2009 18th International Conference on Parallel Architectures and Compilation Techniques

Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures

Amir H. Hormati[1], Yoonseo Choi[1], Manjunath Kudlur[2],
Rodric Rabbah[3], Trevor Mudge[1], and Scott Mahlke[1]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{hormati, yoonseo, tnm, mahlke}@umich.edu

[2]NVIDIA Corp.
Santa Clara, CA
mkudlur@nvidia.com

[3] IBM T.J. Watson Research Center
Hawthorne, NY
rabbah@us.ibm.com

ABSTRACT

Increasing demand for performance and efficiency has driven the computer industry toward multicore systems. These systems have become the industry standard in almost all segments of the computer market from high-end servers to handheld devices. In order to efficiently use these systems, an extensive amount of research and industry support has been devoted to developing explicitly parallel programming paradigms, such as streaming models, and new compiler techniques.

One important challenge that arises in multicore systems is the ability to dynamically adapt a running application to a target architecture in the face of changes in resource availability (e.g., number of cores, available memory or bandwidth). In this paper, we focus on the increasingly important area of streaming computing and introduce Flextream as a flexible compilation framework that can dynamically adapt applications to the changing characteristics of the underlying architecture. We believe this is an important contribution as software developers grapple with the details of parallelism in a rapidly changing architecture landscape. Flextream achieves its goals through a combination of static compilation and dynamic adaptation techniques. Our results indicate that Flextream's approach can achieve high-performance resource allocations that are within an average of 9% of the optimal solution with low overhead for a wide range of streaming applications.

1. INTRODUCTION

Many-core processors provide a lot of flexibility in that they can potentially speed up the execution of individual applications (because of increased parallelism), while also having the ability to run many applications at the same time. As the number of applications that can effectively use multiple cores increases, it will become necessary to develop strategies that can adequately manage the allocation of resources between applications. Resource allocation is a challenging problem because application behavior (and hence resource requirements) can often vary in unpredictable ways, depending on factors that include dynamic workloads and variability in end-user scenarios. The issue is made more challenging by the numerous heterogeneous architectural resources that are already exposed to software (e.g., the compiler). We believe that managing the allocation of resources effectively requires many non-trivial tradeoffs, and we introduce Flextream as a means to address this issue.

Specifically, we address the issue of provisioning an individual application to run on a heterogeneous architecture under varying configurations of resource allotments. In doing so, applications are able to efficiently and effectively adapt, at runtime, to changes in the number and kind of resources at their disposal. For example, consider a mobile device that serves as a multimedia player and an internet browser. If the user is running only one of the two applications, then that application can potentially exploit all of the available resources in the device. However, as soon as the user also starts browsing the web, the resources available for the media player must change to accommodate the new application. If either of the applications is not properly provisioned to run on a varying number of resources, the end-user experience will almost surely be a poor one.

Static compilation approaches, in general, can generate high-quality resource allocations offline. However, such solutions are often sensitive to runtime variations in resource availability. In other words, any change in the underlying architecture's parameters, such as available on-chip memory or the number of cores, will result in an inefficient execution of a statically scheduled application in the best case, or code that can not execute in the worst case.

One potential solution to this problem is to compile alternative versions of an application, and to dynamically switch between versions according to the resources that are available in the architecture. For example, the media application running on an 8-core device can be provisioned to run on either 1, 2, 4, or 8 cores. The obvious deficiencies of this approach are three fold. First, this strategy can lead to large amounts of code bloat. Second, it may be impractical to statically consider a high number of architectural configurations. Lastly, the application may have to execute an inefficient fail-safe implementation (e.g., sequential) if the runtime scenario yields a set of resources that was not statically considered.

An alternative solution is dynamic compilation, where the application is repeatedly compiled at runtime when resources change—this can arise if the number of available cores, or the amount of memory that is available, or the available bandwidth varies. This is a promising approach because it can continuously adapt to changes in resource availability, if only the costs of compilation and adaptation can be made low enough to be practical. In order to keep the costs of compilation down, the runtime compiler is likely to be limited to a small set of optimizations. Furthermore, if we consider all of the resource ingredients that can vary at runtime, it will be quite challenging to engineer an efficient solution that addresses all of them well.

In this work, we propose a compilation and runtime adaptation system called Flextream. It is aimed at addressing the challenges described above in the context of streaming applications. Streaming is an increasingly important programming paradigm because it addresses the parallel programming challenges among several application domains and tiers of the computing industry (from mobile computing to high-end server farms).

In Flextream, a streaming application is represented as a graph, where the nodes encapsulate computation, and the edges between nodes describe dataflow. A stream program (graph) is mapped to a many-core heterogeneous architecture by assigning nodes to cores, and dataflow to communication channels between cores (e.g., DMA transfers between cores, or between main and local memories). The main innovation in Flextream is an *adaptive stream graph modulo scheduling* algorithm that combines the benefits of static scheduling with the advantages of dynamic adaptation. This strategy, of using an adaptive hybrid (static-dynamic) compilation approach, can lead

1089-795X 2009
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/PACT.2009.39

to significantly better resource utilization, and can help deliver the promise of many-cores to end-users.

Flextream consists of two main components. The first part performs static compilation of an application to a virtualized multicore system using heuristics for controlling the amount of parallelism in the graph, and an integer linear programming (ILP) formulation to find the optimal mapping of nodes to resources (i.e., work partitioning). The second part consists of a light-weight online (dynamic) adaptation system that modifies the active schedule based on the available resources in the architecture. Dynamic adaptation consists of several phases including finding a new processor assignment, stage assignment, and buffer allocation. The online phases are designed to be light-weight and yet produce efficient results.

In this paper, we mainly focus on heterogeneous systems with distributed memory similar to the IBM Cell [9] processor. Using the proposed framework, an application is statically compiled for a configuration of the architecture with the greatest number of resources which may include processing elements, on-chip storage and bandwidth. This results in high-quality solutions for a specific configuration. The dynamic light-weight layer uses the result of the static compilation as a hint to quickly discover an efficient solution for the new system configuration. Our experiments show that assisting the online adaptation phase with a static solution reduces runtime overhead and greatly improves the quality of the solutions that the online phase discovers. Our approach eschews the need for recompilation when resources change, and thus enables software developers to produce adaptive and high-quality streaming applications. The online adaptation phase uses a technique similar to [18] (called Multicore Streaming Layer or MSL) to stop the current schedule and distribute the new schedule between the processors. More details about this technique are mentioned in Section 2.2.

This paper makes the following contributions:

- An efficient framework for adaptive compilation of streaming applications to heterogeneous multicore systems is proposed.
- A parallelism-tuning heuristic coupled with a scalable work partitioning based on ILP formulation is proposed to find a static software pipelined scheduling for streaming applications.
- Highly efficient dynamic work redistribution and buffer allocation algorithms are introduced to adapt the software pipelined schedule dynamically to efficiently exploit the capabilities of the target platform.

The rest of the paper is organized as follows. In Section 2, the target architecture, input language, and multicore streaming layer are discussed. Then, the static compilation and online adaptation layer of Flextream are discussed in Section 3. Finally, in Section 4, the framework is evaluated. Section 5 discusses some of the related works that motivated this system.

