SPEs are independent processors with disjoint address spaces. The general code generation strat-

```
    FindStage (actor) :
    $maxstage \leftarrow 0$ ;
    $flag \leftarrow false$ ;
1   foreach producer p of actor do
        if $stage(p) > maxstage$ then
            $maxstage \leftarrow stage(p)$ ;
        end
2       if $Proc(p) \neq Proc(actor)$ then
            $flag \leftarrow true$ ;
        end
    end
3   if flag then
        $stage \leftarrow maxstage + 2$ ;
    else
        $stage \leftarrow maxstage$ ;
    end
4   while $Load(Proc(p), stage) + t(actor) > II$ do
        $stage \leftarrow stage + 1$
    end
    $Load(Proc(p), stage) += t(actor)$ ;
    return stage
```

Algorithm 1: Stage assignment procedure

egy is to spawn one thread per SPE. Each thread makes calls to work functions corresponding to actors that are assigned to the respective SPEs, and perform DMAs to get data from other SPEs. The main program, running on the PPE, just spawns the SPE threads and does not intervene thereafter.

Figure 6 shows pseudo C code that runs on each SPE thread. It mimics the kernel-only [23] code of modulo scheduling for a VLIW processor. The array `stage` functions similar to the *staging predicate*, and its size (`N`) is the maximum number of stages. The main loop starts off with only the first stage active. The `if` conditions that test different elements of `stage` ensure only actors assigned to a particular stage are executed. The last part of the loop *shifts* the elements of the array `stage` to the left, which has the effect of filling up the software pipeline. Finally, when all iterations are done, draining the software pipeline is accomplished by shifting a `0` into the last element of `stage`.

```
void spe_work()
{
  char stage[N] = {0};

  stage[0] = 1;

  for (i=0; i<max_iter+N-1; i++) {
    if (stage[N-1]) {
      // Begin DMA operations
      // Call to filter work functions
    }
    if (stage[N-2]) {
    }
    ...
    if (stage[0]) {
    }
    wait_for_dma_completion();
    // start epilogue
    if (i == max_iter-1)
      stage[0] = 0;
    // Shift-left staging predicate
    for(j=N-1; j>=1; j--)
      stage[j] = stage[j-1];
    barrier();
}
```

Figure 6: Main loop implementing the modulo schedule.

The code corresponding to each active stage are calls to the work functions of the actors assigned to this SPE and the corresponding stage, and the necessary DMAs to fetch data from other SPEs. The Cell processor provides non-blocking DMA functionality [11], which is leveraged for overlapping DMAs and computation. A DMA operation assigned to a particular stage is implemented using the `mfc_get` primitive, which enters the DMA command into a queue and returns immediately. The MFC engine in each SPE processes the queue asynchronously and independent of the processor. After enqueuing the DMA request, the code proceeds to execute work functions for actors. Note that even though the actual DMA operations are asynchronous, the SPE should queue up the DMA requests synchronously using the `mfc_get` primitive. No more DMAs can be queued once work functions begin execution. Therefore, all DMA operations belonging to a stage are queued up before any work functions are called to ensure maximal overlap of actual DMAs and computation. Finally, the `wait_for_dma_completion` uses the `mfc_read_tag_status_all` primitive to ensure all DMAs issued in the current iteration are completed, and a barrier synchronization is executed to ensure the current iteration is completed on all SPEs. `barrier()` is implemented using the `signal` mechanism available on the SPEs, and with the current implementation, 2×10^6 barriers can be performed in 1 second.