2. BACKGROUND

2.1 Architecture

The compilation target in this paper is a streaming memory multicore architecture where on-chip memory structures are addressed as local memory and are explicitly managed. Such architecture provides the compiler with a great deal of flexibility in terms of orchestrating code and data locality, and managing communication granularity, frequency, and latency.

The target system is similar to the Cell processor in terms of the high-level architecture. It consists of a more powerful master processor and several slave processing elements. The master processor is similar to the PowerPC core in the Cell processor running at 2GHZ with 32KB L1 and 1MB L2 cache. Each slave core contains a local memory for instruction and data, called *local store*, and a memory flow control (MFC) unit which can perform DMA operations to and from the local stores independent of the cores. The slave cores can only access the local store, so any sharing of data has to be performed through explicit DMA operations. The ability to perform asynchronous DMA operations allows overlap of computation and communication, and is leveraged for efficient software pipelining of stream graphs. The multicore system used during static compilation (Section 3.1) is similar to the processor in Figure 1 and has 32 slave cores. The actual physical processor used during online adaptation (Section 3.2) also has the same architectural template but the number of slave cores varies in each experiment from 2 to 32.

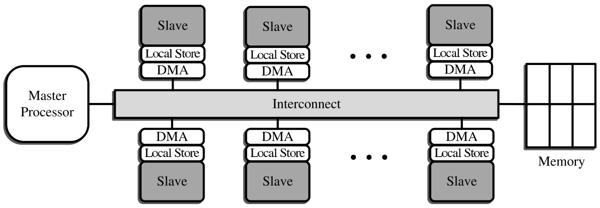


Figure 1: General architecture template

2.2 Multicore Streaming Layer

We use the runtime system introduced in [18] to dynamically manage resource allocations. The runtime system, called the *multicore streaming layer (MSL)*, supports loading and unloading of computation (e.g., streaming actors) on different cores, allocating local and global buffers, and managing DMA transfers for orchestrating communication. The MSL also consists of a set of commands that the online adaptation system can use to migrate from one schedule to another by moving computation between cores, allocating new buffers in different regions of local or global memory, and so on.

In our implementation of the MSL, the master processor generates the commands that are necessary for adapting an extant schedule. These commands are sent to the slave processors through memory mapped registers called mailboxes. Each slave processor runs a very light-weight manager that is able to receive the commands from its input mailbox, decode the instructions, and act on them. Based on the commands, the slave processors can allocate buffers in their local stores, setup DMA transfers and run code for a desired duration. The overhead of delivering the commands varies according to the size of the command and the latency of mailbox transfers. The results that are presented in latter parts of this paper show that we achieve a very low overhead when adapting to resource changes. This paper does not detail the design of the command system. The interested reader is referred to [18] and [10].

2.3 Stream Programming Model

Flextream is best suited for applications with an abundance of parallelism that is amenable for static scheduling. Thus, we focus on stream programming models that are based on synchronous data flow (SDF) models [14]. In SDF, computation is performed by actors, which are autonomous and isolated computational units. Actors communicate through dataflow channels, often realized as FIFOs. SDF and its many variations expose the input and output processing rates of actors, and in turn this affords many optimization opportunities that can lead to very efficient schedules (e.g., allocation of actors to cores, and FIFOs to local stores and DMAs).

We distinguish between stateful and stateless actors. A stateful actor modifies its local state and maintains a persistent history of its execution. For our purposes, we assume that all computation that is performed in an actor is largely embodied in a `work` method. The work method run repeatedly as long as the actor has data to consume on its input port. The amount of data that the work method consumes is called the *pop* rate. Similarly, the amount of data each work invocation produces is called the *push* rate. Some streaming languages (e.g., StreamIt [16]) provide a non-destructive read which does not alter the state of the input channel. The amount of data that is read in this manner is capture by the *peek* rate. Unlike a stateful actor, which restricts opportunities for parallelism, a stateless actor is data-parallel in that every invocation of the work method does not depend on or mutate the actor state. The semantics of stateless actors thus allow us to replicate a stateless actor. This opportunity is quite fruitful in scaling the amount of parallelism that an application can exploit, as past work has shown [5, 6].

We use the StreamIt programming language to implement streaming programs. StreamIt is an architecture-independent streaming language based on SDF. The language allows a programmer to algorithmically describe the computational graph. In StreamIt, actors are known as filters. Filters can be organized hierarchically into *pipelines* (i.e., sequential composition), *split-joins* (i.e., parallel composition), and *feedback loops* (i.e., cyclic composition). StreamIt is a convenient language for describing streaming algorithms, and its accompanying static compilation technology makes it suitable for our work.

3. COMPILER FRAMEWORK

This section describes our method for scheduling a stream graph onto a heterogeneous streaming multicore system. The objective is to obtain a maximal throughput adaptive modulo schedule of the stream graph, taking computation/communication overheads and memory requirements into account. The structure of the Flextream compilation framework is shown in Figure 2. The compilation is divided into two separate phases, static compilation and online(dynamic) adaptation. In the next two sections, the details of the static and online phases are discussed.

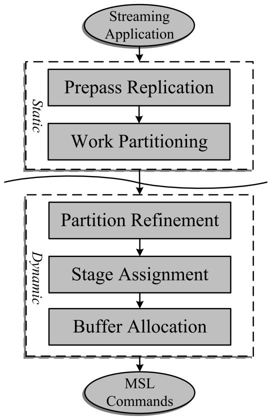


Figure 2: General flow of the Flextream framework

Before talking about details of the compilation steps, it is important to understand how an application compiled by Flextream behaves at runtime in the face of dynamic resource changes. Figure 3 shows an example runtime scenario. At each point during the execution, only one schedule is active. Execution starts with $schedule1$. If some of the currently-used resources become unavailable or new resources become free, an online reschedule becomes necessary. The new schedule is marked by $schedule2$ in the figure. The process of migrating from $schedule1$ to $schedule2$ consists of three main parts. First, the online adaptation phase has to generate the new schedule and the necessary MSL commands using the solution found by static phase. Second, the current schedule has to be stopped(drained). The latency of this case is directly related to the number of stages in the module schedule and the work of the maximally loaded processor. Third, the generated commands have to be sent to the active processors. In the experiments section, the overhead of each of these phases and also the performance of a full runtime scenario are discussed.

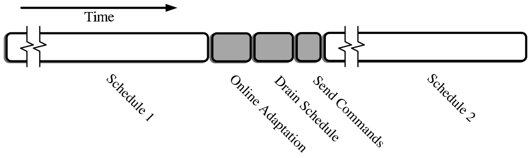


Figure 3: Overall execution flow at runtime in the case of resource changes.

3.1 Static Compilation

The static phase's goal is to find an optimal schedule for a virtualized member of a family of streaming multicore processors while considering bandwidth, storage and the processing capabilities of the system. This phase consists of two major sub-phases shown in Figure 2. First, a prepass replication is performed on the stream graph to adjust the amount of parallelism for the target system by replicating actors. Second, an ILP formulation is used to optimally partition the work between the slave cores of the target system. The virtualized system used in this phase is generally the most powerful processor of a streaming multicore family. For example, if a streaming application should be compiled for the IBM cell processor family with 4, 8, or 16 processors with local store size of 128KB or 256KB, the 16 processor version with 256KB is chosen as the virtualized system. Selecting the virtualized system in this manner, increases the freedom of the next phases to find a high quality schedule in case the program is ported to another configuration with a more limited set of resources or the availability of the resources changes at runtime.

Compared to [12], Flextream's static phase takes a different approach toward static modulo scheduling. The static phase consists of a separate step to perform replication instead of integrating it with the ILP formulation. This greatly improves the scalability of the ILP formulation and enables the inclusion of other crucial constraints about memory allocation and data transfer overheads. Ignoring these factors can have a significant negative impact on the runtime performance in systems with low-bandwidth interconnects.

3.1.1 Prepass Replication

Figure 4 shows the theoretical speedup possible for a set of unmodified stream programs for 2 to 64 processors. The actors present in the programmer-conceived stream graph are assigned to processors in an optimal fashion such that the maximal load (work) 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.

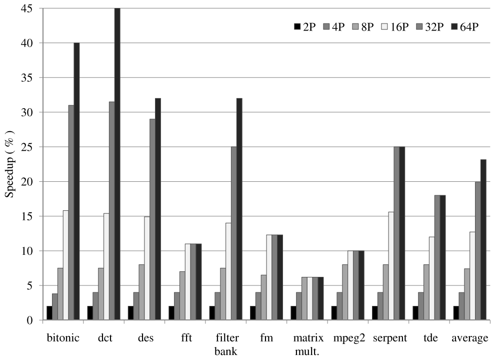
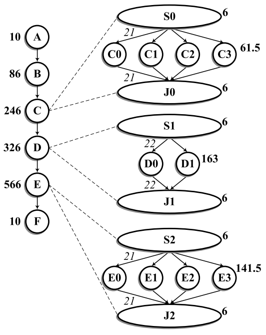


Figure 4: Theoretical speedup in the absence of replication.

Most benchmarks just do not have enough actors to span all processors. For example, `fft` has only 17 actors in its stream graph, therefore no speedup is possible beyond 17 processors. Another reason for the speedup limitation 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 how long an actor's work function will take to execute on a processor when coding the function. This leads to less scaling on 16 or more processors.



(a)

Proc.	*Before*	*After*
P0	A	A, E0
P1	B	B, C0
P2	C	C1,C2, C3
P3	D	D0
P4	E	E1
P5	F	F, E2
P6		E3
P7		D1

(b)

Figure 5: This figure shows an example stream graph and how replication is performed. Part (a) shows the original graph and the version after replication. In part (b), the partitions before and after replication are shown.

Most of the stream benchmarks are completely stateless, i.e., all actors are data parallel [6]. In fact, only `mpeg2` has actors that are stateful. Data parallel actors can be replicated any number of times without changing the meaning of the program. Replicating 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.

To provide the next phases of the compilation flow with ample opportunity to efficiently utilize the target system's capabilities, a prepass replication is performed on the stream graph. Algorithm 1 shows the general steps of this phase. The main task is to heuristically replicate larger actors based on an estimate of the optimal work partitioning of the current graph. Maximally replicating the larger actors may not always result in the best solution for the next phase. Excessive replication of actors is always discouraged, because that increases split/join overhead and overall code size. Therefore, graph partitioning on the original stream graph is used to estimate the solution of the work partitioning phase. The number of requested partitions is set to the number of processors in the virtualized target processor.

Graph partitioning is fairly fast and produces a reasonable estimate of the optimal work distribution of the stream graph for the virtualized target system without considering low-level constraints such as memory size, interconnect bandwidth, etc.. Each resulting partition corresponds to one of the cores in the multicore system. This solution approximately reflects the quality of the optimal solution if the current stream graph is used. Next, the replication algorithm tries to balance the partitions by replicating the largest actor in the partition with the maximum amount of work and moving the new replicas to the partition with minimum work. This process is repeated until the ratio between the maximum workload and minimum workload is less than the balance factor specified as an input to the algorithm or no more replication is possible. The while loop in Algorithm 1 performs the partition balancing task. Lines 8-10 check the degree of imbalance between partitions. Lines 14-16 determine how many replicas of the actor selected from the largest partition should be created.

An example of the prepass replication algorithm is shown in Figure 5. In this example, the virtualized target system has 8 cores. The original graph, shown in the left part of Figure 5(a), has only 6 nodes and clearly will not efficiently use all 8 cores. The replication algorithm performns an initial graph partitioning on this stream graph and then tries to replicate nodes and balance the partitions. The *balance factor* for this example is set to 1.5. Figure 5(b) shows the partitions before and after replication. At the end, the ratio between maximum weight (P1) and minimum wight (P2) is 1.3. The modified graph is illustrated in the right part of Figure 5(a).

Algorithm 1 Prepass Replication Algorithm

```
Input: G:(V, E), #virtualProcessors, balanceFactor
1  partitions ← PartitionGraph(G, #virtualProcessors);
2  while true do
3      SortPartitionsByWeight(partitions);

       { Find partitions with max and min weights. }
4      repeat
5          maxPartition ← NextMaxWeightPartition(partitions);
6      until maxPartition has a dividable node
7      minPartition ← MinPartitionWeight(partitions);

       { Check the overall balance of the partitions.}
8      if (Weight(maxPartition) < Weight( minPartition) * balanceFactor) then
9          Finish;
10     end if

       {Find a actor in the max partition that can be replicated.}
11     repeat
12         actor ← NextLargestFilter(maxPartition);
13     until (actor can be replicated)

       { Find out how many times the actor should be replicated.}
14     replicationFactor ← Work(actor) / (Weight(maxPartition) - Weight(minPartition));
15     replicationFactor ← Max(replicationFactor, 2);
16     newFilters[ ] ← Split(actor, replicationFactor);

       {Modify the min and max partitions.}
17     AddTo(minPartition, newFilters[1]);
18     RemoveFrom(maxPartition, actor);
19     AddTo(maxPartition, newFilters[2..replicationFactor]);
20 end while
```

3.1.2 Work Partitioning

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 the steady state. 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 and data transfer operations to the processors. The throughput of the software pipeline is determined by the load on the maximally loaded processor. For each actor and DMA transfer in the stream graph, the following ILP formulation finds a valid assignment based on the computational power of processors, bandwidth of the interconnect, and amount of on-chip memory.