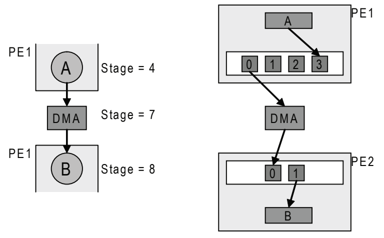


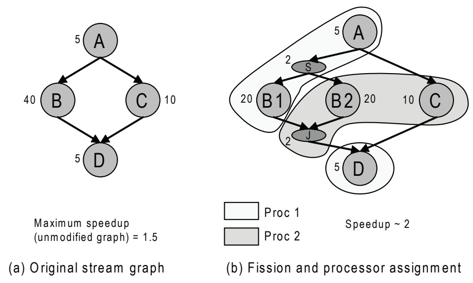
Figure 7: Buffer allocation for the modulo schedule.

Buffer allocation. In the code generation schema described above, several iterations of the original stream graph are in flight concurrently. A producer actor could be executed multiple times before one of its consumers is ever executed. To ensure correct operation, multiple buffers are used to store the outputs of producer actors. The buffers are used in a fashion similar to rotating registers in a traditional modulo schedule. The number of buffers needed for the output of a producer actor assigned to stage S_p feeding a consumer actor on stage S_c can easily be calculated as $S_c - S_p + 1$.

Figure 7 shows the buffer allocation for a producer actor *A* and consumer actor *B*. They are assigned to different processors with an intervening DMA. Since the stage separation between *A* and the DMA is 3, 4 buffers are allocated on the local memory of PE1, and *A* uses them in a round-robin fashion. The arrows on the picture on the right shows the current buffers being used. Note that the DMA operation and actor *A* are executing concurrently by using different buffers. Similarly, *B* is using a buffer different from the DMA. In the current implementation, all buffers are allocated on the local memories of the SPEs. The buffers between a producer actor and a DMA operation are stored on the SPE on which the producer is running. Symmetrically, the buffers between the DMA operation and the consuming actor are stored on the consumer SPE. 256KB of local store is sufficient to hold all the buffers needed by the benchmarks evaluated. This is corroborated by the authors of [6], who report that the buffers needed by the benchmarks would fit on the 512KB cache of the Cell processor.

3.3.1 Example

Figure 8(a) shows an example stream graph. Assume that all actors in the graph are data parallel, i.e., they can be fissed any number of times. The numbers beside the nodes represent the amount of work done by the actors. Note that *B* does the most work of 40 units and the sum of work done by all actors is 60 units. When trying to schedule the unmodified graph on to 2 processors, the maximum



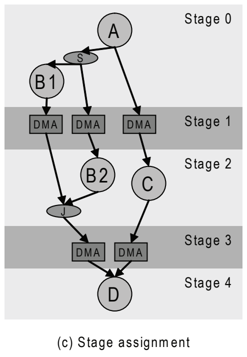


Figure 8: Example illustrating fission, processor assignment and stage assignment.

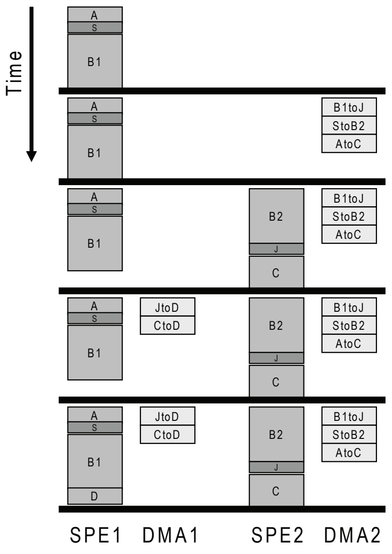


Figure 9: Example illustrating a modulo schedule running on Cell.

achievable speedup is $\frac{60}{40} = 1.5$. Figure 8(b) shows the result of the integrated fission on processor assignment step. Node B has been fissed once, resulting in two new nodes $B1$ and $B2$, and the corresponding splitter S and joiner J, whose work are assumed to be 2 units. The processor assignment obtained has an II of 32, thus resulting in a speedup of $\frac{60}{32} \sim 2$. Finally, Figure 8(c) shows the stage assignment in which DMAs are separated from consumers by one stage, thus ensuring complete overlap of computation and communication.