In the formulation, maximization of throughput is the main objective. 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. A set of 0-1 integer variables is introduced to find the processor assignment for actors and data transfer operations. These variable are explained below:

- $a_{ij} = \{0, 1\}$: Indicates if actor i is running on processor j

- $b_{i_1 i_2 j} = \{0, 1\}$: This variable will be 1 if connected actors (producer-consumer) i_1 and i_2 are both assigned to processor j

Assuming that there are n actors in the stream graph and m processors in the target system, i is between 0 and $(n - 1)$ and j is between 0 and $(m - 1)$. A set of constraints are designed to find a valid actor and DMA assignment under memory, bandwidth and performance characteristics of the target system. The following constraint ensures that each actor is assigned to exactly one processor.

$$\sum_{j=0}^{P} a_{ij} = 1, \qquad \forall i \tag{1}$$

The $b_{i_1 i_2 j}$ indicator variables serve two purposes. First, they are necessary to ensure that a DMA transfer is not introduced between two connected actors if they are on the same processor. Second, the b variables help in buffer allocation constraints because the size of the buffers between a pair of connected actors varies based on when they start execution and whether they are on the same processor. The following inequalities are used for setting the b variables.

$$\begin{aligned} b_{i_1 i_2 j} &\leq a_{i_1 j} && \forall \text{ connected actor pairs } i_1,i_2 \qquad (2)\\ b_{i_1 i_2 j} &\leq a_{i_2 j} && \forall \text{ connected actor pairs } i_1,i_2\\ b_{i_1 i_2 j} &\geq a_{i_2 j} + a_{i_1 j} - 1 && \forall \text{ connected actor pairs } i_1,i_2 \end{aligned}$$

The throughput is decided based on the workload of the maximally loaded processor which is the maximum of the computation workload and the data transfer workload across all processors. In the schedule, it is always assumed that the DMA transfer between a pair of connected actors is located on the processor on which the destination actor is running. The following two inequalities denote the relation between II and the workload of each processor.

$$\sum_{i=0}^{N}(a_{ij} \times W_i) \leq II \qquad \forall j \qquad (3)$$

$$\sum_{(i_1\ i_2)}^{|E|}((a_{i_2 j} - b_{i_1 i_2 j}) \times D_{i_1 i_2}) \leq II \qquad \forall j \qquad (4)$$

W_i in Equation 3 indicates the work estimate of actor i on processor j. $D_{i_1 i_2}$ show the data transfer cost between a pair of connected actors i_1 and i_2. Equation 4 uses $b_{i_1 i_2}$ to ensure that a DMA transfer between actors is only added if they are assigned to different processors.

As it will be discussed later, the amount buffering between two connected actors depends on both where they are running and what stage they are in. Since stage assignment is a phase of the online adaptation layer, the ILP formulation can only have an estimate of the actual memory consumption of the current mapping. To obtain this estimate, it is assumed that two connected actors will be in consecutive stages if they are not on the same processor; otherwise, they are in the same stage. Based on the results of the stage assignment phase, this is a practical overestimate of the actual buffer usage. The following set of inequalities is added to the formation for the purpose of buffer allocation.

$$\sum_{(i_1,i_2)}^{|E|}[(2a_{i_1 j} + 2a_{i_2 j} - 3b_{i_1 i_2 j}) \times Buff(i_1,i_2)] \leq Mem_j, \qquad \forall j \qquad (5)$$

For each pair of connected actors i_1 and i_2 and a processor j, there are four possible values for $a_{i_1 j}$ and $a_{i_2 j}$. In each of these cases, the amount of necessary buffering differs. Equation 5 is an estimate of the actual memory requirement. Sections 3.2.2 and 3.2.3 talks about the mechanics of buffer allocation at runtime in more detail.

Figure 6(a) illustrates the result of the ILP-based work partitioning on the graph shown in Figure 5(a). Since the cores in our system are able to perform DMAs and run computation at the same time, the workload of each processor would be the maximum of the computation and data transfer workloads. The II of this system is determined by the maximally loaded processor, P0. Comparing the achieved II of 184 with the single core performance of the graph (sum of all the weights in the original graph) reveals that a 6.8x speedup is achieved on 8 cores.

3.2 Online Adaptation

After static compilation is performed, the generated code can be efficiently executed on a system that matches the virtual specification used during the static compilation. As mentioned before, due to the desire for porting software within members of a streaming multicore family and also for efficiently tolerating resource availability changes at runtime, online adaptation is crucial for software developers. In this section, we talk about various phases of the light-weight online adaptation layer in the Flextream framework.

Algorithm 2 Algorithm for Partition Refinement

```
Input: processorMap(processor:actor[]), #physicalProcessors
Output: newProcessorMap
     {Assign one workload from the current processor map to each physical processor}
 1   SortByNumberOfFiltersAscending(processorMap)
 2   for i ← 1 to #physicalProcessors do
 3       (processor:actor[]) ← RemoveNextPair(processorMap);
 4       AddTo(newProcessorMap, (processor:actor[]));
 5   end for

     {Prioritize the remaining actors and the chosen processor workloads}
 6   remainingFilters ← AllFiltersIn(processorMap);
 7   SortFiltersByWeightAscending(remainingFilters);
 8   SortByWorkAssignmentDescending(newProcessorMap);

     {Distribute the remaining actor between the chosen processor workloads}
 9   weightThreshold ← TotalRemainingWeight(remainingFilters ) / #physicalProcessors;
10   repeat
11       actor ← RemoveNextFilter(remainingFilters);
12       currentWeight ← Weight(actor);
13       AddTo(currentList, actor);
14       if (currentWeight > weightThreshold) then
15           processor ← NextPhysicalProcessor(newProcessorMap);
16           AddTo(newProcessorMap, processor:currentList);
17           Clear(currentList);
18           currentWeight ← 0;
19       end if
20   until remainingFilters is not empty
```

Online adaptation, is mainly designed to perform light-weight adaptation of modulo scheduling solutions at runtime for the current active configuration. As shown in Figure 2, this part consists of three main steps, *Partition Refinement*, *Stage Assignment*, and *Buffer Allocation*. The first step tries to change the mapping of actors to processors based on the number of available processors to rebalance work assignment and memory consumption on each core. The solution specifies how actor executions are overlapped across processors (in space). The stage assignment step determines how the executions are overlapped in time by specifying the starting order of the actors and DMAs. The last step of the online adaptation, buffer allocation, tries to efficiently fit the buffers in the available storage units.

3.2.1 Partition Refinement

The virtual multicore system used in static compilation is always a superset of the actual physical system meaning that it has more cores, more memory, etc.. Therefore, the runtime configuration, which Flextream has to target, will always have more limited resources. Partition refinement is a general step that, at runtime, tunes the actor-processor mapping to the real configuration of the system. The algorithm discussed here for performing the refinement concentrates only on the computation workload of each core in a streaming multicore system, but the heurisitics can be extended to account for memory and bandwidth.

Assume that the virtualized system had n slave cores (number of virtual cores) and the real system has m cores (number of physical cores). m is less than n because the real system is a less powerful member of the multicore family or some of the cores in system with n cores have to be used to perform more critical tasks. The main objective here is to reassign the actors to m cores with low overhead at runtime.

As shown in Algorithm 2, the general idea is to choose m processor assignments from the original n assignments created by the static phase. Then, take all the actors in the $(n - m)$ remaining partitions and try to evenly distribute them between the chosen m partitions. Since solving this problem based on another ILP formulation or graph partitioning will have significant overhead at runtime, a heuristic-based approach is taken.

In the algorithm, lines 1-5 choose the m work assignments with the least number of actors from the original n. The reason the assignments with least number of actors are chosen first is to increase the freedom of the second phase of the algorithm to evenly distribute the remaining actors. Then, in lines 6-8, the remaining actors and the m chosen assignments are prioritized. The remaining actors are all put in one list and sorted by work estimate (weight) in ascending order. The chosen assignments are sorted based on the total weight of each assignment in descending order. Line 9 calculates, in the ideal situation, what fraction of the remaining actors will go to each of the chosen assignments. Lines 10-20 walk through the remaining actors(sorted by ascending weight) and assigns them to the currently chosen processors(sorted by descending weight) based on the weight threshold calculated in line 9.

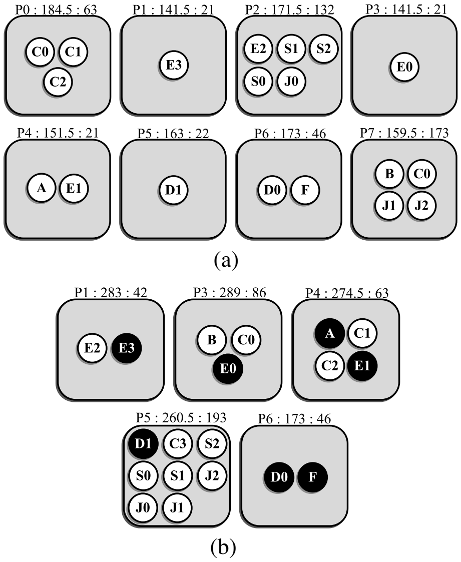


Figure 6: Part (a) shows the solution of the work partitioning onto 8 cores for the example shown in Figure 5(a). Part (b) illustrates the solution of the partition refinement if number of cores changes to 5. The actors shaded in black exist in the related processors in the original solution(a) as well as final solution(b).

Figure 6(b) shows the refinement solution for the example in Figure 5(a) when the number of cores is reduced from eight to five. In this figure, the five processors are the processors that are chosen from the original work assignment shown in Figure 6(a). The highlighted nodes denote the nodes that were originally assigned to these processors. The rest of the nodes are mapped to these processors as a result of the refinement pass. The text above each processor shows the name of the processor in the original work assignment, and the computation workload followed by the data transfer workload. In the new work assignment, the II is 289 determined by P3. The optimal static solution for the 5-core problem will have II of 283 which is about 3% faster than the solution shown here.

Although the algorithm in this section ignores memory requirements, it is sufficient to modify the heuristics used here to consider memory requirements of the assignments. Prioritization of the remaining nodes after the initial selection can be done based on an affinity function that estimates the extra necessary memory if a node is added to a chosen processor. This type of priority function helps to keep the memory usage of each assignment under control.

3.2.2 *Stage Assignment*

The processor assignment obtained by the method described in the previous section provides only partial information for a software pipeline. Namely, it specifies how actor executions are overlapped across processors, but it does not specify how they are overlapped in time. The only goal of the processor assignment step is to 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*. 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 pipeline is drained by deactivating stages during the final few iterations.

Algorithm 3 Actor Stage Assignment Algorithm

```
Input: G:(V, E), processorMap(processor:actor[])
Output: actorStageMap(actor:int)
1  for all (actor f1 in G in topological order) do
2      maxStage ← 0; flag ← false;
3      for all actor f2 in parents f1 do
4          if (Stage(actorStageMap, f2) $\geq$ maxStage) then
5              maxStage ← Stage(actorStageMap, f2);
6              if (Processor(processorMap, f1) != Processor(processorMap, f2))
               then
7                  flag ← true;
8              end if
9          end if
10     end for
11     if (flag) then
12         stage ← maxStage + 2;
13     else
14         stage ← maxStage;
15     end if
16     AddTo(actorStageMap, f1:stage)
17 end for
```

The main 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 i is assigned is denoted by S_i. In addition, the processor to which i is assigned is denoted by p_i. The following rules enforce data dependence and ensure DMA overlap.

- $(i_1, i_2) \in E \Rightarrow S_{i_2} \geq S_{i_1}$, i.e., the stage number of a consuming actor should come after the producing actor. This is to preserve data dependence.
- If $(i_1, i_2) \in E$ and $p_{i_1} \neq p_{i_2}$, then a DMA operation has to be performed to transfer the data from p_{i_1} to p_{i_2}. The DMA operation is given a separate stage number S_{DMA}. The inequality $S_{i_1} < S_{DMA} < S_{i_2}$ 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.

As shown in Algorithm 3, a topological traversal of the stream graph is necessary to assign stages to actors. For each actor, the maximum stage of its parents is found and a flag is set if the parent with maximum stage is not on the same processor as the actor. This part of the algorithm is done in lines 3-10. For each actor, if the parent with maximum stage number is on a different processor, there will be a two stage gap between the parent and the child. Otherwise

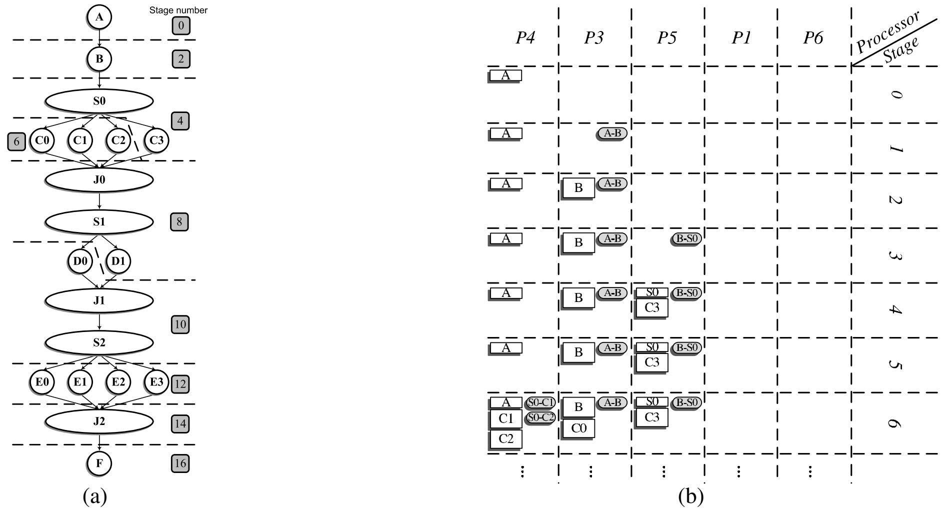


Figure 7: The example shown in 5(a) after stage assignment is illustrated in part (a). The number in the gray boxes show the stage number of the actors marked by the dashed lines. Part (b) demonstrates the execution of the first 6 stages of the schedule found by Flextream.