Figure 9 shows the execution timeline of the code running on two SPEs. The main feature to note is the steady state execution, which starts from the 5th iteration in Figure 9. In the steady state,

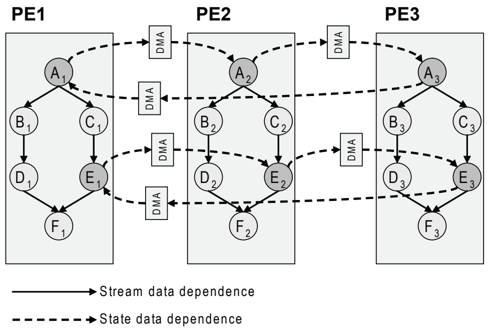


Figure 10: Mapping of an unfolded stream graph on to 3 processors.

all actors and all DMAs are active. The 4 iterations shown before the steady state correspond to the prologue of the modulo schedule, in which some actor executions and DMAs do not happen as they are predicated by the `stage` array. The DMA operations are started before actor executions on the SPEs, thus ensuring overlap with computation. Due to the overlap, the purported speedup of 2 is achieved by the schedule.

4. Evaluation

This section presents our evaluation of SGMS. First, a simple alternative scheme which naïvely unfolds the entire stream graph is presented. Then, various aspects of SGMS are evaluated, including a comparison to naïve unfolding.

4.1 Naïve Unfolding

This technique is based on a simple observation: when all actors in a stream program are stateless, the graph can be unfolded P times (where P is the number of available processors) and each copy of the graph can be run on one of the processors without incurring any communication overhead, and thus achieving a speedup of P. Unfolding [20] refers to the process of making multiple copies of the stream program and is analogous to unrolling a loop in traditional compilation. Unfolding is different from fission presented earlier in the paper. Fissing an actor introduces additional split and join nodes, and stream program semantics does not allow fissing a stateful actor. Unfolding the entire stream graph, including stateful actors, is possible if the additional dependences introduced due to carried state are honored. Also, when the entire graph is unfolded, stateless actors that peek more elements than they pop should also be considered stateful. This is because the extra elements that are peeked have to be "remembered" until the next invocation.

Figure 10 shows a stream graph unfolded 3 times and mapped on to 3 processors. The copies of nodes A and E shown as darker circles are stateful actors in the original graph. In the unfolded version, new edges $A_1 \rightarrow A_2, A_2 \rightarrow A_3$, and $A_3 \rightarrow A_1$ enforce dependencies due to persistent state in actor A. These edges, referred to as state data dependence edges, are different from the edges which denote flow of stream data. State data dependence edges enforce the fact that the second copy of the unfolded actor can execute only after the first copy has finished its execution and passed on the values of state variables.

Unfolding the stream graph and mapping it to processors as shown in Figure 10 introduces recurrence cycles which is one of the limiting factors of performance of such a mapping. Consider a stream graph $G = (V, E)$. Let the stateful nodes in V be denoted by the set $V_s \subseteq V$. Suppose $t(v)$ be the execution time of actor $v \in V$ and $s(v)$, the amount of time taken to transfer the state data associated with $v \in V_s$. $s(v)$ depends on the size of the persistent state of actor v and the communication latency. We assume that the size of persistent state is constant and does not grow during runtime. StreamIt does not allow dynamic memory allocation, and thus this property holds for all benchmarks in our evaluation. As evident from Figure 10, for every stateful node, a recurrence cycle of length $n \times s(v) + n \times t(v)$ is introduced in the unfolded version, where n is the unfold factor. The longest cycle in the graph constrains the maximum throughput achievable for the graph. We adopt the terminology used in traditional instruction centric software pipelining, and refer to the critical path length as "recurrence constrained minimum initiation interval", or RecMII. Thus, the RecMII in the unfolded graph is