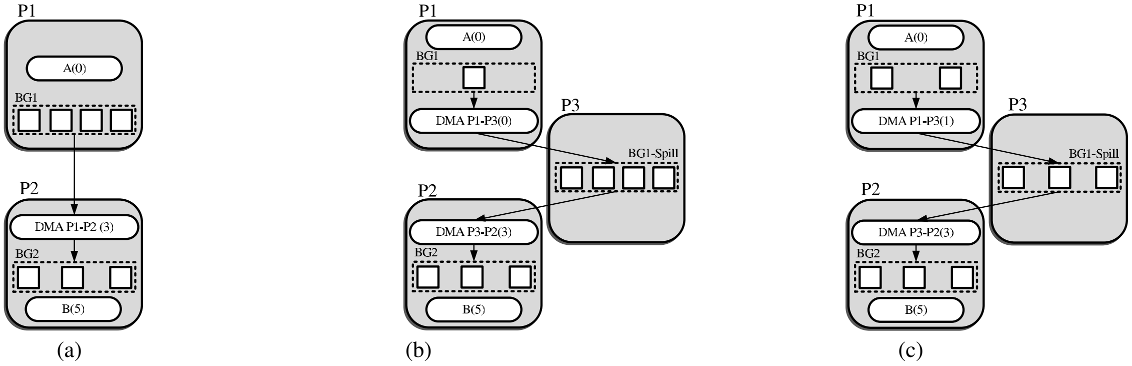


Figure 8: Different approaches to buffer allocation for a producer-consumer pair is demonstrated in this figure. In (a), the original arrangement of buffers before performing buffer allocation is shown. In parts (b) and (c), two approaches that Flextream could take and their effects on the overall schedule and memory consumption is illustrated.

the child actor can be placed in the same stage as the parent with maximum stage (lines 11-16). The result of the stage assignment is illustrated in Figure 7(a). There are total of 16 stages in this schedule. One interesting point in this figure is that actors $D0$ and $D1$ are not in the same stage. This is because $D1$ is located on the same processor as $S1$. This figure does not show the stages for DMA operations for the sake of figure readability. Figure 7(b) shows how the schedule runs based on the work assignment and stage assignment. In this figure, DMAs are shown as shaded boxes. This figure demonstrates how stage assignment specifies the ordering between actors in time and work partitioning (and partition refinement) determines actor-to-processor (space) assignment.

3.2.3 Buffer Allocation

Buffer allocation tries to efficiently fit the storage requirements of the schedule, found by the previous phases, into the available memory units. In the software pipelined schedule, connected actors communicate through a set of buffers. The number of necessary buffers for a producer-consumer pair varies depending on the time they start (stage mapping). In this section, the set of buffers between a producer-consumer pair is called a *buffer group*. Based on its stage number, a producer actor could be executed multiple times before one of its consumers is ever executed. The number of buffers in a buffer group needed to store the output of a producer (actor or DMA operation) assigned to stage S_p feeding a consumer(actor to DMA operation) on stage S_c can be calculated as $S_c - S_p + 1$. For example, in Figure 7(b), the number of buffers necessary between actor $S0$ and DMA operation $S0$-$C1$ is 2 because they are in stages 4 and 5, respectively. All the phases before buffer allocation assume that the buffers between a producer actor and a DMA operation are stored in the *local memory* of the processor on which the producer is running. Symmetrically, the buffers between a DMA operation and a consuming actor are stored on the local store of the consuming processor.

In the work partitioning, partition refinement and stage assignment, it is assumed that all the buffer groups will fit in the local stores of the cores on which the corresponding actors are running. Therefore all the DMAs are from local store to local store. In some situations, based on the stage map and amount of buffering that is needed between a pair of actors, the local store may not be large enough to fit all the buffers. In those cases, in order to have a schedule that can actually run on the target system, some of the buffer groups have to be spilled to other local stores that have empty space or main memory. Spilling buffer groups will result in changes in the schedule. Basically, after moving a buffer group to another storage unit, new DMAs have to be added to the schedule. These DMAs are needed to ship the data between the local store of the processors on

Algorithm 4 Buffer Allocation Algorithm

```
Input: procMap(processor:actor[]), stageMap(actor:int)
     {Compute memory usage per local store based on work and stage assignment}
 1   memoryUsage[processor:long] $\leftarrow$ Update(procMap, stageMap);

     {Find the processors that their local store needs spilling}
 2   (victimProcs[], nonVictimProcs[]) $\leftarrow$ FindVictims(memoryUsage);
 3   SortByWorkLoadDescending(victimProcs);

     {Find victim buffers}
 4   for all Processor p in victimProcs do
 5       savings = 0;
 6       BufferGroup buffs[] = BufferGroups(p);
 7       SortBySpillSizeDescending(buffs);
 8       for all BufferGroup bg in buffs do
 9           savings $\leftarrow$ savings + SpillSize(bg);
10           if (memoryUsage[p] - savings < LocalStoreSize(p)) then
11               break;
12           end if
13           add(victimBuffers, bg);
14       end for
15   end for

     {Find target location for the victim buffers and fix the schedule}
16   for all BufferGroup bg in victimBuffers do
17       target = findTarget(bg, memoryUsage, nonVictimProcs);
18       MoveBufferTo(bg, target);
19       newDMA[] = CreatNewDMA(bg);
20       UpdateStageMap(newDMA);
21       Update(memoryUsage);
22   end for
```

which the related actor is running and the new memory unit. The addition of the DMAs can increase the workload of the processors resulting in an increase of II. Since the cost of a DMA to and from main memory is significantly higher than the cost of a transfer between local stores, it is desirable to first exploit the free space in the local stores before utilizing the main memory.

The buffer allocation algorithm is shown in Algorithm 4. First, the memory usage of the current schedule is calculated based on the processor and stage assignments (line 1). Then, the list of victim (overcommitted) processors is formed. This list contains all processors that exceed the size of their local stores (line 2) and is sorted in descending order by the amount of work that is assigned to each processor (line 3). The victim processors are given the chance to make use of other local stores with priority given to processors with more work. It is more beneficial to spill the buffers into the processors with more work first, because these spilled buffers are more likely to fit in other processors' local stores, resulting in less DMA overhead. Then, in lines 4 to 15, the list of buffer groups that do not fit in the corresponding local stores is extracted. This part tries to spill as few buffer groups as possible (by spilling the largest ones first) to reduce the overhead of DMA transfers. At the end(lines 16-22), the algorithm goes through the selected buffer groups and tries to move them to other local stores first and then main memory. After finding the target (local store, main memory), for each spilled buffer group, new DMAs are added to the schedule and the current stage assignment is updated.