$$RecMII = \max_{v \in V_s} (n \times s(v) + n \times t(v)) \tag{9}$$

The maximum achievable throughput is also limited by the resources, in this case the limited number of processors available to execute the graph. The constraint on throughput due to resources is referred to as "resource constrained minimum initiation interval", or ResMII. In the mapping shown in Figure 10, each processor executes all the actors in the original stream graph. In addition, for every stateful actor, the processor performs a DMA to move the state data. Thus, every processor is equally loaded, and the load is

$$ResMII = \sum_{v \in V_s} s(v) + \sum_{v \in V} t(v) \tag{10}$$

The best throughput for the graph using the above mapping described, referred to as the "minimum initiation interval", or MII, is simple the maximum of RecMII and ResMII. Suppose the number of actors in the stream program are much larger than the number of available processors, i.e., $|V| \gg P$. Then, RecMII would be much smaller than ResMII because ResMII is the sum of work on all actors, whereas RecMII depends on the work of one actor. As long as the stream program does not have a large stateful filter that dominates the run time, which is true of our benchmark set, we have ResMII > RecMII. Given that MII = ResMII, in steady state, the above mapping on n processors completes n iterations in MII cycles. Thus the speedup achieved by this mapping over one processor is given by

$$Speedup = \frac{n \times \sum_{v \in V} t(v)}{\sum_{v \in V_s} s(v) + \sum_{v \in V} t(v)} \tag{11}$$

The code to run the naïve unfolding schedule on the Cell processor consists of one thread per SPE. SPEs are ordered to keep track of which iterations are executed on which SPE. Each SPE executes all actors in the stream graph in data flow order. Before executing a stateful actor, an SPE synchronizes with the "previous" SPE, and gets the values of state variables. The SPE then synchronizes with the "next" SPE and passes on the values of state variables. This is done repetitively, so that an SPE executes iterations $i, i + n, i + 2n...$, where n is the total number of SPEs.

The main differences between naïve unfolding and SGMS can be summarized as below.

- All DMA transfers of stream data can be overlapped with computation in SGMS where as DMA transfers of state data cannot be overlapped with any computation as it is present in the critical path.
- In the naïve unfolding method, each SPE runs all actors in the original stream graph, whereas in SGMS, an SPE runs only a subset of the actors. Therefore, the memory footprint of code for naïve unfolding is much larger than for SGMS.

Benchmark	Actors	Stateful	Peeking	State size (bytes)
bitonic	28	2	0	4
channel	54	2	34	252
dct	36	2	0	4
des	33	2	0	4
fft	17	2	0	4
filterbank	68	2	32	508
fmradio	29	2	14	508
tde	28	2	0	4
mpeg2	26	3	0	4
vocoder	96	11	17	112
radar	54	44	0	1032

Table 1: Benchmark characteristics.

- The latency for one iteration of the original stream graph is equal to the uni-processor execution time of an iteration in the naïve unfolding method. This is because all actors belonging to one iteration is executed sequentially by an SPE. In contrast, task level parallelism is exploited within an iteration in SGMS, and therefore, the latency for an iteration could be much smaller.

Despite the shortcomings compared to SGMS, naïve unfolding is a simple method which requires no sophisticated compiler analyses, and is straightforward to implement for the Cell processor. We compare SGMS with naïve unfolding in the following section.

4.2 Experiments

This section presents the results of the experimental evaluation of SGMS, and comparison to the naïve unfolding method. A uniprocessor schedule was first generated for one SPE, with instrumentations added for measuring running time of each actor. The SPU "decrementer", a low overhead timing measurement mechanism, is used for profiling. The timing profile for each actor is used by the SGMS scheduler that generates schedules for 2-16 processors. The scheduler uses the CPLEX mixed integer program solver during the integrated fission and processor assignment phase. The code generation phase outputs plain C code that is divided into code that runs on the Power processor and code that runs on individual SPEs. The main thread running on Power processor spawns one thread per SPE. Each SPE thread executes a code pattern that was described in Section 3.3. IBM's Cell SDK 2.1 was used to implement the DMA copies, and the barrier synchronization. The GNU C compiler gcc 4.1.1 targeting the SPE was used to compile the programs. Note that only vectorization that was automatically discovered by gcc were performed on the actors' codes. The hardware used for our evaluation is an IBM QS20 Blade server. It is equipped with 2 Cell BE processors and 1 GB XDRAM.

Benchmark suite. The set of benchmarks available with StreamIt software version 2.1.1 was used to evaluate the scheduling methods. Most benchmarks are from the signal processing domain. `bitonic` implements the parallel bitonic sorting algorithm. `des` is a pipelined version of DES encryption cipher. [6] provides descriptions of the benchmarks. Table 1 shows the details relevant to our evaluation. Number of stateful actors with explicit state and peeking actors with implicit state are important to understand the speedups from naïve unfolding. Typical sizes of states in these benchmarks are also shown.

SGMS performance. Figure 11 shows the speedups obtained by SGMS over single processor execution on 2 to 16 processors for the benchmark suite. SGMS obtains near linear speedup for all benchmarks, resulting in the geometric mean speedup of 14.7x on 16 processors. The main reasons for near linear speedups are listed below.

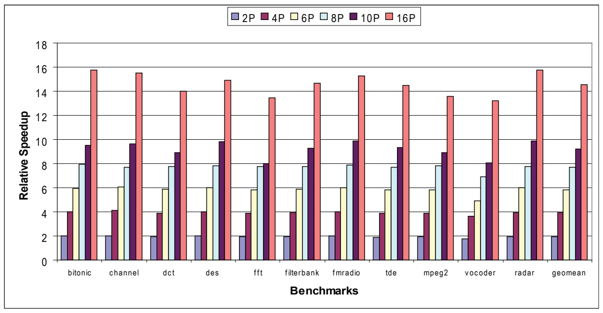


Figure 11: Stream graph modulo scheduling speedup normalized to single SPE.

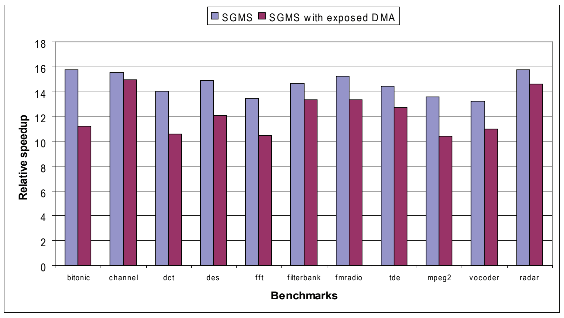


Figure 13: Effect of exposed DMA latency.

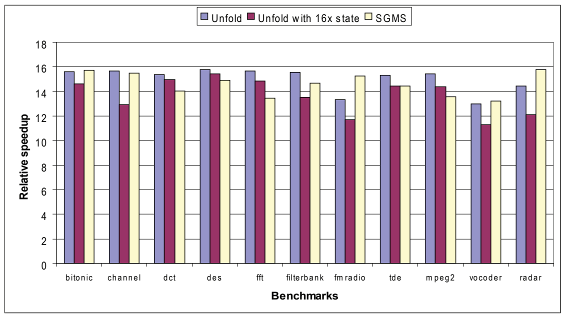


Figure 12: Comparing naïve unfolding to SGMS.

- The integrated fission and partitioning step fisses enough data parallel actors and the resulting number of actors is enough to span all available processors.
- The partitioning assigns actors to processors with maximal load balance.
- Stage assignment separates data transfers and actors that use the data into different stages. This ensures that all data transfers are overlapped with computation.