The function, *UpdateStageMap*, in line 20 of Algorithm 4 can take two different approaches for updating the stage assignment and adding the new DMAs. These approaches are illustrated in Figure 8. The first part of the figure shows the stage and processor assignment for a pair of actors. Actors A and B are mapped $P1$ and $P2$ and start at stages 0 and 5. A DMA located on $P2$ in stage 3 transfers data between A and B. The buffer groups and their placement before running the buffer allocation are shown in Figure 8(a). Assume that out of the 4 buffers in buffer group 1 ($BG1$), 2 will not fit in $P1$'s local store. $P3$ is a candidate for spilling in this buffer group. In Figure 8(b), the first possible solution to buffer allocation is shown. In this case, the buffer group is moved to $P3$'s local store and a new DMA is added to $P1$ in stage 0. The original DMA between $P1$ and $P2$ is modified to read from $P3$'s local store. The number of buffers on $P1$'s local store is reduced to 1. Since the new DMA (between $P1$ and $P3$) is in stage 0 and there is only 1 buffer between this DMA and A, the DMA has to run sequentially after A is done, increasing the workload of $P1$. The second approach, shown in Figure 8(c), tries to place the new DMA (between $P1$ and $P3$) 2 stage after A's stage (1 in this example). In this case, the number of buffers needed in $P3$ decreases to 3, but 1 more buffer from buffer group 1 remains in $P1$'s local store. The benefit of this approach is that the new DMA can be executed in parallel with A, eliminating the possibility of increasing the workload of $P1$. Each of these approaches has its own benefit(memory usage vs. performance) and the buffer allocation algorithm chooses between them based on the size of the local stores and workload of each victim processor.

4. EXPERIMENTS

We evaluated Flextream using a heterogeneous multicore simulator that we have built. We also leveraged the StreamIt compiler as a starting point for implementing our heuristics and used Metis [11] for graph partitioning. For the evaluation and results, we simulated a multicore system with 32 slave cores and one master core. The master core is similar to a PowerPC processor running at 2GHZ with a 32KB L1 and a 1MB L2 cache. Each slave core includes a local store for instructions and data, and a memory flow control (MFC) unit that performs DMA operations to and from the local stores independent of the slave cores.

Performance Comparison: We first compare the performance achieved using Flextream to that achieved using online whole-program graph partitioning. The graph partitioner uses the work estimate of the actors as the node weights, and the communication costs as the edge weights. We perform prepass replication for both approaches. In this experiment, we measure the performance degradation caused by either strategy, compared to the optimal schedule. We use benchmarks drawn from the StreamIt benchmark suite. Each benchmark is run 31 times, and in each run $1 < i \leq 32$, the total number of processors starts at 32 cores, and is subsequently reduced to a smaller number of cores equal to i. The average slowdown per benchmark is shown in Figure 9. Flextream is 9% worse than then the performance achieved using an optimal schedule, but 8% better than applying graph partitioning at runtime. The main reason for Flextream's performance edge is that Flextream leverages the optimal scheduling solution found by the static compilation phase.

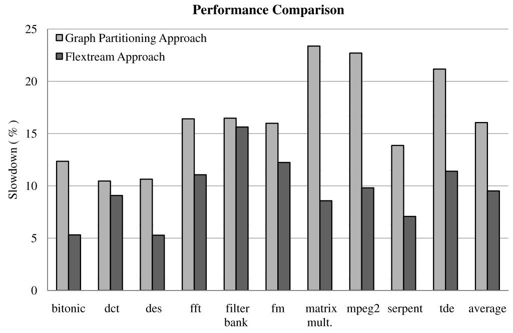


Figure 9: This graph shows performance degradation when online adaptation is carried out using two different strategies.

Figure 10 compares the average time that Flextream's partition refinement step needs to generate a new processor mapping to the time taken by the graph partitioning approach. On average, Flextream's approach is 50%(3ms) faster than the graph partitioning approach. The results suggest that Flextream is a superior strategy to repartitioning, considering that the scheduling solutions are derived faster and yield better performance. It is also worthy to note that the runtime overheads are likely to be very high in the absence of good starting solutions. The combination of static compilation (ILP and prepass fission) and dynamic adaptation is an attractive combination that combines the benefits of static and dynamic paradigms.

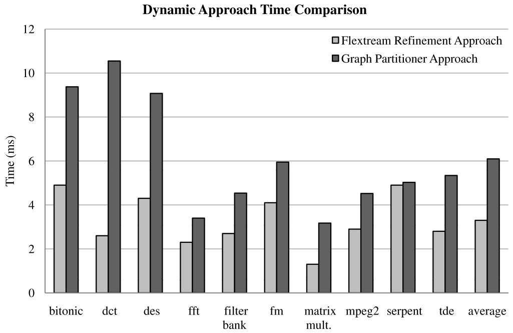


Figure 10: This graph illustrates the amount of time Flextream's partition refinement takes and compares it with the graph partitioning approach.

Overhead: We measured the overhead associated with each Flextream phase. Figure 11 illustrates the relative and absolute values of the times taken by each phase. We exclude from this graph the time taken to perform work partitioning since it can take several minutes for the work partitioning to find a valid ILP solution. Each of the bars in the figure include a label that represents the absolute time (in milliseconds) taken by that phase. The prepass replication requires 1283ms and is significantly longer than the time taken by the other 3 *online* phases (notice that the Y-axis starts at 90%). Among the online phases, stage assignment is the longest, followed by buffer allocation and work refinement. Most of the overhead for stage assignment is due to the topological traversal of the graph. The results indicate that the time spent in prepass replication is proportional to the size of the application (graph). Overall, this experiment supports the hypothesis that performing online adaptation using Flextream is an efficient endeavor.

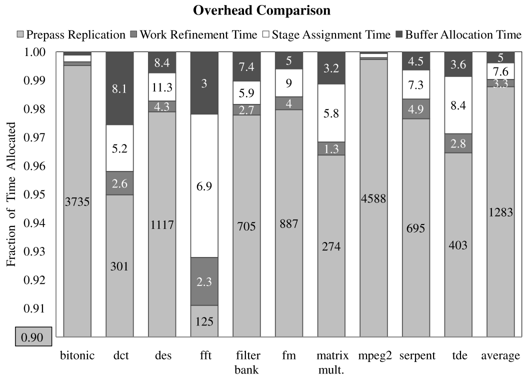


Figure 11: Flextream overhead categorized by phases. Each bar has 4 parts, showing the relative (Y-axis) and absolute (labels within the bars) times spent in each of the static and online phases. Note that the Y-axis starts at 90%.

Buffer Allocation: Buffer allocation is the last Flextream phase. This step can lead to new DMA requests and can increase the processor workloads. Buffer allocation attempts to maximize the use of the local store in order to avoid the long latencies associated with accessing main memory. The graph in Figure 12 shows how this optimization impacts overall performance. For this experiment the number of processors is changed at runtime from 32 cores to 8. We gradually decrease the size of the local store, starting at `Max Mem` which is large enough to ensure that no spilling occurs. This experiment shows the effectiveness of the buffer allocation algorithm in using local stores. As expected, the performance degrades when the size of the local store is reduced. The buffer allocator uses the local stores until it exhausts their capacity, at which point it has only one recourse, and that is to use main memory. For some benchmarks, reducing the local store capacity has negligible impact (e.g., `mpeg2`) because new DMA requests are added to the processors that have less work according to the original schedule (before buffer allocation). In other words, the overhead of the new DMA operations do not increase the size of the maximum workload.