Note that with perfect load balance and complete overlap of all communication with computation should always result in a speedup of N on N processors. However, the observed geometric mean speedup is only 14.7x on 16 processors. One of the main overheads in our implementation arises from the barrier synchronization. As shown in Figure 6, all SPEs do a barrier synchronization at the end of every iteration of the loop implementing the modulo schedule. Our implementation of the `barrier` on the SPEs adds an overhead of 1 second for every 2×10^6 calls. Depending on the number iterations the stream graph is executed, barrier synchronization adds an overhead of up to 3 seconds in some benchmarks. A notable benchmark is `vocoder` for which the 16 processor speedup is only 13x. `vocoder` has 96 actors in the stream graph. On 16 processors, the partitioning results in over 30 DMA operations being in flight at the same time, which adds some overhead to the steady state. SGMS relies on static work estimates during the partitioning phase. Any deviation from the static estimate during runtime would change the balance of work across processors and cause a reduction in speedup. However, this effect is difficult to quantify.

Comparing naïve unfolding to SGMS. Figure 12 compares the speedup obtained by SGMS and naïve unfolding on 16 processors. There are 3 bars per benchmark. The first bar is the speedup obtained by naïve unfolding for the original stream graph. The second bar is the speedup obtained by naïve unfolding on the same set of benchmarks, but with the size of state variables artificially increased by 16x compared to the original implementation. The last bar the speedup obtained by SGMS for the original stream graph. Figure 12 has to be correlated with Table 1 for better understanding. For benchmarks that are almost completely stateless, such as `dct`, `des` and `mpeg2`, naïve unfolding achieves over 15.5x speedup on 16 processors. This is not surprising as independent iterations run on different processors without any communication. Note that each benchmark nominally has 2 stateful actors, which are the input and output actors. These are used for preserving program order. The small amount of communication needed for these two stateful filters adds very little overhead, and thus completely stateless stream programs achieve close to 16x speedup on 16 processors. The SGMS method for these programs does not unfold the stream graph completely, but only fisses enough actors to get an even work distribution. The selective fissing adds extra splitters and joiners that add non-zero overhead to the steady state. Also, SGMS uses a barrier synchronization at the end of each iteration, whereas in naïve unfolding, the stateful actors perform a point to point synchronization. Because of these two facts, naïve unfolding performs 5-10% better than SGMS for completely stateless stream programs.

For stream programs with many stateful and peeking actors, such as `vocoder`, `radar`, and `fmradio`, SGMS outperforms naïve unfolding by up to 20%. The DMA transfer of state data in naïve unfolding is completely exposed as it is in the critical path. However, all DMA transfers of stream data are overlapped with computation in SGMS. The exposed DMA overhead for naïve unfolding is more pronounced when the state size is artificially increased to 16x the original state size. In this case, SGMS, whose performance is unaffected by the state size increase, outperforms naïve unfolding by up to 35%.

Effect of exposed DMA latency. Figure 13 illustrates the effectiveness of computation/communication overlap. For each benchmark, a version of the C code for SPEs was generated in which the data transfer overhead was completely exposed. For this case, the stage assignment did not separate the DMA operation and the consumer actor into different stages. Rather, they were put in the same stage and the consumer SPE stalls until the DMA operation is completed. The effect of exposed DMA latency is detrimental for all benchmarks. For `channel`, `filterbank`, and `radar`, which have high computation to communication ratios, the effect is not very pronounced and they retain most of their speedups even with exposed DMA latency. `bitonic` and `des` have low computation to communication ratios, and they suffer up to 25% perfomance loss when the DMA latencies are exposed.