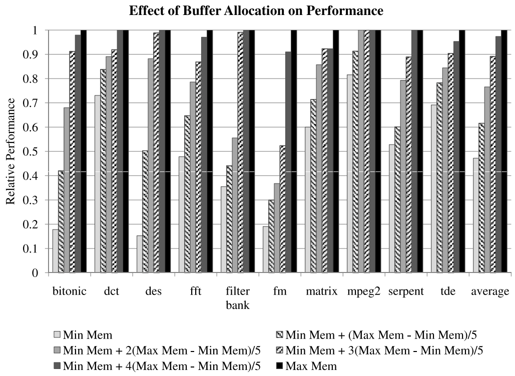


Figure 12: Effect of buffer allocation on benchmark throughput. For each benchmark, the amount of memory is increased from a minimum to a maximum capacity. Throughput is recorded for 6 uniformly distributed memory sizes per benchmark.

A Full Runtime Scenario: We also carried out an experiment to demonstrate how Flextream might perform in a real scenario where resource availability changes multiple times at runtime. Each time the the number of available cores changes, a new schedule is generated using the online adaptation mechanism. The extant schedule is drained and the new schedule is communicated to the slave processors using the multicore stream layer (see section 2.2). The adaptation overhead therefore is the sum of the time taken by each of these steps. Among all of the benchmarks, the maximum overhead for sending commands is 11 micro seconds. This assumes the overhead for sending each command is 20 cycles.

	Drain(ms)	Adaptation(ms)	1K sec-Flextream	1K sec-Static
bitonic sort	6.14	89.42	350M	356M
dct	0.79	42.80	380 M	452 M
des	32.39	113.80	148 M	150 M
fft	2.37	142.95	222 M	230 M
filter bank	0.44	142.95	448 M	490 M
fm	2.16	65.71	133 M	143 M
matrix mult.	3.07	37.19	62 M	71 M
mpeg2	4619	43	156 K	177 K
serpent	81.11	79.09	52 M	54 M
tde	780	66.08	1.2 M	1.3 M

Table 1: Performance comparison between Flextream and optimal for a runtime scenario in which number of cores varies in this order: 32, 16, 10, 6. Each configuration runs for 250 seconds.

Table 1 compares the performance of our approach with the theoretical optimal in a scenario where the number of available cores at runtime changes from 32 to 16, then to 10, and finally to 6. We assume each configuration runs for 250 seconds, for a total processing time of 1000 seconds. The theoretical optimal solution, for each runtime configuration, uses a schedule found by the static phase. The first column shows the total time needed to drain the schedules. The overhead of the online adaptation is shown in the second column. The last two columns show how many iterations of each stream graph can be executed using Flextream versus the optimal approaches. The largest difference between the last two columns occurs in `dct` which loses 16% of its throughput when using Flextream. The best performing benchmarks are `bitonic sort` and `serpent`, losing only 3% of their throughput compared to optimal. Overall, these results imply that solutions found by Flextream, in real execution scenarios, can perform close to theoretical optimal solutions.

5. RELATED WORK

There is a large body of literature that deals with exploiting parallelism in streaming codes for better performance. The most recent and relevant works include compilation of new streaming languages such as StreamIt, Brook, StreamC/KernelC, and Cg to multicores or data-parallel architectures. For example, Gordon et al. [5] and [6] perform stream graph refinements to statically determine the best mapping of a StreamIt program to a multicore like the one we consider in this paper. Kudlur and Mahlke apply modulo scheduling to an unrefined stream graph to maximize throughput [12]. Liao et al. apply classic affine partitioning techniques to exploit properties of stream operators [17]. There is also a rich history of scheduling and resource allocation techniques developed in Ptolemy that make fundamental contributions to stream-scheduling (e.g., [15, 8]). Flextream is unique relative to these past contributions in its ability to dynamically adapt a static schedule and resource allocation to changes in available resource at runtime. Viewed in this way, Flextream is complimentary to some static scheduling techniques, and can be applied more generally as long as we can extract a graph-representation of the streaming computation.

In contrast to static compilation techniques, there are also many existing ideas related to compilation for multicores. In [7], the authors dynamically map an abstract representation of a stream program [13] to threads that can execute in parallel on a general purpose multiprocessor. In CellSs [1], computation is expressed as functions that may be composed to form a dataflow graph. A runtime scheduler treats this graph in the same way a superscalar processor treats operations, and schedules these functions onto the Cell cores as soon as their inputs are ready. In [2], the authors describe an application-specific parallelization strategy that they applied manually. They were able to target for various configurations of the Cell architecture, which varied the number of cores in each configuration. Our work is distinctly different from these works in that we use a static compile-time schedule to automatically perform dynamic optimizations that lead to new and efficient resource allocations.

Adaptive compilation to a virtualized system is not an entirely new idea. Recent examples include Veal [4] and Liquid SIMD [3]. The authors in these works take similar approaches to the one in this work but in very different domains than the one we address in this work. In [4], adaptive loop modulo scheduling is performed for a virtualized loop accelerator system. The authors in [3] propose hybrid compilation techniques for mapping a vectorizable program to SIMD engines that have different vector lengths.

6. CONCLUSION

In this work, we focus on the increasingly important area of streaming computing and introduce Flextream as a flexible compilation framework that can dynamically adapt applications to the changing characteristics of the underlying architecture. This is an important contribution as software developers grapple with the details of parallelism in a rapidly changing architecture landscape. The main innovation in Flextream is an adaptive stream graph modulo scheduling algorithm that combines the benefits of static scheduling with the advantages of dynamic adaptation. Our results indicate that Flextream's approach can achieve high-performance resource allocations that are within an average of 9% compared the optimal solutions with low overhead.

Acknowledgement

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. We owe thanks to Mark Woh, and Shuguang Feng who went to great lengths in assisting with preliminary drafts of this work. This research was supported by ARM Limited. This work was also supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-356-D00200).

7. REFERENCES

[1] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell be architecture. 00(1):5, 2006.

[2] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *Proc. 12th PPoPP*, pages 90–100, New York, NY, USA, 2007. ACM Press.

[3] N. Clark et al. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proc. 13th HPCA*, pages 216–227, 2007.

[4] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. 35th ISCA*, pages 389–400, June 2008.

[5] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *10th ASPLOS*, pages 291–303, Oct. 2002.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th ASPLOS*, pages 151–162, 2006.

[7] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proc. 38th MICRO*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[8] S. Ha and E. A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *Trans. on Computers*, 40(11):1225–1238, 1991.

[9] H. P. Hofstee. Power efficient processor design and the Cell processor. In *Proc. 11th HPCA*, pages 258–262, Feb. 2005.

[10] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A multilevel computing architecture for embedded multimedia applications. *IEEE Micro*, 24(3):56–66, 2004.

[11] 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, Sept. 1998.

[12] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. '08 PLDI*, pages 114–124, June 2008.

[13] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. In *Proc. 13th PACT*, pages 267–277, 2004.

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

[15] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Technical Report UCB/ERL M95/36, University of California, Berkeley, May 1995.

[16] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 02 CC*, pages 179–196, 2002.

[17] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. *Proc. 2006 CGO*, 0(1):196–207, 2006.

[18] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *ACM SIGARCH Computer Architecture News*, 36(2):18–27, 2008.