Comparing ILP partitioning to greedy partitioning. The integrated fission and processor assignment phase is in part an optimal formulation for bin packing. In addition to deciding how many

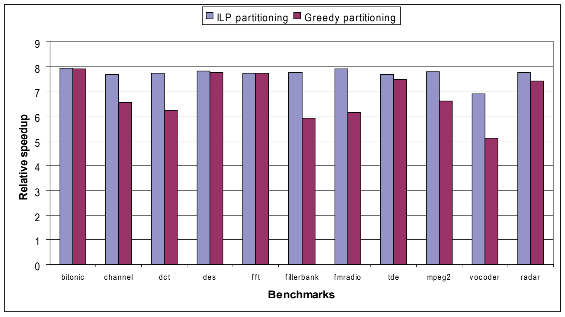


Figure 14: Comparing ILP partitioning to greedy partitioning.

times each actor has to be fissed, this phase also does the assignment with maximal load balancing. Figure 14 compares the optimal formulation with a greedy heuristic. We only compare the 8 processor speedup. This is because the programmer conceived stream graph already has enough parallelism to span 8 processors as shown in Figure 3 and the fission part of the formulation does not fiss any actors. Thus, Figure 14 effectively compares an optimal bin packing formulation to a greedy strategy. We use the Metis [12] graph partitioner as our greedy strategy. The original stream graph is partitioned into *N* parts using Metis, where *N* is the number of processors. The same work estimates are used as weights on the nodes of the graph. Note that this greedy partitioning is similar to the one used in [6]. In [6], a separate communication stage is introduced between steady states to shuffle data between banks. However, to make the comparison fair, the same algorithm for stage assignment is used in both cases which overlaps all DMA transfers with computation. Figure 14 shows that the quality of graph partition using a greedy method depends greatly on the structure of the graph. For example, `fft` and `tde` are just linear graphs with no splitters or joiners. For these cases, the greedy graph partitioner is able to achieve the same load balance as the optimal partitioner. For highly parallel graphs like `filterbank` and `vocoder`, heuristics perform up to 35% worse than an optimal formulation. Overall, the optimal partitioner achieves a geometric mean speedup of 7.6x, whereas the greedy partitioner achieves 6.7x on 8 processors.

Scaling of ILP formulation. The `vocoder` benchmark is used to study how the CPLEX solver run times scales when trying to partition the graph for 2 to 128 processors. `vocoder` is the largest benchmark in the suite, and the solver run times are smaller for all other benchmarks. The solver run times were under 30 seconds for up to 16 processors. The time taken for partitioning on 32, 64 and 128 processors were 2, 6, and 16 minutes, respectively on a Intel Pentium D running at 3.2GHz.

5. Related Work

There is a large body of literature on synchronous dataflow graphs, on languages to express stream graphs, and methods to exploit the parallelism expressed in stream graphs. Even though SDF is a powerful explicitly parallel programming model, its niche has been in DSP domain for a long time. Early works from the Ptolemy group [17, 16, 15] has focused on expressing DSP algorithms as stream graphs. Some of their scheduling techniques [21, 9] have focused on scheduling stream graphs to multiprocessor systems. However, they focus on acyclic scheduling and do not evaluate scheduling to a real architecture.

There has been other programming systems based on the stream programming paradigm, and each of those systems have compilers which target multiprocessors. [8] maps StreamC to a multithreaded processor. This was more of a feasibility study, and the scheduling was done manually. In [27], the authors map the Brook language to a multicore processor. They make use of affine partitioning techniques which are more suitable for parameterized loop based programs. With StreamIt, the stream graph is completely resolved at compile time, and a direct scheduling technique like ours is more effective. Note that any stream programming system in which the computation can be expressed as an stream graph could utilize our scheduling method.

There has been a recent spur of research in the domain of compiling to the Cell processor. CellSs [1] is a stylized C model for programming the cell. The computation is expressed as functions which make all their inputs and outputs explicit in terms of parameters. Functions can be stringed together to form a data flow graph. A run time scheduler treats this graph in the same way a superscalar processor treats operations, and schedules these functions on to the cell SPEs as soon as their inputs are ready. Our work is distinctly different from theirs in that, we use a static compile time schedule which does not have run time scheduling overheads. [13] talks about compiling the Sequoia language to the Cell processor. This paper's focus is more on representing machines with multiple levels of memories, possibly with disjoint address spaces, in a reusable way, and a compiler to automatically target such representations. Our work focuses more on the actual scheduler, and assumes a fixed machine. [2] talks about parallelizing a specific application at multi levels of granularity on the Cell processor. This is more of an experiences paper, and the parallelization was done manually.

The problem scheduling coarse grain actors to processors on a multicore with distributed memory is conceptually similar to scheduling operations to the function units in a multicluster VLIW processor [22, 24]. However, stream graph exposes more optimization opportunities such as the ability to fiss actors. Also, the constraints of limited register space is not an issue on multicores as there is ample memory available to hold the intermediate buffers.

6. Conclusion

The widespread use of multicore processors is pushing explicitly parallel high-level programming models to the forefront. Stream programming is a promising approach as it naturally expresses parallelism in applications from a wide variety of domains. In this paper, we develop methods to automatically map a stream program on to the Cell processor. One of the main issues of getting an even distribution of computation across processors is dealt in an integrated fission and partitioning step that breaks up computation units just enough to span the available processors. The issue of communication overhead is overcome by an intelligent stage assignment, which overlaps all communication with computation. A detailed evaluation of our method on real hardware shows consistent speedup for a wide range of benchmarks. Stream graph modulo scheduling provides a geometric mean speedup of 14.7x over single processor execution across the StreamIt benchmark suite. We compare our method to naïve unfolding that unfolds all actors as many times as the number of processors. Even though naïve unfolding gets speedups similar to SGMS for completely stateless programs, SGMS demonstrates wider applicability by offering consistent speedups on both stateless and stateful programs. Finally, the integrated fission and partitioning phase is largely independent of the underlying architecture, and can be used when compiling to different multicore platforms.

7. Acknowledgments

We thank Dr. Rodric Rabbah at IBM Research for his time, energy, and useful feedback on this research. We also extend our thanks to the anonymous referees who provided excellent comments. This research was supported by the National Science Foundation grants CNS-0615261 and CCF-0347411, ARM Ltd., and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. *Proceedings Supercomputing '06*, 00(1):5, 2006.

[2] Filip Blagojevic, Dimitris S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–100, New York, NY, USA, 2007. ACM Press.

[3] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[4] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. of the SIGPLAN '05 Conference on Programming Language Design and Implementation*, pages 224–236, June 2005.

[5] W. Eatherton. The push of network processing to the top of the pyramid, 2005.

[6] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, New York, NY, USA, 2006. ACM Press.

[7] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, October 2002.

[8] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Soonhoi Ha and Edward A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11):1225–1238, 1991.

[10] H. P. Hofstee. Power efficient processor design and the Cell processor. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, February 2005.

[11] IBM. *Cell Broadband Engine Architecture*, March 2006.

[12] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, September 1998.

[13] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 226–236, New York, NY, USA, 2007. ACM Press.

[14] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, February 2005.

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

[16] E. A. Lee and D. Messerschmitt. Pipeline interleaved programmable dsp's: Synchronous data flow programming. 35(9):1334–1345, 1987.

[17] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[18] W. Mark, R. Glanville, K. Akeley, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proc. of the 30thInternational Conference on Computer Graphics and Interactive Techniques*, pages 893–907, July 2003.

[19] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. In *Microprocessor Forum*, May 2007.

[20] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, 1991.

[21] Jose Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Technical Report UCB/ERL M95/36, University of California, Berkeley, May 1995.

[22] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[23] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation for modulo scheduled loops. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, November 1992.

[24] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, December 2000.

[25] Michael Bedford Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[26] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[27] Shih wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. *Proc. of the 2006 International Symposium on Code Generation and Optimization*, 0(1):196–207, 2006.

[28] D. Zhang, Z. Li, H. Song, and L Liu. A programming model for an embedded media processing architecture. In *Proc. of the 5thInternational Symposium on Systems, Architectures, Modeling, and Simulation*, volume 3553 of *Lecture Notes in Computer Science*, pages 251–261, July 2005.